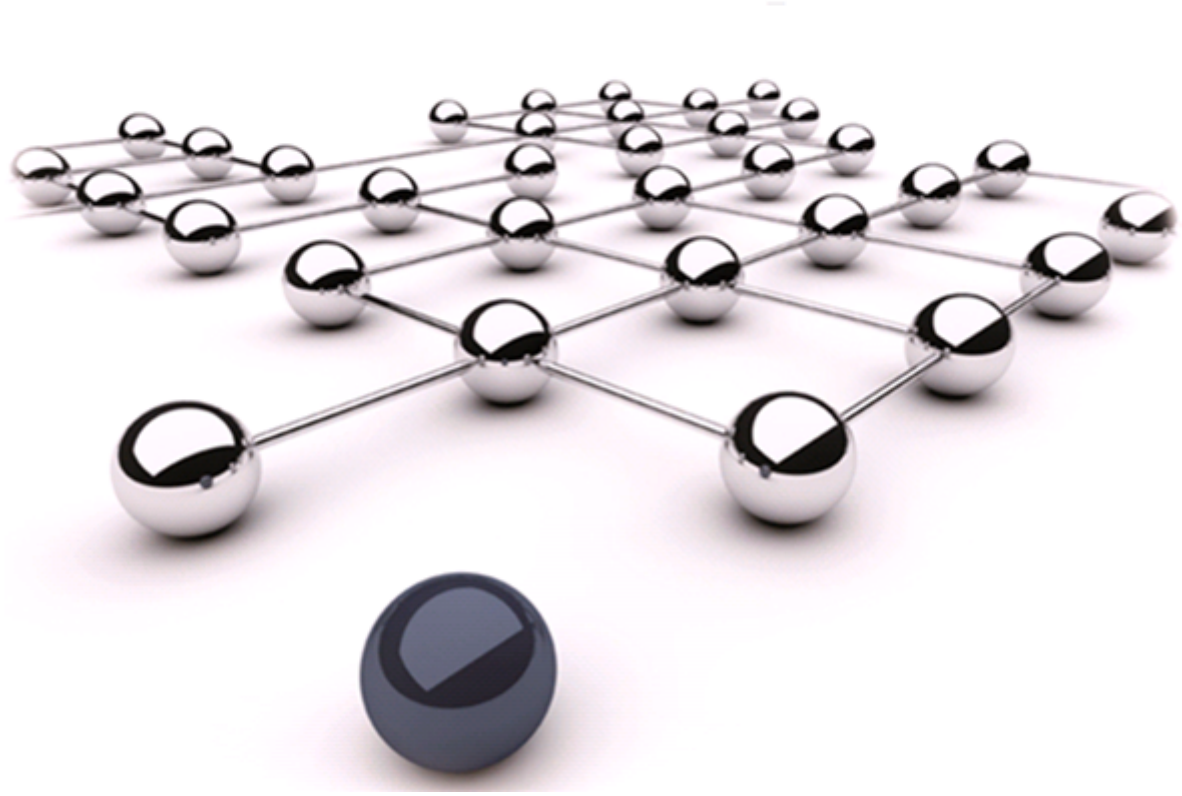
OPPIDA

EXPERT EN SÉCURITÉ
DES SYSTÈMES D'INFORMATION



Challenge SSTIC 2015

Auteur :
Damien Millescamps

Référence : OPPIDA/DOC/2015/INT/716/1.0

17/04/2015

Table des matières

1 Introduction

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

La solution présentée ici se déroule en 5 étapes principales, appelés "stage" dans le challenge, ainsi que d'un prologue et d'un épilogue.

L'ensemble du code présenté ici pourra être mis à disposition. Probablement par le biais de `github`.

2 Prologue

2.1 Outils

- `mount`
- `hte`

2.2 Inspection du système de fichier

2.2.1 Montage du disque

Il est possible de directement monter l'image disque par :

```
mount -o loop sdcard.img /mnt/sdcard
```

Le système de fichier est au format FAT16, et ne contient qu'un seul fichier : `inject.bin`.

2.2.2 Recherche des fichiers effacés

En éditant le fichier sdcard dans un éditeur hexadécimal, à partir de l'offset contenant le "root directory", on s'aperçoit rapidement qu'un fichier est effacé (octet 0xE5 en préfixe de son entrée). Le nom complet du fichier se retrouve dans l'entrée LFE (Long Format Entry), qui contrairement à l'entrée 8.3 ne perd pas l'information du nom de fichier en cas d'effacement. Le fichier est `build.sh`, et il se trouve dans le cluster 2.
Toujours à l'aide de l'éditeur hexadécimal, il est possible de restaurer le fichier avec les modifications suivantes sur l'image de la microSD :

```
--- sdcard.img
+++ sdcard_recover.img
@@0x00000200
-F8 FF FF FF 00 00 00 00 05 00 06 00 07 00 08 00 |................|
+F8 FF FF FF 00 00 FF FF 05 00 06 00 07 00 08 00 |................|
--- sdcard.img
+++ sdcard_recover.img
@@0x0001EA00
-F8 FF FF FF 00 00 00 00 05 00 06 00 07 00 08 00 |................|
+F8 FF FF FF 00 00 FF FF 05 00 06 00 07 00 08 00 |................|
--- sdcard.img
+++ sdcard_recover.img
@@0x0003D200
-E5 62 00 75 00 69 00 6C 00 64 00 0F 00 BD 2E 00 |.b.u.i.l.d......|
+41 62 00 75 00 69 00 6C 00 64 00 0F 00 BD 2E 00 |Ab.u.i.l.d......|
--- sdcard.img
+++ sdcard_recover.img
@@0x0003D220
-E5 55 49 4C 44 20 20 20 53 48 20 20 00 00 2D 1E |.UILD...SH....-.|
+42 55 49 4C 44 20 20 20 53 48 20 20 00 00 2D 1E |BUILD...SH....-.|
```

Si on remonte le système de fichier après ces modifications, on pourra copier `build.sh`. Ce fichier contient la commande suivante :

```
java -jar encoder.jar -i /tmp/duckyscript.txt
```

3 USB Rubber Ducky

3.1 Outils

- egrep
- iconv
- base64
- gcc

3.2 Décodeur Rubber Ducky

Le fichier restauré à l'étape précédente contient un indice pour la suite. Le binaire `inject.bin` a été généré à partir d'un fichier source `/tmp/duckyscript.txt`. C'est donc très probablement le résultat du codeur pour USB Rubber Ducky. Le binaire contient alors des codes de touches clavier, avec leurs modificateurs (shift, control, etc.).
Il est possible de décoder le binaire avec l'outil `ducky-decode.pl` que l'on peut retrouver dans la suite logiciel de Rubber Ducky, mais sa sortie n'est pas directement utilisable.
À la place, on peut écrire notre propre décodeur. Il n'y a besoin de gérer qu'un sous ensemble des codes :

- Modificateurs : Super, Shift
- Touches : "A-Z"/"a-z", "-"/"_", "="/"+", "0-9", " ", "\n", "\b", "\t"

Le but est d'obtenir une sortie directement utilisable pour pouvoir être traitée.

```
$ make decode
$ ./decode ../prologue/inject.bin >duckyscript.txt
```

La sortie est une série de commandes sous la forme `powershell -enc $BASE64` :

```
// wait 2000 ms
GUI r
// wait 500 ms
?
// wait 1000 ms
cmd
// wait 50 ms
powershell -enc ZgB1AG4AYwB0AGkAbwBuACAAdwByAGkAdABlAF8AZgBpAGwAZQBfAGIAeQB0AGUAcwB7AHAAYQB[...]
// wait 10 ms
powershell -enc ZgB1AG4AYwB0AGkAbwBuACAAdwByAGkAdABlAF8AZgBpAGwAZQBfAGIAeQB0AGUAcwB7AHAAYQB[...]
[...]
```

La sortie de la base64 donne des commandes et déclarations de fonctions powershell encodées en UTF16LE. Pour faciliter la suite, il est préférable de convertir le tout directement en UTF8 :

```
grep powershell duckyscript.txt | cut -d' ' -f3 | base64 -d | iconv -f utf-16 -t utf-8 > script.
    psh
```

Le script récupère le nom d'utilisateur et le répertoire courant de l'environnement. En fonction du résultat, une des deux chaînes codées en base64 sera décodée et ajoutée à la fin d'un fichier. L'une de ces chaînes est constantes entre toutes les invocations de powershell et une fois décodée donne : "TryHarder".
La dernière invocation de powershell par le Ducky Script sert à vérifier le résultat du hachage du fichier de sortie. La fonction utilisée est SHA1, et le résultat doit donner : "EA-9B-8A-6F-5B-52-7E-72-65-20-19-31-3C-25-B5-6A-D2-7C-7E-C6".

```
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;
    $e -eq "challenge2015sstic";
}
if (check_correct_environment) {
    write_file_bytes([Convert]::FromBase64String('UEsDBAoDAAAAADa[...]dXtOh6gUsBzWnXw=='));
} else {
    write_file_bytes([Convert]::FromBase64String('VAByAHkASABhAHIAZABlAHIA'));
}
[...]
function hash_file
{
    param([string]$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";
}
```

Afin de sortir la bonne chaîne codée en base64, le script vérifie que l'environnement est, par exemple :

- utilisateur = "sstic"
- répertoire courant = "challenge2015"

Dans ce cas, le contenu en base64 est décodé, et la sortie doit donner le fichier : .\stage2.zip, dont le résultat du hachage SHA1 est : "EA-9B-8A-6F-5B-52-7E-72-65-20-19-31-3C-25-B5-6A-D2-7C-7E-C6"

Finalement, la commande suivante permet d'extraire le fichier stage2.zip du script powershell :

```
$ egrep -o "correct_environment\){write_file_bytes\(\[Convert\]::FromBase64String\('[^']+"
    script.psh  | cut -d"'" -f2 | base64 -d >stage2.zip
$ sha1sum stage2.zip
ea9b8a6f5b527e72652019313c25b56ad27c7ec6  stage2.zip
```

3.3 En résumé

```
make decode && ./decode ../intro/inject.bin | grep powershell | cut -d' ' -f3 | base64 -d |
    iconv -f utf-16 -t utf-8 | egrep -o "correct_environment\){write_file_bytes\(\[Convert\]::
    FromBase64String\('[^']+" | cut -d"'" -f2 | base64 -d >stage2.zip && sha1sum stage2.zip
```

4 OpenArena

4.1 Outils

- `OpenSSL`
- Autres :
  - `OpenArena` (avec support \devmap)

4.2 Contenu de l'archive récupérée au stage précédent

- `memo.txt`
- `sstic.pk3`
- `encrypted`

4.3 Analyse du contenu

Le fichier `memo.txt` contient l'énoncé suivant :

```
Cipher: AES-OFB
IV: 0x53535449433230313 52d537461676532
Key: Damn... I ALWAYS forget it. Fortunately I found a way to hide it into my favorite game !

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

4.3.1 Analyse de la carte OpenArena

Le fichier `sstic.pk3` est un fichier de carte pour OpenArena. Le format des .pk3 est en fait une archive Zip. Une fois décompressé, cela permet d'obtenir les textures ainsi que la carte au format BSP. La carte est une modification d'une déjà existante : `085am_underworks2`.

Deux opérations peuvent s'avérer utiles pour la suite :
- Parcourir les textures à la recherche d'indices
- Chercher la partie scriptée du jeu via un `strings` sur le fichier BSP.

Les textures Dans les ressources on retrouve les textures sous forme d'image.

Certaines images paraissent intéressantes pour la suite. Elles contiennent :
- Une icône, il y en a 8 différentes : drapeau, signal, disquette, goutte, maillon, wifi, téléviseur et soleil.
- Trois mots de 32 bits, aux couleurs : verte, orange et blanche.

Pour chacune des icônes disponibles, il existe 10 images associées. Même si un bruteforce ne semble pas envisageable ($A_n^k = \frac{n!}{(n-k)!} \approx 10^{19}, n = 240, k = 8$), la recopie (sans erreurs) de chacun de ces mots inscrits serait de toute façon bien trop fastidieuse.

Le script La phrase suivante est présente dans le fichier BSP : “Yes !\n You found my key !”. Après analyse rapide, il semblerait qu'un interrupteur permette de déclencher un téléporteur qui permet d'arriver dans une zone : “Secret Arena”. Cette zone contient un autre interrupteur déclenchant lui aussi un téléporteur qui permet d'arriver dans une salle dont les coordonnées sont en dehors de la carte d'origine.

4.4 Le jeu

Le chargement de la carte peut se faire avec la commande '\devmap sstic' dans la console. Pour de l'exploration, les commandes “/god” et “/noclip” peuvent être pratiques. Cependant, il est plus ludique de jouer pour trouver les informations qui nous permettront de reconstituer la clef.
Pour trouver la salle secrète affichant le message “Yes !\n You found my key !”, les étapes suivantes sont nécessaires :

- Activer l'interrupteur de la salle avec l'ordinateur
- Activer l'interrupteur qui déclenche le message “secret arena” derrière le panneau du SSTIC dans la salle principale
- Pour franchir le passage de lave, les plus adroits pourront le faire grâce à un “rocket jump”, sinon le god mode peut être une alternative plus rapide
- Un téléporteur est présent après le passage de lave, il ne faut le contourner ou passer au dessus grâce à un rocket jump
- Finalement, il faut activer l'interrupteur sur un panneau au dessus du joueur pour arriver dans le salle secrète

On y trouve un indice sur le format de la clef, qui est donnée par des icônes associés a une couleur :

On a donc considérablement réduit l'espace de recherche. Il reste cependant 10 images possibles pour chaque icône. Dans la carte, pour chaque icône, une seule image possible est visible, on peut donc en déduire que c'est l'image contenant les éléments de clef à utiliser. On peut partir à la recherche de ces images dans le jeu, on y trouve :

- Soleil, sur le mur de la première salle (niveau 0)
- Goutte, sur les caisses première salle (niveau 0)
- Drapeau, sous l'escalier (niveau -1)
- Signal, sur la caisse flottante (niveau 1)
- Maillon, derrière le panneau marqué d'un œil (niveau -1), qui s'ouvre par un interrupteur que l'on trouve sur un pilier au niveau 1
- Disquette, derrière le gros bloc dans un coin (niveau 0)
- Téléviseur, sur l'ordinateur dans la cage, visible de dessus (niveau 0)

L'image contenant le symbole WiFi n'a pas été trouvée, et n'est probablement pas présente.

4.5 La clef

On peut maintenant reconstituer la clef avec les images visibles :

Drapeau	Vert	`9e2f31f7`
Signal	Blanc	`8153296b`
Disquette	Orange	`3d9b0ba6`
Goutte	Blanc	`7695dc7c`
Drapeau	Orange	`b0daf152`
Maillon	Vert	`b54cdc34`
WiFi	Vert	`????????`
Téléviseur	Blanc	`26609fac`

Pour l'icône WiFi, la liste extraite pour la couleur verte depuis le répertoire de texture donne :

- `ffe0d355`
- `ca42dc3b`
- `90830de3`
- `2281d257`

- 45239f7f
- e600f08f
- df0574cf
- ca42dc3b
- 3f8f3287
- 2d2bb194

Le premier mot de 32 bits s'avèrera être le bon, formant la clef :

```
9e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac
```

4.6 Déchiffrement du fichier

Pour construire la commande OpenSSL, l'algorithme peut être sélectionné avec "-aes-256-ofb" et l'IV et la clef peuvent être fournis avec "-iv" et "-K". Pour une raison inconnue (mais dont on se doute), OpenSSL ne retire pas le padding (qui est de la taille d'un bloc, ce qui peut expliquer) malgré l'absence de l'option "-nopad", il faut donc utiliser `dd` afin d'obtenir un résultat de la fonction de hachage SHA256 valide.

```
$ openssl enc -d -aes-256-ofb -iv 53535449433230313 52d537461676532 -K 9
    e2f31f78153296b3d9b0ba67695dc7cb0daf152b54cdc34ffe0d35526609fac -in encrypted | dd of=stage3
    .zip bs=256 count=$((0x7a5)) 2>/dev/null
$ sha256sum stage3.zip
845f8b000f70597cf55720350454f6f3af3420d8d038bb14ce74d6f4ac5b9187  stage3.zip
```

5 Paint

5.1 Outils

- `wireshark`
- `gnuplot`
- `gcc`
- Autres :
  - `blake_ref.c`, implémentation de référence de BLAKE
  - "floppy1" de la soumission officielle de l'algorithme Serpent

5.2 Contenu de l'archive récupérée au stage précédent

- `paint.cap`
- `memo.txt`
- `encrypted`

5.3 Analyse du contenu

Le fichier `memo.txt` contient l'énoncé suivant :

```
Cipher: Serpent-1-CBC-With-CTS
IV: 0x53535449433230313352d537461676533
Key: Well, definitely can't remember it... So this time I securely stored it with Paint.

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

5.4 Analyse du fichier PCAP

Après ouverture de `paint.pcap` dans wireshark, on y retrouve une communication de souris USB. Cette information se retrouve grâce aux premiers échanges pendant lequel le péripherique USB s'identifie. Le protocole des souris USB est assez basique. La communication se fait par interruption, et les données sont sur 4 octets sous la forme suivante :

```
struct usbmouse {
    unsigned char button1:1;
    unsigned char button2:1;
    unsigned char button3:1;
    unsigned char button4:1;
    unsigned char button5:1;
    unsigned char rfu:3;
    char          x;
    char          y;
    char          wheel;
};
```

L'indice donné dans `memo.txt` permet de savoir que la communication a été enregistrée pendant une session sous Paint. La démarche pour la suite va donc consister à reproduire l'image qui a pu être dessinée pendant la session.

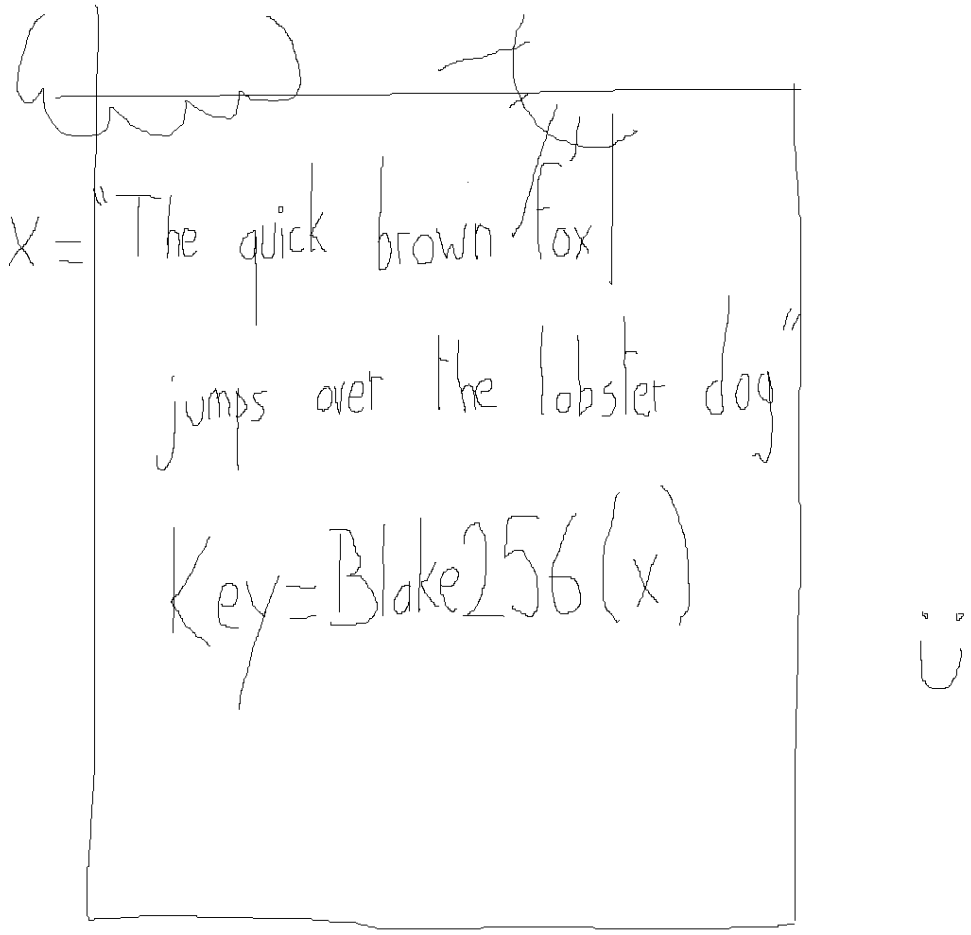
5.5 Reconstruction de l'image Paint

Afin de reconstruire l'image, la méthode consiste à :
- Lire le fichier `PCAP` à la recherche des données envoyées de la souris USB
- Traquer la position de la souris à chaque mise à jour de position
- Générer une liste de segments correspondant aux moments pendant lesquels le bouton 1 est enclenché
- Afficher les segments à l'aide d'un `plotter`

Pour reconstruire une image au format PNG, un premier code a été développé afin de générer un fichier `gnuplot` :

```
$ make parse_mouse_pcap && ./parse_mouse_pcap paint.cap | gnuplot > image.png
```

Le résultat donne une image au format PNG :



L'image contient l'information suivante :

```
x = ''The quick brown fox jumps over the lobster dog''
key = Blake256(x)
```

5.6 Calcul de la clef

Grâce a l'image précédente, on sait comment calculer la clef. L'implémentation de référence de BLAKE a été utilisée pour effectuer ce calcul. Le code est disponible a cette adresse : `https://131002.net/blake/`.
Pour obtenir la clef, l'implémentation rapide d'une fonction main s'impose :

```
BitSequence hash[64];
Hash(256, av[1], strlen(av[1]) * 8, hash);
for(i = 0; i < 32; i++) {
    printf("%02x", hash[i]);
}
printf("\n");
```

On peut finalement obtenir le résultat de la fonction de hachage qui correspond à la clef :

```
$ make -C blake blake_ref && ./blake/blake_ref "The quick brown fox jumps over the lobster dog"
66c1ba5e8ca29a8ab6c105a9be9e75fe0ba07997a839ffeae9700b00b7269c8d
```

5.7 Déchiffrement du fichier

L'algorithme utilisé pour le chiffrement du fichier est Serpent-1-CBC-With-CTS. Le Serpent-CBC n'étant pas supporté par OpenSSL, l'implémentation de référence de Serpent a été utilisée. Le code de référence peut se trouver à cette adresse : `http://kheldar.insomnia247.nl/floppy1/`. L'utilisation de cette implémentation n'était probablement pas une si bonne idée. L'utilisation abusive de `long` pour typer des mots de 32 bits rend le code non portable sur une plateforme autre que 32 bits. De plus, les fonctions fournies pour transformer l'ASCII en tableau d'octet considèrent forcément du little-endian, ce qui a pour effet d'inverser l'ordre des octets dans la clef et pour le vecteur d'initialisation. A noter aussi que cette implémentation est extrêmement lente. Finalement, pour la partie CTS (CipherText Stealing), il faudra rajouter un peu de code.

En partant du code de la soumission officielle de l'algorithme, on peut récupérer la partie propre à l'implémentation de référence :

- serpent-api.h
- serpent-tables.h
- serpent-reference.c
- serpent-reference.h
- serpent-aux.c
- serpent-aux.h

Il faut ensuite rajouter le code pour gérer le TCS. Pour ce faire, `mmap` a été utilisé afin de "mapper" le ficher à déchiffrer en mémoire, et ainsi pouvoir modifier facilement les deux derniers blocs. Le code propre au TCS est le suivant :

```
buffer = (char *)mmap(NULL, sstat.st_size + BYTES_PER_BLOCK, PROT_READ|PROT_WRITE, MAP_PRIVATE,
    fd, 0);

lastbloc = (sstat.st_size / BYTES_PER_BLOCK);
pad_len = (lastbloc + 1) * BYTES_PER_BLOCK - sstat.st_size;

/* CTS implementation */
makeKey(&key, DIR_DECRYPT, 256, "8
    d9c26b7000b70e9eaff39a89779a00bfe759ebea905c1b68a9aa28c5ebac166");
cipherInit(&cipher, MODE_ECB, 0);

memcpy(bufIN, &buffer[(lastbloc - 1) * BYTES_PER_BLOCK], BYTES_PER_BLOCK);
blockDecrypt(&cipher, &key, (BYTE*) bufIN, BITS_PER_BLOCK, (BYTE*) bufOUT);
memcpy(bufIN, &buffer[lastbloc * BYTES_PER_BLOCK], BYTES_PER_BLOCK - pad_len);
memcpy(bufIN + BYTES_PER_BLOCK - pad_len, bufOUT + BYTES_PER_BLOCK - pad_len, pad_len);
memcpy(&buffer[lastbloc * BYTES_PER_BLOCK], &buffer[(lastbloc - 1) * BYTES_PER_BLOCK],
    BYTES_PER_BLOCK);
memcpy(&buffer[(lastbloc - 1) * BYTES_PER_BLOCK], bufIN, BYTES_PER_BLOCK);
```

Le déchiffrement en CBC se poursuit ensuite normalement :

```
makeKey(&key, DIR_DECRYPT, 256, "8
    d9c26b7000b70e9eaff39a89779a00bfe759ebea905c1b68a9aa28c5ebac166");
cipherInit(&cipher, MODE_CBC, "3365676174532d3531303243495545353");

for (bl = 0; bl < lastbloc + 1; bl++) {
    result = blockDecrypt(&cipher, &key, (BYTE*) &buffer[BYTES_PER_BLOCK * bl], BITS_PER_BLOCK,
    (BYTE*) bufOUT);
    if (result < 0) {
        printf("blockDecrypt error %d\n", result);
        goto err;
    }
    if (bl == lastbloc) {
        fwrite(bufOUT, BYTES_PER_BLOCK - pad_len, 1, fs);
    } else {
        fwrite(bufOUT, BYTES_PER_BLOCK, 1, fs);
    }
}
```

Le déchiffrement complet du fichier peut alors se faire :

```
$ make -C serpent && ./serpent/serpent_cbc_tcs encrypted stage4.zip
$ sha256sum stage4.zip
7beabe40888fbbf3f8ff8f4ee826bb371c596dd0cebe0796d2dae9f9868dd2d2  stage4.zip
```

6 JSFuck

6.1 Outils

- sed
- egrep
- xxd
- bash
- openssl
- Firefox Developer Edition

6.2 Contenu de l'archive récupérée au stage précédent

- stage4.html

6.3 Analyse du contenu

Le fichier HTML contient du javascript. Deux variables sont définies, "data" et "hash". Le script JS semble avoir été généré par un obfuscateur se basant sur le même principe que JSFuck. Afin de l'obtenir sous une forme plus lisible, la console de Firefox va s'avérer utile pour résoudre les chaînes de caractères formées pour générer le javascript valide.

6.4 La méthode d'obfuscation

Afin de retirer les caractères alphanumériques du script , la technique consiste a effectuer des opérations dont le retour, sous forme de chaîne de caractères, pourra être traité comme un tableau de caractères qu'il est possible d'indexer pour reconstruire des mots du langage. Il y a 4 chaînes de caractères principales utilisées par l'obfuscateur. A l'aide de la console de Firefox, on peut récupérer leurs valeurs :

```
print(![] + "")
"false"
print({} + "")
"[object Object]"
print(!"" + "")
"true"
print($[$] + "")
"undefined"
```

Pour l'index, il faut pouvoir récupérer un nombre, ce qui est possible grâce a :

```
print(~[])
"-1"
```

Afin de générer une fonction, la technique utilisée consiste à prendre un objet, puis à appeler le constructeur de son constructeur. Cela renvoie un objet `Function()`, par exemple :

```
function myfunc { function_definition };
```

peut aussi s'écrire sous la forme :

```
myfunc = "0".constructor().constructor(return "function_definition");
```

Finalement, les appels de fonctions se font par obj["funcion"]() qui produit le même résultat que obj.function() :

```
myfunc = 0["constructor"]["constructor"](return "function_definition");
```

Évidemment, les noms de variables sont tous obfusqués. La proximité de leur nom rend leur substitution automatique plus ardue puisqu'il faut le faire dans un certain ordre afin de ne pas substituer les mauvaises variables.

6.5 Analyse du "constructeur"

Une méthode qui ne nécessite pas de compétences particulières en javascript consiste à indenter le script avant de le soumettre au debugger de Firefox. On s'aperçoit assez rapidement que la première partie du script est relativement courte :

```
$ = ~[];
$ = {
    ___: ++$,
    $$$$: (![] + "")[$],
    __$: ++$,
    $_$_: (![] + "")[$],
    _$_: ++$,
    $_$$: ({} + "")[$],
    $$_$: ($[$] + "")[$],
    _$$: ++$,
    $$$_: (!"" + "")[$],
    $__: ++$,
    $_$: ++$,
    $$__: ({} + "")[$],
    $$_: ++$,
    $$$: ++$,
    $___: ++$,
    $__$: ++$
};

$.$_ = ($.$_ = $ + "")[$.$_$]
     + ($._$ = $.$_[$.__$])
     + ($.$$ = ($.$ + "")[$.__$])
     + ((!$) + "")[$._$$]
     + ($.__ = $.$_[$.$$_])
     + ($.$ = (!"" + "")[$.__$])
     + ($._ = (!"" + "")[$._$_])
     + $.$_[$.$_$]
     + $.__
     + $._$
     + $.$;

$.$$ = $.$
     + (!"" + "")[$._$$]
     + $.__
     + $._
     + $.$
     + $.$$;

$.$ = ($.___)[$.$_][$.$_];

$.$(JS_BLOB)();
```

La partie `JS_BLOB` contient le script JS lui-même. Afin de le rendre plus lisible, il faut substituer les variables calculées dans le "constructeur" par leurs valeurs. Leurs valeurs peuvent se retrouver en statique à partir des observations faites précédemment, ou à l'aide du debugger de Firefox. Cela donne l'ensemble de valeurs suivantes :

```
    $.___          0
    $.$$$$         "f"
    $.__$          1
    $.$_$_         "a"
    $._$_          2
    $.$_$$         "b"
    $.$$_$         "d"
    $._$$          3
    $.$$$_         "e"
    $.$__          4
    $.$_$          5
    $.$$__         "c"
    $.$$_          6
    $.$$$          7
    $.$___         8
    $.$__$         9

    $._$           "o"
    $.__           "t"
    $._            "u"

    $.$_           "constructor"
    $.$$           "return"
    $.$            Function()
```

En utilisant `egrep` et `sed`, on peut substituer ces variables par leurs valeurs. Évidemment, l'ordre des `sed` est important pour ne pas substituer une variable dont le début du nom correspondrait à un autre nom de variable :

```
egrep -o '\$\.\$\(.*;' stage4.html |\
    sed 's/\$\.\$\$_\$/"d"/g' |\
    sed 's/\$\.\$\$\$_/"e"/g' |\
    sed 's/\$\.\$\$__/"c"/g' |\
    sed 's/\$\.\$\$_/6/g' |\
    sed 's/\$\.\$_\$\$/"b"/g' |\
    sed 's/\$\.\$_\$_/"a"/g' |\
    sed 's/\$\.\$_\$/5/g' |\
    sed 's/\$\.\$___/8/g' |\
    sed 's/\$\.\$__\$/9/g' |\
    sed 's/\$\.\$__/4/g' |\
    sed 's/\$\.\$_/"constructor "/g' |\
    sed 's/\$\.\$\$\$\$\$/"f"/g' |\
    sed 's/\$\.\$\$\$\$/7/g' |\
    sed 's/\$\.\$\$/"return "/g' |\
    sed 's/\$\.\$/Function/g' |\
    sed 's/\$\._\$\$/3/g' |\
    sed 's/\$\._\$_/2/g' |\
    sed 's/\$\.__\$/1/g' |\
    sed 's/\$\.___/0/g' |\
    sed 's/\$\.__/"t"/g' |\
    sed 's/\$\._\$/"o"/g' |\
    sed 's/\$\._/"u"/g'
```

Le résultat est sous la forme :

```
Function(Function("return "+"\""+JS_BLOB2+"\"")())();
```

6.6 Extraction du script JS

Le script "final" se trouvera donc dans `JS_BLOB2`. Un `print(JS_BLOB2)` dans la console de debug de Firefox permet de l'avoir sous une forme presque lisible :

Après indentation du script JS, on peut s'aider du debugger intégré dans Firefox pour tracer le script pas à pas. L'analyse est aussi faisable directement en statique, compte tenu du peu de ligne présentes dans le script final. En appliquant la même technique que pour le constructeur (avec `sed`), le script se reconstruit en grande partie. Après une dernière étape pour nettoyer le résultat, on obtient le script totalement désobfusqué suivant :

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

function ascii2hex(indata) {
    out = [];
    for (i = 0; i < indata.length; ++i)
        out.push(indata.charCodeAt(i));
    return new Uint8Array(out);
}
function hexDecode(indata) {
    out = [];
    for (i = 0; i < indata.length / 2; ++i)
        out.push(parseInt(indata.substr(i * 2, 2), 16));
    return new Uint8Array(out);
}
function hexEncode(indata) {
    out = '';
    for (i = 0; i < indata.byteLength; ++i) {
        curbyte = indata[i].toString(16);
        if (curbyte.length < 2) curbyte += "0";
        out += curbyte;
    }
    return out;
}
function decryptStage5() {
    iv  = ascii2hex(window.navigator.userAgent.substr(window.navigator.userAgent.indexOf('(') +
    1, 16));
    key = ascii2hex(window.navigator.userAgent.substr(window.navigator.userAgent.indexOf(')') -
    16, 16));
    algo = {
      name: 'AES-CBC',
      iv: iv,
      length: key.length * 8,
    };
    window.crypto.subtle.importKey("raw", key, algo, false, [ "decrypt" ]).then(function(result)
     {
        window.crypto.subtle.decrypt(algo, result, hexDecode(data)).then(function(result) {
            decrypted = new Uint8Array(result);
            window.crypto.subtle.decrypt.digest({ name: 'SHA-1' }, decrypted).then(function(
    result) {
                if (hash == hexEncode(new Uint8Array(result))) {
                    blobType = {
                      type: 'application/octet-stream';
                    };
                    blob = new Blob([decrypted], blobType);
                    url = URL.createObjectURL(blob);
                    document.getElementById("status").innerHTML = '<a href="' + url + '"
    download="stage5.zip">download stage5</a>';
                } else {
                    document.getElementById("status").innerHTML = "<b>Failed to load stage5</b>"
    ;
                }
            });
        }).catch(function() {
            document.getElementById("status").innerHTML = "<b>Failed to load stage5</b>";
        });
    }).catch(function() {
        document.getElementById("status").innerHTML = "<b>Failed to load stage5</b>";
    });
}
window.setTimeout(decryptStage5, 1000);
```

6.7 Calcul de la clef

Le script JS maintenant lisible, on sait que les données présentes dans la variable "data" sont chiffrées en AES128-CBC, avec un vecteur d'initialisation et une clef dépendants du User-Agent du navigateur, et que le condensat SHA-1 du résultat déchiffré est présent dans la variable "hash". Les lignes correspondantes au calcul du vecteur d'initialisation et de la clefs sont les suivantes :

```
iv  = ascii2hex(window.navigator.userAgent.substr(window.navigator.userAgent.indexOf('(') + 1,
    16));
key = ascii2hex(window.navigator.userAgent.substr(window.navigator.userAgent.indexOf(')') - 16,
    16));
```

La partie utilisée du User-Agent s'avère être le "commentaire optionnel". Cette partie du User-Agent n'a aucune forme prédéfinie, si ce n'est les caractères autorisés. Malgré cette information, l'espace de recherche reste bien trop grand pour une attaque par bruteforce. Cependant, il est important de noter la présence de la ligne suivante :

```
document.write('<div style="display:none"><a target="blank" href="chrome://browser/content/
    preferences/preferences.xul">Back to preferences</a></div>');
```

Ce lien ne peut fonctionner que sur un navigateur Firefox.

6.7.1 Generation de User-Agent pouvant provenir de Firefox

Le lien présent dans le script mais qui ne s'affiche pas de manière visible sur la page ne peut fonctionner que sur un navigateur Firefox. Si l'on se restreint a des versions relativement récentes de Firefox, le User-Agent semble prédictible. En effet, il est toujours sous la forme :

```
${OS}; ${Arch}; rv:${Version}
```

Il y a trois type de valeurs possible pour OS : "X11", "Windows NT X.Y" et "Macintosh".

Cas Windows NT Les versions majeurs 5 et 6, ainsi que les revisions mineures de 0 a 3 ont été retenues. Les architectures suivantes on été retenues : "", "WOW64" et "Win64" Le script bash correspondant est le suivant :

```
for ((major = 5; major < 7; major++)); do
    for ((minor = 0; minor < 4; minor++)); do
        param="Windows NT "${major}"."${minor}
        test_uagent "$param"
        for arch in WOW64 Win64; do
            param="Windows NT "${major}"."${minor}"; "${arch}
            test_uagent "$param"
        done
    done
done
```

Cas X11 Le nom du système d'exploitation sera en fait dans la partie Arch, les systèmes suivants ont été retenus : FreeBSD, OpenBSD, Linux, Ubuntu et Debian. Les architectures suivantes ont été retenues : i686, i586, x86_64, armv7l, ppc, amd64 et AMD64. Le script bash correspondant est le suivant :

```
for nixos in FreeBSD OpenBSD Linux Ubuntu Debian; do
    for arch in i686 i586 x86_64 armv7l ppc amd64 AMD64; do
        param="X11; "${nixos}" "$arch
        test_uagent "$param"
    done
done
```

Cas Macintosh Comme précédemment, le nom du système d'exploitation est dans la partie Arch : Mac OS X. Les versions 10.0 à 10.10 ont été retenus. La seule architecture retenue est : Intel. Le script bash correspondant est le suivant :

```
for ((minor = 0; minor < 11; minor++)); do
    param="Macintosh; Intel Mac OS X 10_"${minor}
    test_uagent "$param"
    param="Macintosh; Intel Mac OS X 10."${minor}
    test_uagent "$param"
done
```

Les versions de Firefox considérées vont de 20 a 37, et les révisions vont de 0 a 3.

6.7.2 Récupération du vecteur d'initialisation et de la clef

Pour commencer, il faut extraire les données à déchiffrer dans un fichier temporaire. Le script bash suivant effectue cette opération, ainsi que la création d'un fichier temporaire de sortie pour tester le résultat :

```
TMPF=$(mktemp)
grep data stage4.html | cut -f 2 -d'"' | xxd -r -p > ${TMPF}
OUTF=$(mktemp)
```

Le déchiffrement peut s'effectuer par OpenSSL. L'agorithme sélectionné étant "aes-128-cbc", il faut ensuite passer en paramètre le vecteur d'initialisation et la clef. Une première vérification du résultat peut se faire sur le code d'erreur retourné par OpenSSL. En effet, après déchiffrement OpenSSL vérifie la validité du padding afin de le retirer. Cela laisse cependant encore trop de faux positifs. On sait que la sortie est un fichier Zip, on peut donc vérifier que le fichier obtenu contient bien une signature Zip. Au final, le script de bruteforce ressemble a ça :

```
function test_uagent()
{
    for ((fxmajor = 20; fxmajor < 38; fxmajor++)); do
        for ((fxminor = 0; fxminor < 4; fxminor++)); do
            uagent=$1"; rv:"${fxmajor}"."${fxminor}
            pos=$(expr length "$uagent" - 16)
            IV=$(echo -n "${uagent:0:16}" | xxd -p)
            KEY=$(echo -n "${uagent:$pos:16}" | xxd -p)
            openssl enc -d -aes-128-cbc -iv $IV -K $KEY -in $TMPF -out $OUTF 2>/dev/null && \
                file ${OUTF} | grep -q Zip && \
                echo -e "Found possible User-Agent: "${uagent}"\n  IV:  "$IV"\n  KEY: "$KEY
        done
    done
}
```

Le choix du langage bash pour effectuer le brute force est discutable. En effet, l'opération de génération des paramètres de la commande OpenSSL correspond a 30% du temps total. Cependant, compte tenu du faible nombre d'itération, le temps total nécessaire reste acceptable. Après avoir fait tourner le script, on obtient finalement la sortie suivante :

```
$ ./bfuseragent.sh
Found possible User-Agent: Macintosh; Intel Mac OS X 10.6; rv:35.0
  IV:  4d6163696e746f73683b20496e74656c
  KEY: 20582031302e363b2072763a33352e30
```

6.8 Déchiffrement des données

Il est possible de modifier le User-Agent de Firefox en créant une clef "general.useragent.override" dans `about:config`. En lui attribuant la valeur "(Macintosh; Intel Mac OS X 10.6; rv :35.0)", le javascript devrait permettre de déchiffrer et télécharger le fichier stage5.zip. On peut aussi directement obtenir le résultat, en une ligne de bash :

```
$ grep data stage4.html | cut -f 2 -d'"' |  xxd -r -p | openssl enc -d -aes-128-cbc -iv 4
    d6163696e746f73683b20496e74656c -K 20582031302e363b2072763a33352e30 -out stage5.zip
```

On vérifie ensuite le résultat du hachage basé sur SHA-1 :

```
$ grep hash stage4.html
            var hash = "08c3be636f7dffd91971f65be4cec3c6d162cb1c";
$ sha1sum stage5.zip
08c3be636f7dffd91971f65be4cec3c6d162cb1c  stage5.zip
```

7 I love ST20 architecture

7.1 Outils

- `hexdump`
- `awk`
- `gcc`
- `ST20 Embedded Toolset R2.3.1`

7.2 Documents de référence

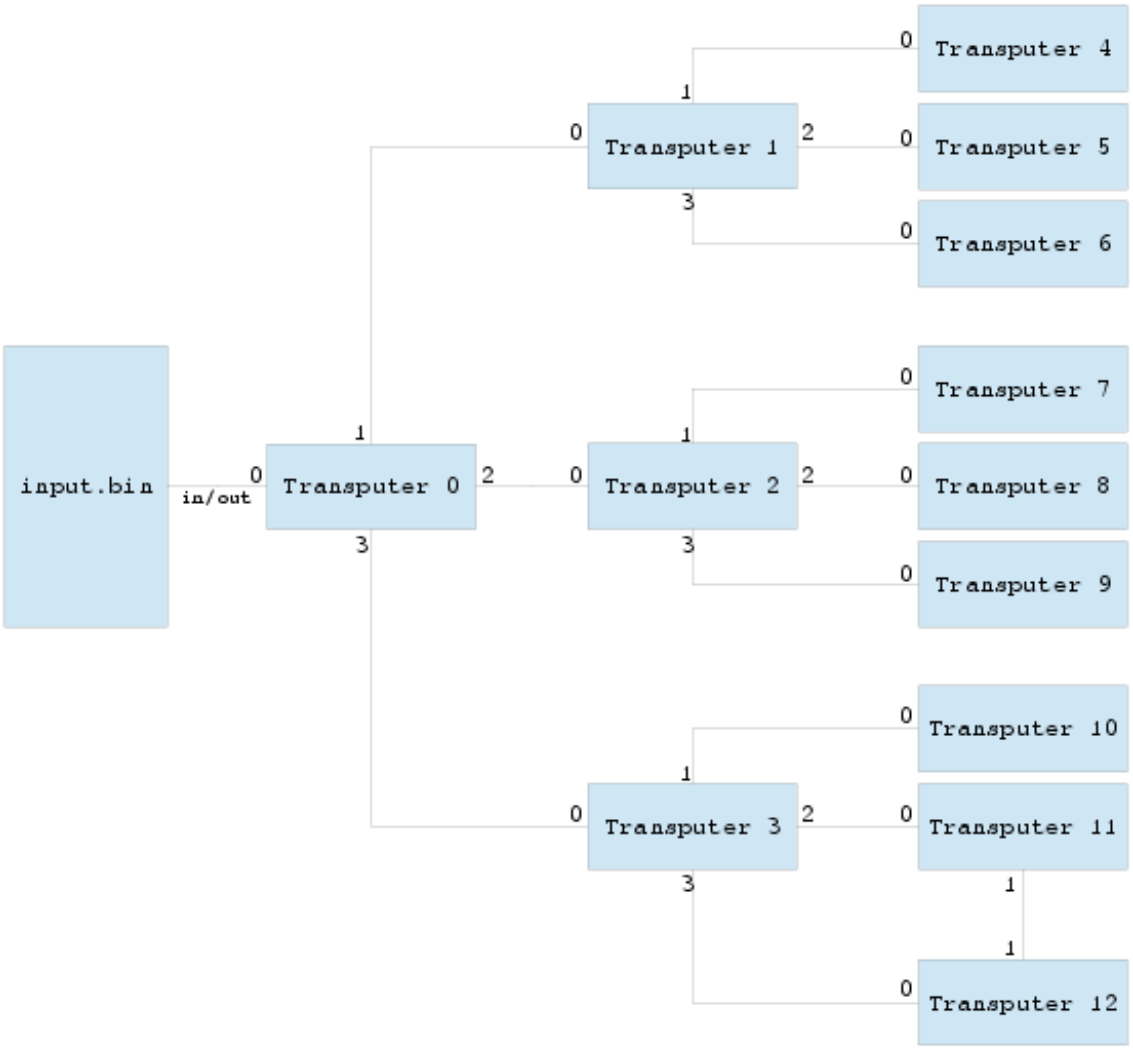
- `ST20C2/C4 Core Instruction Set Reference Manual`
- `ST20 Embedded Toolset R2.3 Reference Manual`

7.3 Contenu de l'archive récupérée au stage précédent

- `schematic.pdf`
- `input.bin`

7.4 Analyse du contenu

Le schéma est le suivant :



Il contient en outre un vecteur de test ainsi que les classiques SHA-256 des fichiers d'entrée et de sortie :

```
SHA256:
    a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81 - encrypted
    9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321 - decrypted
Test vector:
    key = "*SSTIC-2015*"
    data = "1d87c4c4e0ee40383c59447f23798d9fefe74fb82480766e".decode("hex")
    decrypt(key, data) == "I love ST20 architecture"
```

7.5 Analyse du binaire

7.5.1 Identification du jeu d'instruction

Le vecteur de test donné dans le PDF fait référence au ST20 de SGS-Thomson. Deux versions du jeu d'instruction existent en fonction de la version du cœur : c1 ou c2/c4.

Le début du binaire input.bin n'a pas de signature connue. On a donc peut-être affaire à une image mémoire brute contenant des instructions ST20. Le tout premier octet est 0xf8. Cela se traduit par l'instruction NOT en ST20c1 et PROD pour les ST20c2/c4. Dans les deux cas, la présence de cette instruction à cet endroit est anormale, puisque cela signifierait que les registres sont déjà initialisés et qu'une inversion bit à bit ou un produit de deux registre puisse avoir un sens à ce moment de l'exécution du code, ce qui est hautement improbable. Le mieux est d'ignorer cet octet pour le moment et de se focaliser sur l'identification du jeu d'instruction ainsi que de la partie du code responsable du démarrage du système.

Une première revue du binaire avec `strings` nous donne les chaînes de caractères suivantes présentes en début de fichier :

```
$ strings -n5 input.bin | head -3
Boot ok
Code Ok
Decrypt
```

Ces chaines de caractères sont très probablement écrites sur la sortie 0 du transputeur T0 lors de l'initialisation. La suite de l'analyse va consister à retrouver les instructions en charge du passage de l'adresse de ces chaînes de caractères dans le bon registre avant le code d'opération en charge des entrées/sorties. En partant du principe que l'adressage se fait relativement au compteur ordinal, on peut s'attendre à ce que l'instruction LDPI soit utilisée. Cette instruction possède un code différent en fonction de la version du cœur : "23 FA" pour les c1 et "21 FB" pour les c2/c4. Il faut commencer par récupérer l'offset de la première chaîne de caractères :

```
$ grep -aob 'Boot ok' input.bin
221:Boot ok
```

Maintenant, toujours à l'aide de `grep`, on peut récupérer les offsets des instructions LDPI potentielles :

```
$ grep -aob -P '\x23\xfa' input.bin  | cut -d':' -f1 | head -3
15410
41326
76878
$ grep -aob -P '\x21\xfb' input.bin  | cut -d':' -f1 | head -3
18
37
47
```

Le binaire contenant un fichier chiffré, il n'est pas étonnant de trouver autant d'occurrences d'un si petit motif. Cependant, compte tenu de la position de la première chaîne de caractères dans le binaire, le jeu d'instruction des c2/c4 est bien plus probable. Pour que l'instruction soit totalement validée, il faudrait qu'elle soit précédée d'un chargement d'une valeur constante égale à 221 - (18 + 2) = 201 = 0xc9. Le chargement de constante, LDC, est effectué par l'instruction primaire 4n, pour les deux jeux d'instructions. Afin de pouvoir charger plus de 4 bits, deux instructions primaires existent : PFIX (2n) et

NFIX (6n) pour préfixe positif et négatif respectivement. Dans notre cas, la valeur à charger est positive, si le jeu d'instruction est bien celui des c2/c4 et que l'offset trouvé de 18 correspond bien au chargement de l'adresse de "Boot ok". Dans ce cas, la suite d'instruction nécessaire au chargement de cette chaîne s'écrirait :

```
2C 49 21 FB
```

Cela peut se vérifier par `hexdump` :

```
$ hexdump -C -s 16 -n 4 input.bin
00000010  2c 49 21 fb                                       |,I!.|
00000014
```

Pour pouvoir valider définitivement, on peut décoder manuellement les instructions suivant le chargement de l'adresse identifiée. L'instruction MINT permet de charger MIN_INT32 (0x80000000) dans le registre A, et l'instruction OUT correspondrait à un appel système `write` sous la forme :

```
write(GET_FD(Breg), Creg, Areg)
```

Pour la lecture du code, il est important de noter que seul le registre A peut être directement assigné. En fonction de l'instruction, les registres vont être décalés, la plupart du temps sous cette forme : A → B → C. Cela nous donne le code désassemblé suivant :

```
2C 49       LDC     0xC9 ; A = 0xc9
21 FB       LDPI         ; A = 0xc9 + (0x12 + 2) = 0xdd = "Boot ok"
24 F2       MINT         ; B = "Boot ok", A = 0x80000000
48          LDC     8    ; C = "Boot ok", B = 0x80000000, A = 8
FB          OUT          ; write(0x80000000, "Boot ok", 8)
```

Il apparait clairement que ce morceau de code écrit "Boot ok" sur sa sortie 0, qui est désignée par 0x80000000. On peut donc en conclure que le jeu d'instruction est celui du ST20c2/c4, et le code responsable du démarrage du système se termine a l'offset 18.

7.5.2 Récupération du fichier chiffré et format de la sortie

Si on continue l'analyse à l'aide de `strings`, on peut trouver d'autres chaînes de caractères intéressantes à regarder. À l'offset 0x985 du fichier, on peut trouver la chaîne de caractères "KEY :" suivie de 12 octets à 0xff, qui représentent très certainement la valeur par défaut de la clef de déchiffrement, celle donnée pour le vecteur de test faisant 12 octets. L'octet suivant vaut 0x17 = 23, qui est la taille de la chaîne de caractères qui suit : "congratulations.tar.bz2", qui est a priori le nom du fichier en sortie.

Cette dernière information est très utile, puisque cela nous permet d'avoir une idée des premiers octets une fois déchiffrés. En effet, pour la version 2 du format bzip, les 10 premiers octets sont fixes :

```
42 5a 68 39 31 41 59 26 53 59
```

Les octets suivants ne semblent pas structurés. Il est probable que ce soit le fichier chiffré contenu dans le binaire. Cela peut se vérifier grâce au résultat du hachage SHA-256 donné dans le PDF :

```
$ dd if=input.bin of=encrypted bs=1 skip=$((0x9ad)) 2>/dev/null
$ sha256sum encrypted
a5790b4427bc13e4f4e9f524c684809ce96cd2f724e29d94dc999ec25e166a81  encrypted
```

À noter la taille du fichier chiffré : 250606 octets. Compte tenu de la taille, on peut s'attendre à ce que ça ne soit pas tout a fait la dernière étape. Une autre information intéressante est que ce nombre n'a que deux facteurs premiers : 2 et 125303. L'algorithme de chiffrement opère alors sur des mots de 16 bits au maximum et non des blocs de la taille de la clef. Le plus vraisemblable est que l'algorithme soit du chiffrement de flux, et qu'il travaille au niveau de l'octet.

Cette dernière information est très utile pour déterminer la manière d'attaquer le problème. On connait les 10 premiers octets du résultat du déchiffrement, et on sait que le déchiffrement d'un octet n'est pas dépendant d'autres octets puisque le chiffrement ne se fait visiblement pas par bloc. La clef faisant 12 octets, il manquerait a priori au moins 2 octets à attaquer par bruteforce pour retrouver la clef.

7.6 ST20 Embedded Toolset R2.3.1

7.6.1 Introduction

Pour la suite, il nous faudrait pouvoir désassembler le binaire. Quatre possibilités sont disponibles pour le désassemblage :

- `IDA`, qui est payant et dont le support ST20, comme pour toute architecture hors x86 et ARM, est particulièrement minimaliste
- `st20dis`, dont les sources ne sont pas disponibles mais dont le support ST20 est plutôt bon
- `st20run` du `ST20 Embedded Toolset`, dont les sources ne sont pas disponibles mais dont le support ST20 est correct
- Développer un désassembleur qui supporte exactement ce dont on a besoin pour mener la rétro-ingénierie

Le choix de `st20dis` pourrait paraître évident, cependant, l'absence du code source rend l'intérêt de cet outil marginal. En effet, il ne permet pas l'émulation d'un cœur ST20c2 et ses capacités d'analyse symbolique du code sont bien trop limitées pour une rétro-ingénierie efficace. Le choix retenu est finalement `st20run` pour la première partie de l'analyse. En effet, le fait de pouvoir analyser le binaire dynamiquement va nous permettre de déterminer les bases pour l'élaboration d'une décompilation dont le but à terme est de pouvoir générer du code émulant le fonctionnement de l'architecture proposée.

7.6.2 Installation

Un paquetage RPM peut se trouver à l'adresse : `ftp://ftp.stlinux.com/pub/tools/products/st20tools/R2.3/R2.3.1/linux/stm-st20.231-2.3.1-1.i386.rpm`. La commande suivante permet d'extraire le contenu :

```
$ rpm2cpio stm-st20.231-2.3.1-1.i386.rpm | cpio -idmv
```

Un certain nombre de variables d'environnements sont nécessaires pour pouvoir utiliser la suite de développement :

```
$ export ST20ROOT=‘pwd‘/opt/STM/ST20R2.3.1/
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ST20ROOT/lib
$ export PATH=$PATH:$ST20ROOT/bin
```

Des fichiers d'exemples sont présents dans le répertoire `$ST20ROOT/examples/getstart/` dont notamment :

- `hello.c`
- `sti5500.cfg`

Le STi5500 est à base de cœur ST20c2, ce qui correspond au jeu d'instruction utilisé dans notre binaire. Un simulateur ainsi qu'un debugger sont présents dans la suite de développement. Cela va s'avérer utile pour mener la première étape de la rétro-ingénierie du binaire `input.bin`. La compilation d'un programme peut se faire avec la commande suivante :

```
$ st20cc hello.c -T sti5500.cfg -p link -g
```

Pour debugger le programme en mode graphique, il faut ensuite lancer la commande suivante :

```
$ st20run -i sti5500.cfg -t eval5500sim -g hello.lku
```

7.6.3 Analyse dynamique du code

La première étape consiste à générer un binaire dans un format lisible par le simulateur ST20. Pour ça, il faut créer un programme ne contenant que la fonction `main` et dont le contenu de la fonction est une série de directive “byte” pour l'assembleur qui reprennent les octets du binaire `input.bin`. Le fichier peut se générer avec `hexdump`, puis être compilé avec `st20cc` et finalement debuggé avec `st20run` :

```
$ (echo -e "int main() {\n  __asm {\n" ; hexdump -vn $((0xf9)) -e '"byte " 1/1 "0x%02x"";\n"'
    input.bin ; echo -e "\n  }\n}") > input.c
$ st20cc input.c -T sti5500.cfg -p link -g
$ st20run -i sti5500.cfg -t eval5500sim -g input.lku
```

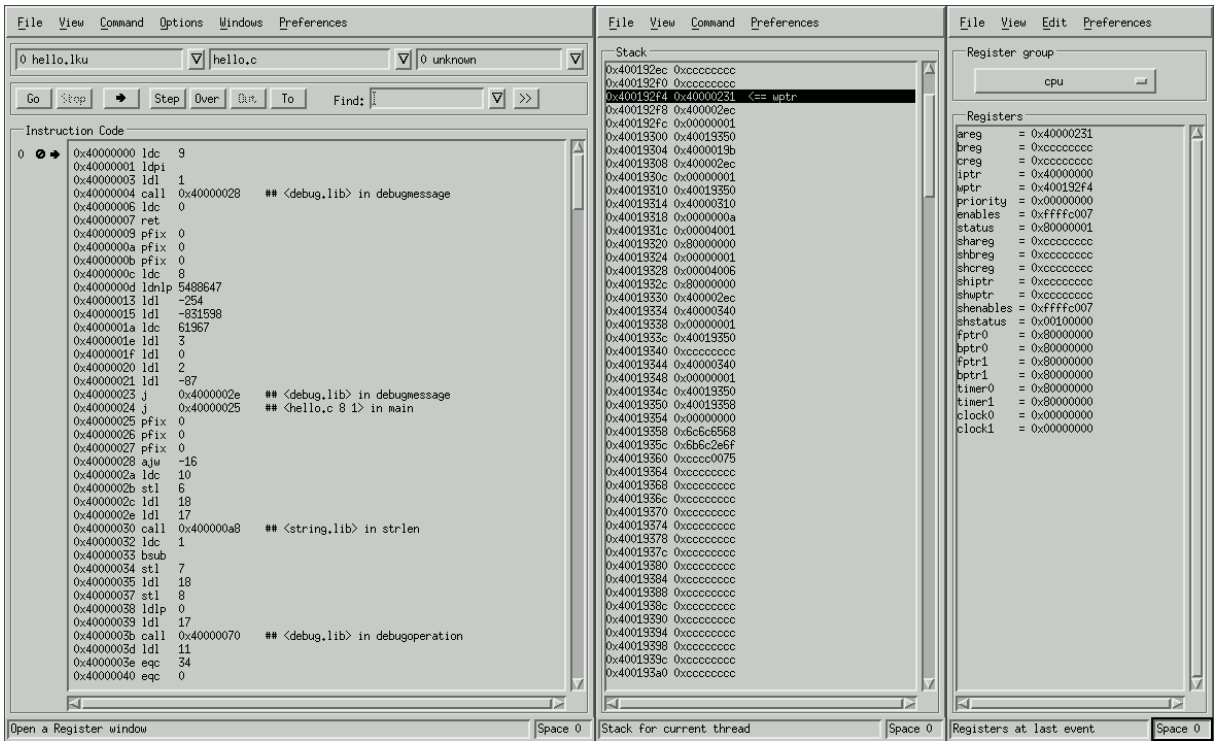


Fig. 1: Une interface futuriste pour une architecture futuriste.

La taille du code (0xf9) a pu être identifiée grâce à l'instruction `ret` qui se termine à cet endroit. Le premier octet qui posait problème lors de l'analyse est 0xf8, soit la taille effective du code. On peut donc en déduire le format du binaire :

Code size [1B]	Code [Code Size]	Data read by IN ...

7.6.4 Désassemblage

Cette méthode est donnée a titre purement informatif. C'est la méthode qui a été utilisée lors de la résolution du challenge. Le programme `st20run` implémente 3 classes C++ pour gérer les différents type de cœur ST20. Pour le désassemblage, la méthode qui nous intéresse est la suivante :

```
uchar * CST20C2Machine::Disassemble(int Iptr, int *NextIptr, uchar *ins, int insLen, bool
    hexDisplay);
```

Cette méthode est appelée lors de la sélection View → Instructions de l'interface graphique. Le résultat global est stocké dans de la mémoire sur le tas. Un fois la vue "instructions" sélectionnée, il est possible de récupérer le code désassemblé dans le tas du processus `st20run` pour pallier l'absence de possibilité de copier/coller dans l'interface graphique :

```
PID=$(pidof st20run); cat /proc/$PID/maps | grep  heap  | cut -d' ' -f1 | sed 's/-/ /' | while
    read start stop ; do dd if=/proc/$PID/mem bs=4k skip=$((0x$start/0x1000)) count=$(((0x$stop
    -0x$start)/0x1000)) 2>/dev/null; done | strings -n 10 | egrep '^0x4000.*$' | awk 'BEGIN{a=0;
     b="";}{if ($0 ~ /prod/) { a=1; b=""} else if (a==1 && $0 ~ /0x400000f9/) { a=0; print b;}
    if (a == 1) {b=b $0 "\n";} }' > input.S
```

Muni du résultat du désassemblage et avec le code tournant dans un émulateur avec un debugger attaché, il est maintenant possible d'annoter l'assembleur avec du pseudo code C qui correspondrait.

Avant de partir sur l'analyse dynamique, on peut noter certaines choses sur le préambule de la fonction :

```
0x40000001 ajw   -76
0x40000003 ldc   0
0x40000004 stl   1
0x40000005 ldc   0
0x40000006 stl   3

0x40000007 mint
0x40000009 ldnlp 1024
0x4000000c gajw
0x4000000e ajw   -76
```

La première partie du code assembleur correspond à un préambule standard de fonction sur ST20. La fonction commence par réserver 304 octets pour des variables locales. De ce que l'on peut voir, deux de ces variables locales sont initialisées a zéro : var_4 et var_C. Le deuxième bloc assembleur provient très probablement d'une directive `__asm{}`, et permet d'assigner une valeur valide au pointeur de pile : 0x80001000. Le développeur a ensuite pris le soin de remettre l'opération permettant l'ajustement du pointeur de pile, mais en oubliant d'initialiser les deux variables initialisées précédemment. Cela n'a pas d'effet sur la suite puisque ces variables seront assignées avant d'être utilisées. On peut donc avoir une idée de la façon le développeur s'y est pris pour génerer le binaire, ainsi que la manière dont l'architecture globale est censée fonctionner.

7.7 Analyse dynamique

L'analyse dynamique du code permet de se familiariser avec ce jeu d'instruction finalement peu répandu. Cela permet aussi de résoudre les adresses calculées ainsi que la détermination du rôle des variables locales. La démarche consiste à tracer le programme, instruction par instruction ou par bloc, pour clarifier les opérations effectuées par le binaire.

Il existe un bug connu sur la gestion de GAJW par `st20run`. Pour ce genre de cas, il est possible de remplacer les opérations qui posent problème par un NOP (63 F0).

Le jeu d'instruction s'avère relativement simple. La plupart des instructions servent uniquement a faire des chargements dans des registres. Ces chargements peuvent être une constante (LDC), une variable locale (LDL), un pointeur local (LDLP) ou un pointeur global (LDNLP). L'avantage du debugger est que cela permet de rapidement suivre les rotations des registres entre chaque instruction et d'en déduire l'opération finale effectuée ainsi que l'endroit où le résultat sera stocké.

Après analyse, on peut reconstruire le programme (valide) complet. Pour que le code soit valide, certaines définitions sont à rajouter au début :

```
static const char msg0[] = "Boot ok";
static const char msg1[] = "Code Ok";
static const char msg2[] = "Decrypt";
#define read(chan, buf, size)  __asm { \
                                    ldabc (size), (chan), (buf); \
                                    in; \
                               }
#define write(chan, buf, size)  __asm { \
                                    ldabc (size), (chan), (buf); \
                                    out; \
                               }
```

Le corps de la fonction `main` commence avec la réservation de 76 mots de 32 bits sur la pile, sous la forme suivante :

```
int main()
{
    struct {                                                /*  0x40000001 ajw   -76        */
        uint32_t size;
        uint32_t channel;
        uint32_t unused;
    } ibuf;
    uint32_t fname[64];
    uint32_t unused;
    uint8_t  key[12];
    uint32_t key_idx;
    uint32_t fname_sz = 0;                                  /*  0x40000005 ldc   0
                                                                0x40000006 stl   3          */
    uint32_t keyname;
    uint32_t curbyte = 0;                                   /*  0x40000003 ldc   0
                                                                0x40000004 stl   1          */
    uint8_t  byteT3;
    uint8_t  byteT2;
    uint8_t  byteT1;
    uint8_t  byteT0;
```

On retrouve ensuite un rajout manuel d'assembleur afin de donner une valeur valide au pointeur de pile :

```
    /* Stack pointer = 0x80001000 */
    __asm {
        mint;                                               /*  0x40000007 mint             */
        ldnlp 0x1000;                                       /*  0x40000009 ldnlp 1024       */
        gajw;                                               /*  0x4000000c gajw             */
        ajw -76;                                            /*  0x4000000e ajw   -76        */
    }
```

Suivi du message comme quoi le démarrage s'est bien passé :

```
    /* Boot ok */
    write(0x80000000, msg0, 8);                             /*  0x40000010 ldc   201
                                                                0x40000012 ldpi
                                                                0x40000014 mint
                                                                0x40000016 ldc   8
                                                                0x40000017 out              */
```

Une première boucle est présente dans le code, elle sert à charger le code des transputeurs 1 à 3 :

```
    /* Load program for T1, T2 and T3 */
    while (1) {
        read(0x80000010, &ibuf, 12);                    /*  0x40000018 ldlp  73
                                                            0x4000001a mint
                                                            0x4000001c ldnlp 4
                                                            0x4000001d ldc   12
                                                            0x4000001e in              */

        if (ibuf.size == 0) break;                      /*  0x4000001f ldl   73
                                                            0x40000021 cj    0x40000038 */

        read(0x80000010, 0x400000F4, ibuf.size);        /*  0x40000023 ldc   205
                                                            0x40000025 ldpi
                                                            0x40000027 mint
                                                            0x40000029 ldnlp 4
                                                            0x4000002a ldl   73
                                                            0x4000002c in              */

        write(ibuf.channel, 0x400000F4, ibuf.size);     /*  0x4000002d ldc   195
                                                            0x4000002f ldpi
                                                            0x40000031 ldl   74
                                                            0x40000033 ldl   73
                                                            0x40000035 out             */

    }                                                   /*  0x40000036 j     0x40000018 */

    write(0x80000004, &ibuf, 12);                       /*  0x40000038 ldlp  73
                                                            0x4000003a mint
                                                            0x4000003c ldnlp 1
                                                            0x4000003d ldc   12
                                                            0x4000003e out             */

    write(0x80000008, &ibuf, 12);                       /*  0x4000003f ldlp  73
                                                            0x40000041 mint
                                                            0x40000043 ldnlp 2
                                                            0x40000044 ldc   12
                                                            0x40000045 out             */

    write(0x8000000C, &ibuf, 12);                       /*  0x40000046 ldlp  73
                                                            0x40000048 mint
                                                            0x4000004a ldnlp 3
                                                            0x4000004b ldc   12
                                                            0x4000004c out             */
```

Après le chargement du code, on retrouve l'écriture de "Code Ok" sur le port 0, suivi de la lecture des champs déjà identifiés dans le binaire concernant la clef et le nom du fichier de sortie :

```
    /* Code Ok */
    write(0x80000000, msg1, 8);                    /*  0x4000004d ldc   148
                                                       0x4000004f ldpi
                                                       0x40000051 mint
                                                       0x40000053 ldc   8
                                                       0x40000054 out              */

    read(0x80000010, keyname, 4);                  /*  0x40000055 ldlp  2
                                                       0x40000056 mint
                                                       0x40000058 ldnlp 4
                                                       0x40000059 ldc   4
                                                       0x4000005a in               */

    /* Read key */
    read(0x80000010, key, 12);                     /*  0x4000005b ldlp  5
                                                       0x4000005c mint
                                                       0x4000005e ldnlp 4
                                                       0x4000005f ldc   12
                                                       0x40000060 in               */

    /* Decrypt */
    write(0x80000000, msg2, 8);                    /*  0x40000061 ldc   136
                                                       0x40000063 ldpi
                                                       0x40000065 mint
                                                       0x40000067 ldc   8
                                                       0x40000068 out              */

    read(0x80000010, &fname_sz, 1);                /*  0x40000069 ldlp  3
                                                       0x4000006a mint
                                                       0x4000006c ldnlp 4
                                                       0x4000006d ldc   1
                                                       0x4000006e in               */

    read(0x80000010, fname, fname_sz);             /*  0x4000006f ldlp  9
                                                       0x40000070 mint
                                                       0x40000072 ldnlp 4
                                                       0x40000073 ldlp  3
                                                       0x40000074 lb
                                                       0x40000075 in               */
```

On arrive à la boucle de déchiffrement. Pour commencer, le transputeur 0 récupère un octet à déchiffrer, puis envoie la clef courante aux transputeurs 1 à 3 :

```
    key_idx = 0;                                        /*  0x40000076 ldc   0
                                                            0x40000077 stl   4          */

    /* Decryption loop */
    while (1) {
        read(0x80000010, &curbyte, 1);                  /*  0x40000078 ldlp  1
                                                            0x40000079 mint
                                                            0x4000007b ldnlp 4
                                                            0x4000007c ldc   1
                                                            0x4000007d in               */

        write(0x80000004, key, 12);                     /*  0x4000007e ldlp  5
                                                            0x4000007f mint
                                                            0x40000081 ldnlp 1
                                                            0x40000082 ldc   12
                                                            0x40000083 out              */

        write(0x80000008, key, 12);                     /*  0x40000084 ldlp  5
                                                            0x40000085 mint
                                                            0x40000087 ldnlp 2
                                                            0x40000088 ldc   12
                                                            0x40000089 out              */

        write(0x8000000C, key, 12);                     /*  0x4000008a ldlp  5
                                                            0x4000008b mint
                                                            0x4000008d ldnlp 3
                                                            0x4000008e ldc   12
                                                            0x4000008f out              */
```

Le transputeur 0 récupère ensuite le résultat sur un octet du traitement de la clef par les transputeurs esclaves. Une opération de ou-exclusif est effectuée afin de n'obtenir qu'un seul octet au final :

```
        read(0x80000014, &byteT1, 1);                    /*  0x40000090 ldlp  0
                                                             0x40000091 adc   1
                                                             0x40000092 mint
                                                             0x40000094 ldnlp 5
                                                             0x40000095 ldc   1
                                                             0x40000096 in              */

        read(0x80000018, &byteT2, 1);                    /*  0x40000097 ldlp  0
                                                             0x40000098 adc   2
                                                             0x40000099 mint
                                                             0x4000009b ldnlp 6
                                                             0x4000009c ldc   1
                                                             0x4000009d in              */

        read(0x8000001C, &byteT3, 1);                    /*  0x4000009e ldlp  0
                                                             0x4000009f adc   3
                                                             0x400000a0 mint
                                                             0x400000a2 ldnlp 7
                                                             0x400000a3 ldc   1
                                                             0x400000a4 in              */

        byteT1 ^= byteT2;                                /*  0x400000a5 ldlp  0
                                                             0x400000a6 adc   1
                                                             0x400000a7 lb
                                                             0x400000a8 ldlp  0
                                                             0x400000a9 adc   2
                                                             0x400000aa lb
                                                             0x400000ab xor             */
        byteT1 ^= byteT3;                                /*  0x400000ad ldlp  0
                                                             0x400000ae adc   3
                                                             0x400000af lb
                                                             0x400000b0 xor
                                                             0x400000b2 ldlp  0
                                                             0x400000b3 adc   1
                                                             0x400000b4 sb              */
```

Cette partie là du code constitue un étape très importante dans l'analyse de l'algorithme de chiffrement. On peut voir que la clef est utilisée quasiment telle quelle pour donner le premier octet déchiffré. L'octet de la clef ayant servi à cette opération est ensuite remplacé par le résultat précédemment obtenu des transputeurs esclaves :

```
        byteT0 = curbyte ^ ((key[key_idx] << 1) + key_idx); /*  0x400000b6 ldlp  1
                                                             0x400000b7 lb
                                                             0x400000b8 ldl   4
                                                             0x400000b9 ldlp  5
                                                             0x400000ba bsub
                                                             0x400000bb lb
                                                             0x400000bc ldl   4
                                                             0x400000bd ssub
                                                             0x400000bf xor
                                                             0x400000c1 ldlp  0
                                                             0x400000c2 sb              */

        key[key_idx] = byteT1;                           /*  0x400000c4 ldlp  0
                                                             0x400000c5 adc   1
                                                             0x400000c6 lb
                                                             0x400000c7 ldl   4
                                                             0x400000c8 ldlp  5
                                                             0x400000c9 bsub
                                                             0x400000ca sb              */
```

Finalement, l'octet déchiffré est écrit sur le port 0, et l'index de la clef est incrémenté :

```
        key_idx++;                                          /*  0x400000cc ldl   4
                                                                0x400000cd adc   1
                                                                0x400000ce dup
                                                                0x400000d0 stl   4          */

        if (key_idx == 12) {                                /*  0x400000d1 eqc   12
                                                                0x400000d2 cj    0x400000d6 */

            key_idx = 0;                                    /*  0x400000d3 adc   0
                                                                0x400000d4 ldc   0
                                                                0x400000d5 stl   4          */
        }

        write(0x80000000, &byteT0, 1);                      /*  0x400000d6 ldlp  0
                                                                0x400000d7 mint
                                                                0x400000d9 ldc   1
                                                                0x400000da out              */

    }                                                       /*  0x400000db j     0x40000078 */

}                                                           /*  0x400000f5 ajw   76
                                                                0x400000f7 ret              */
```

Le résultat de la compilation de ce programme ne donne pas exactement la même chose que le binaire d'origine. Soit le compilateur est différent, soit le code obtenu par la rétro-ingénierie est sensiblement différent de l'original.

Cette décompilation réussie, on connait maintenant mieux la manière dont est mis en place le programme dans l'architecture. À l'état initial, les transputeurs possèdent un premier programme (booloader) qui effectue les opérations suivantes :

- Lecture d'un octet sur le port 0 dans une variable locale N ;
- lecture de N octets sur le port 0 et écriture en mémoire ;
- saut à l'adresse de copie ;
- exécution du programme chargé.

La suite des donnés sera chargée par le programme ainsi obtenu. À ce stade, nous avons tous les éléments pour élaborer un émulateur capable d'interpréter le binaire `input.bin` fourni.

Un autre résultat encore plus intéressant concerne l'utilisation de la clef. Nous connaissons déjà les 10 premiers octets du résultat déchiffré grâce a la signature bzip2, et nous savons maintenant que 7 bits de la clefs sont utilisés pour chacun de ces octets. Cela signifie donc que nous avons 70 bits sur les 96 de la clef complète :

```
Fichier chiffré         FE F3 50 DC 81 BC 97 27 89 AC ...
Fichier en clair        42 5A 68 39 31 41 59 26 53 59 ...
                   XOR ---------------------------------
                        BC A9 38 E5 B0 FD CE 01 DA F5 ...
Index de clef           00 01 02 03 04 05 06 07 08 09 ...
                    -   ---------------------------------
                        BC A8 36 E2 AC F8 C8 FA D2 EC ...
Division par 2          02 02 02 02 02 02 02 02 02 02
                    /   ---------------------------------
Clef                    5E 54 1B 71 56 7C 64 7D 69 76 ...
```

Ce qui donne une clef en binaire, avec 'X' pour les bits inconnus :

```
Clef = X101 1110 X101 0100 X001 1011 X111 0001 X101 0110 X111 1100
       X110 0100 X111 1101 X110 1001 X111 0110 XXXX XXXX XXXX XXXX
```

Les 26 bits restant représentent un espace de recherche d'environ 64 millions de possibilités. Si l'on souhaite respecter la règle "inférieur à une minute" (qui provient d'un autre challenge, mais qui a l'avantage

de permettre de savoir si une solution est suffisamment bonne), cela implique de pouvoir déchiffrer et valider la sortie à un rythme supérieur à 1 million par seconde.

Un possibilité d'oracle pour vérifier le bon format de la sortie peut se baser sur les octets correspondants au "huffman used bitmaps". Compte tenu de la taille assez large du fichier de sortie, il y a de grandes chances que les premiers octets de ce bitmap soient égaux à 0xff car la compression va générer beaucoup de symboles différents. Pour notre cas, nous pouvons donc tester que les octets déchiffrés 18 à 31 soient bien égaux a 0xff.

En conclusion, pour que le bruteforce soit suffisamment rapide sur une machine "standard" (2GHz), il faudrait donc compter un maximum de 100 cycles/octets pour le déchiffrement. Pour viser le classement rapidité du challenge, cette observation était importante car elle permet de prendre la décision qui fait parvenir le plus rapidement au stage suivant : ne pas émuler l'architecture complète mais se concentrer sur la rétro-ingénierie pour en faire une implémentation performante.

7.8 Extraction des programmes des transputeurs

Afin d'extraire le code, on peut commencer par isoler celui du transputeur 0 pour ensuite extraire les codes des transputeurs 1 à 3. Cela peut se faire avec du code proche de ce qui a été obtenu par rétro-ingénierie précédemment :

```
FILE *fp, *fs;
int i;
struct {
    uint32_t size;
    uint32_t dest;
    uint32_t null;
} ibuf;
uint8_t buf[13][1024];
uint8_t code_sz[13];
int buf_idx[13] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

fp = fopen(av[1], "r");

// Read transputeur 0 code size
fread(&code_sz[0], 1, 1, fp);

// Read transputeur 0 code
fread(buf[0], code_sz[0], 1, fp);

// Parse transputeur 1 to 3 payload
while (fread(&ibuf, 12, 1, fp) > 0) {
    if (ibuf.size == 0) {
        break;
    }
    switch (ibuf.dest) {
        case 0x80000004:
            fread(&buf[1][buf_idx[1]], ibuf.size, 1, fp);
            buf_idx[1] += ibuf.size;
            break;
        case 0x80000008:
            fread(&buf[2][buf_idx[2]], ibuf.size, 1, fp);
            buf_idx[2] += ibuf.size;
            break;
        case 0x8000000C:
            fread(&buf[3][buf_idx[3]], ibuf.size, 1, fp);
            buf_idx[3] += ibuf.size;
            break;
    }
}
```

Avant de passer à la suite, il est intéressant de voir ce que fait le début du code des transputeurs 1 à 3. En réutilisant la méthode précédente qui consiste à :

– Générer un fichier C valide avec des directives assembleur `byte` contenant les octets du programmes

– Compiler le fichier C avec `st20cc`
– Simuler et debugger le binaire obtenu avec `st20run`

Les 3 transputeurs possèdent le même code de démarrage. Le désassemblage de leur code nous permet de retrouver la même boucle de chargement du code pour les transputeurs 4 à 12.

```
0x40000000 ajw   -8
0x40000002 mint                  ; set stack pointer
0x40000004 ldnlp 1024
0x40000007 gajw

0x40000009 ajw   -8              ; 8x 32bits word reserved

0x4000000b ldlp  5               ; struct {uint32_t size; uint32_t dest; uint32_t unused;} ibuf;
0x4000000c mint
0x4000000e ldnlp 4
0x4000000f ldc   12
0x40000010 in                    ; read(PORT_0, &ibuf, 12);
0x40000011 ldl   5A              ; if (ibuf.size == 0)
0x40000012 cj    0x40000026      ; break;
0x40000014 ldc   84
0x40000016 ldpi
0x40000018 mint
0x4000001a ldnlp 4
0x4000001b ldl   5
0x4000001c in                    ; read(PORT_0, buffer, ibuf.size);
0x4000001d ldc   75
0x4000001f ldpi
0x40000021 ldl   6
0x40000022 ldl   5
0x40000023 outA                  ; write(ibuf.dest, buffer, ibuf.size);
0x40000024 j     0x4000000b      ; } while (1);
```

Le découpage peut donc continuer avec une nouvelle boucle :

```
for (i = 1; i < 4; i++) {
    int idx;
    code_sz[i] = buf[i][0];
    idx = code_sz[i] + 1;
    while (idx < buf_idx[i]) {
        memcpy(&ibuf, &buf[i][idx], 12);
        idx += 12;
        switch (ibuf.dest) {
            case 0x80000004:
                memcpy(&buf[3 * i + 1][buf_idx[3 * i + 1]], &buf[i][idx], ibuf.size);
                buf_idx[3 * i + 1] += ibuf.size;
                break;
            case 0x80000008:
                memcpy(&buf[3 * i + 2][buf_idx[3 * i + 2]], &buf[i][idx], ibuf.size);
                buf_idx[3 * i + 2] += ibuf.size;
                break;
            case 0x8000000C:
                memcpy(&buf[3 * i + 3][buf_idx[3 * i + 3]], &buf[i][idx], ibuf.size);
                buf_idx[3 * i + 3] += ibuf.size;
                break;
        }
        idx += ibuf.size;
    }
}
```

Le code de démarrage des 9 transputeurs restants est aussi le même pour tous. Il est cependant différent des boucles de chargement précédentes puisqu'ils sont en bout de chaîne. Le désassemblage va s'avérer important pour la suite :

```
0x40000000 ajw   -3
0x40000002 mint                 ; set stack pointer
0x40000004 ldnlp 1024
0x40000007 gajw

0x40000009 ajw   -3             ; struct {uint32_t size; uint32_t dest; uint32_t unused;} ibuf;
0x4000000b ldlp  0
0x4000000c mint
0x4000000e ldnlp 4
0x4000000f ldc   12
0x40000010 in                   ; read(PORT_0, &ibuf, 12)
0x40000011 ldc   11
0x40000012 ldpi
0x40000014 mint
0x40000016 ldnlp 4
0x40000017 ldl   0
0x40000018 in                   ; read(PORT_0, 0x4000001f, ibuf.size);
0x40000019 ldc   3
0x4000001a ldpi
0x4000001c ldl   2
0x4000001d bsub                 ; A = 0x4000001f + ibuf.unused
0x4000001e gcall                ; call *A
0x4000001f j     0x40000020    ## <Transputeur11.tco> in main
0x40000020 ajw   3
0x40000021 ret
```

Les transputeurs de fin de chaîne recupèrent donc un dernier bloc de code, qui est écrit à une adresse bien spécifique en mémoire. La valeur du champ "unused" pour ce dernier bloc est 0xC. Pour tomber sur une vraie fonction, il est donc important que le code soit reconstruit correctement. Une dernière boucle d'extraction du code donnerait donc cela :

```
for (i = 4; i < 13; i++) {
    int idx = buf[i][0] + 1;
    int j;
    char tmp[512];
    memcpy(&ibuf, &buf[i][idx], 12);
    idx += 12;
    memcpy(tmp, &buf[i][idx], ibuf.size);
    memcpy(&buf[i][0x1f + 1], tmp, ibuf.size);
    buf[i][0] = 0x1f;
    buf[i][0] += ibuf.size;
}
```

L'adresse de la fonction appelée par GSUB est 0x1f (obtenu par LDPI + 3) additionnée au troisième champ de 32 bits obtenu lors de la lecture sur le port 0, et qui vaut 0xc pour les transputeurs 4 à 12. Cela donne donc une adresse de <base_addr+0x2b>. On peut désassembler à partir de cette adresse, ce qui nous donne :

```
0x4000002b ajw   -5
0x4000002d ldc   0
0x4000002e stl   1
0x4000002f ldc   0
0x40000030 ldlp  1
0x40000031 sb
0x40000033 ldc   12
0x40000034 stl   0
0x40000035 ldlp  2
0x40000036 mint
0x40000038 ldnlp 4
0x40000039 ldl   6
0x4000003a call  0x4000001f    ## <Transputeur4.tco> in main
[...]
```

On tombe bien sur un préambule de fonction valide. On peut aussi voir qu'une sous fonction à l'adresse 0x1f est appelée. Son rôle est de faire des IN. Le code assembleur précédent lis donc les 12 octets de la clef sur le port 0.

La génération des codes C debuggables par st20run peut se faire par :

```
$ make extract && ./extract input.bin
cc     extract.c   -o extract
Transputeur0.c: code size: 248
Transputeur1.c: code size: 112
Transputeur2.c: code size: 112
Transputeur3.c: code size: 112
Transputeur4.c: code size: 99
Transputeur5.c: code size: 99
Transputeur6.c: code size: 159
Transputeur7.c: code size: 119
Transputeur8.c: code size: 175
Transputeur9.c: code size: 103
Transputeur10.c: code size: 171
Transputeur11.c: code size: 131
Transputeur12.c: code size: 151
```

7.9 Rétro-ingénierie du code des transputeurs

Avec les information qu'on a obtenues, on peut désormais s'attaquer à une implémentation de l'algorithme de chiffrement en code natif. Le code s'avère relativement basique au final. Pour des raisons de clarté, le code obtenu par rétro-ingénierie est présenté directement en C sans l'assembleur ST20c2 qui rendrait cette section assez lourde.

7.9.1 Transputeur 0

Le code de ce transputeur a déjà été obtenu par rétro-ingénierie. La partie qui nous intéresse (après réécriture) est la suivante :

```
key_idx = 0;
read(0, key, 12);
while (1) {
    read(0, &curbyte, 1);
    out = curbyte ^ ((key[key_idx] << 1) + key_idx);
    write(1, &out, 1);

    t1 = T1(key);
    t2 = T2(key);
    t3 = T3(key);
    key[key_idx] = t1 ^ t2 ^ t3;
    key_idx++;
    key_idx %= 12;
}
```

7.9.2 Transputeurs 1, 2 et 3

Le code des transputeurs 1 à 3 est similaire et particulièrement simple à reconstruire :

```
return T4(key) ^ T5(key) ^ T6(key);

return T7(key) ^ T8(key) ^ T9(key);

return T10(key) ^ T11(key) ^ T11(key);
```

7.9.3 Transputeur 4

Dans ce code là, il faut bien faire attention à séparer les variables locales à la boucle principale des variables initialisées en début de fonction. En effet, l'octet renvoyé par ce transputeur est constamment mis à jour au cours du déchiffrement. Sa seule mise à 0 intervient lors du démarrage du transputeur :

```
int i;
static unsigned char c = 0;
for (i = 0; i < 12; i++) {
    c += key[i];
}
return c;
```

7.9.4 Transputeur 5

Le code est très similaire au précédent. L'opération d'addition est remplacée par un ou-exclusif :

```
int i;
static unsigned char c = 0;
for (i = 0; i < 12; i++) {
    c ^= key[i];
}
return c;
```

7.9.5 Transputeur 6

Le code se complexifie très légèrement. Mais grâce au debugger, on peut le tracer et obtenir le résultat suivant :

```
int i;
unsigned char c = 0, d, b0, b1;
static unsigned short sum = 0;
static int firstime = 1;

if (firstime) {
    for (i = 0; i < 12; i++) {
        sum += key[i];
    }
    firstime = 0;
}
b0 = ((sum & 0x8000) >> 15);
b1 = ((sum & 0x4000) >> 14);
d = b0 ^ b1;
sum <<= 1;
sum ^= d;
return sum&0xff;
```

7.9.6 Transputeur 7

Le code est assez similaire au transputeur 4, à la différence qu'ici il n'y a pas de variables initialisées hors de la boucle de déchiffrement :

```
int i;
unsigned char v1 = 0, v2 = 0;
for (i = 0; i < 6; i++) {
    v1 += key[i];
    v2 += key[6+i];
}
return v1 ^ v2;
```

7.9.7 Transputeur 8

Pour ce transputeur, le calcul se fait sur les 4 dernières clefs reçues. Il faut donc garder en mémoire les clefs utilisées. Pour le reste, c'est une variation du transputeur 4 :

```
static unsigned char kbuf[4][12];
static int currnd = 0;

unsigned char out = 0;
int i, j;

memcpy(kbuf[currnd], key, 12);
for(i = 0; i < 4; i++) {
    unsigned char c = 0;
    for(j = 0; j < 12; j++) {
        c += kbuf[i][j];
    }
    out ^= c;
}
currnd++;
currnd %= 4;
return out;
```

7.9.8 Transputeur 9

Le code de celui ci redevient plus simple. C'est une variation du code du transputeur 5 :

```
int i;
unsigned char c = 0;
for (i = 0; i < 12; i++) {
    c ^= (key[i] << (i & 7));
}
return c;
```

7.9.9 Transputeur 10

Comme pour le transpondeur précédent, celui ci travaille sur les 4 dernières valeurs de clefs reçues. Un index est calcule sur 4 fois 12 octets, et l'octet correspondant des clefs en mémoire sera retourné :

```
static unsigned char kbuf[4][12];
static int currnd = 0;

unsigned char out = 0;
int i, j;
unsigned char c = 0;

memcpy(kbuf[currnd], key, 12);
for (i = 0; i < 4; i++) {
    c += kbuf[i][0];
}
out = kbuf[c & 3][(c >> 4) % 12];

currnd++;
currnd %= 4;
return out;
```

7.9.10 Transputeurs 11 et 12

Ces deux transputeurs sont liés ensembles par leur port 1. Après analyse du code, il s'avère que ces deux transputeurs commencent par calculer un index d'octet de la clef (entre 0 et 11), puis s'échangent cet index. Finalement chacun d'eux renvoie l'octet de la clef correspondant a cet index. À noter que le

transputeur 11 travaille sur la clef courante pour le calcul de l'index tandis que le transputeur 12 travaille sur la clef précédente. Le code donnerait quelque chose comme ça :

```
static unsigned char kbuf[12];
uint8_t idx11, idx12;

idx12 = kbuf[1] ^ kbuf[5] ^ kbuf[9];
memcpy(kbuf, key, 12);
idx11 = key[0] ^ key[3] ^ key[7];
return key[idx11 % 12] ^ key[idx12 % 12];
```

7.10 Récupération de la clef

Nous avons maintenant tous les éléments pour calculer la clef. Il faut parcourir l'ensemble des clefs possibles qui se limite a 2^{26} possibilités pour obtenir un fichier bzip2 au moins partiellement valide en sortie :

```
int i, j, k;
const unsigned char *ktemplate = "\x5E\x54\x1B\x71\x56\x7C\x64\x7D\x69\x76\x00\x00";
unsigned char key[12];

for(k = 0; k < 1024; k++) {
    for(i = 0; i < 10; i++) {
        key[i] = ktemplate[i];
        if (k & (1 << i)) {
            key[i] |= 0x80;
        }
    }
    for(i = 0; i < 256; i++) {
        for(j = 0; j < 256; j++) {
            key[10] = i;
            key[11] = j;
            if (decrypt(data, key)) {
                int l;
                printf("Found possible key: ");
                for(l = 0; l < 12; l++) {
                    printf("%02x", key[l]);
                }
                printf("\n");
            }
        }
    }
}
```

L'oracle utilisé par la fonction `decrypt` se base sur le bitmap des symboles `huffman` utilisés. La vérification se fait dans la fonction `decrypt` afin de pouvoir renvoyer "false" dès qu'un octet ne correspond pas à ce qu'on attendrait suite au résultat du déchiffrement. Avec `i` l'index de déchiffrement courant, le code donnerait :

```
if (i >= 18 && out[i] != 0xff) {
    return 0;
}
```

Pour le reste de la fonction `decrypt`, on peut s'apercevoir que les transputeurs travaillent au maximum sur tous les octets des 3 dernières clefs ainsi que de la clef courante. On peut donc factoriser les boucles en deux boucles imbriquées de 0 à 3 et de 0 à 11. Il faut aussi penser à initialiser toutes les variables statiques identifiées lors de la phase précédente. L'initialisation donne :

```
memset(kbuf, 0, sizeof(kbuf));
s0 = 0;
c0 = 0;
c1 = 0;
round = 0;
memcpy(kbuf[round], key, 12);
for (i = 0; i < 12; i++) {
    s0 += kbuf[round][i];
}
```

Nous allons nous limiter a 32 octets à déchiffrer au maximum, ce qui donne donc le corps de la fonction `decrypt` :

```
for (i = 0; i < 32; i++) {
    register uint8_t tmp0, tmp1, tmp2, tmp3, tmp4, ki;

    out[i] = in[i] ^ ((kbuf[round][i % 12] << 1) + (i % 12));
    if (i >= 18 && out[i] != 0xff) {
        return 0;
    }

    ki = tmp0 = tmp1 = tmp2 = tmp3 = tmp4 = 0;
    for(k = 0; k < 4; k++) {
        tmp2 = 0;
        tmp3 += kbuf[(k + round) & 3][0];
        for (j = 0; j < 12; j++) {
            tmp2 += kbuf[(k + round) & 3][j];
            if (k == round) {
                c0 += kbuf[round][j];
                c1 ^= kbuf[round][j];
                ki ^= (kbuf[round][j] << (j & 7));
                if (j < 6) {
                    tmp0 += kbuf[round][j];
                    tmp1 += kbuf[round][j + 6];
                }
            }
        }
        tmp4 ^= tmp2;
    }
    ki ^= c0 ^ c1 ^ tmp0 ^ tmp1 ^ tmp4 ^ kbuf[tmp3 & 3][(tmp3 >> 4) % 12];

    tmp0 = ((s0 & 0x8000) >> 15) ^ ((s0 & 0x4000) >> 14);
    s0 = (s0 << 1) | tmp0;
    ki ^= s0 & 0xff;

    tmp0 = kbuf[(round - 1) & 3][1] ^ kbuf[(round - 1) & 3][5] ^ kbuf[(round - 1) & 3][9];
    ki ^= kbuf[round][tmp0 % 12];

    tmp1 = kbuf[round][0] ^ kbuf[round][3] ^ kbuf[round][7];
    ki ^= kbuf[round][tmp1 % 12];

    memcpy(kbuf[(round + 1) & 3], kbuf[round], 12);
    round++;
    round %= 4;
    kbuf[round][i % 12] = ki;
}
return 1;
```

L'attaque par bruteforce est maintenant prête. Un rapide benchmark montre une moyenne inférieure a 65 cycles/octet pour le déchiffrement et une petite 100-aine de cycles pour l'initialisation. Sur une machine a 2GHz, le parcours de l'espace de recherche devrait prendre moins d'une minute, et même plutôt de l'ordre de 45 secondes.

```
$ time ./bfkey
Found possible key: 5ed49b7156fce47de976dac5

real    0m44.087s
user    0m44.128s
sys     0m0.004s
```

Une clef a bien été trouvée, l'oracle était donc probablement correct. Le fait qu'une seule clef ait été trouvée est aussi très bien car nous n'avons pas besoin d'affiner la recherche.

7.11 Déchiffrement des données

Évidemment, le code qui a servi pour le bruteforce de la clef pourrait être utilisé pour déchiffrer le fichier. Cependant, lors du challenge, un émulateur avait été écrit, donc la solution ne serait pas complète sans qu'une section ne lui soit dédiée. L'émulateur a une moyenne d'environ 300000 cycles/octets pour le déchiffrement. Le cas où l'émulateur aurait été réellement utile est si le code tournant sur les ST20 avait été beaucoup plus complexe, et surtout obfusqué. En effet, il aurait pu être modifié assez facilement pour obtenir une exécution symbolique en plus de concrète. Mais finalement, le code n'étant pas obfusqué, le simulateur fourni dans la suite ST20 Embedded Toolset s'est trouvé largement suffisant.

L'architecture retenue pour l'émulateur, très discutable, consiste en la création d'une émulation simple ne gérant que les instructions utilisées par le code tournant sur les différents transputeurs, puis de le forker 12 fois en simulant les liens séries bidirectionnels synchrones par des pipes. Ce choix est très discutable car pour chaque lecture/écriture, plusieurs changements de contexte utilisateur/noyaux sont nécessaires. Chaque changement de contexte coute extrêmement cher (plusieurs 10-aines de milliers de cycles) et représentent 99.9% du temps passé par l'émulateur. Une bien meilleure solution aurait pu consister en l'utilisation de mémoire partagée protégée par des spinlock implémentés par des `cmpxchg`. Ca servira de leçon pour une prochaine fois.

La première étape consiste en la récupération des différentes instructions utilisées dans le code des transputeurs. C'est un processus itératif, qui donne la liste suivante :

- JUMP (0n)
- LDLP (1n)
- PFIX (2n)
- LDC (4n)
- LDNLP (5n)
- NFIX (6n)
- LDL (7n)
- ADC (8n)
- CALL (9n)
- CJMP (An)
- AJW (Bn)
- EQC (Cn)
- STL (Dn)
- MISC (Fn)
  - LB
  - BSUB
  - GCALL
  - IN
  - GT
  - WSUB
  - OUT
  - PROD
  - DUP
  - GAJW
  - LDPI
  - SHL
  - SHR
  - XOR
  - MINT

- AND
- REM
- RET
- SB
- SSUB

L'implémentation de ces instructions est directement issue du `ST20C2/C4 Core Instruction Set Reference Manual`. La description de ce que font les instructions peut se traduire directement en C. Ensuite, il faut maintenir un contexte pour le CPU. Les structures ci-dessous servent à l'initialisation et au cours de l'émulation :

```
struct channel {
    int rx[4];
    int tx[4];
};

struct cpureg {
    uint32_t Areg;
    uint32_t Breg;
    uint32_t Creg;
    uint32_t *Wptr;
    uint32_t *pWptr;
    uint8_t  *Iptr;
    uint8_t  *NextIptr;
    uint8_t  *readpos;
    uint8_t  *lastAddr;
    uint8_t  *membase;
    struct channel *chans;
    int      id;
};

int temu(struct channel *chans, int id);
```

Le reste de l'implémentation n'est pas spécialement passionnant. Il faut faire attention à bien fermer les bon `file descriptors` dans les processus créés ainsi que dans le parent.

Les canaux de communication étant implémentés par des pipes, le cas du port 0 du transputeur 0 peut se gérer avec les `stdin` et `stdout` directement. Cela permet d'invoquer directement l'émulateur en lui redirigeant le binaire du début dans son entrée standard. Il faut cependant penser à retirer les 24 premiers octets issus des premières `write` donnant l'état du démarrage et le début du déchiffrement :

```
$ ./temu/temu < input.bin | dd of=congratulations.tar.bz2 bs=8 skip=3 2>/dev/null
$ sha256sum congratulations.tar.bz2
9128135129d2be652809f5a1d337211affad91ed5827474bf9bd7e285ecef321  congratulations.tar.bz2
```

Il ne reste plus qu'à trouver l'adresse email, car évidemment ce n'est pas encore fini.


Félicitations !
SSTIC
SYMPOSIUM
SUR LA SÉCURITÉ DES TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION
... un dernier petit effort ?

8 Épilogue

8.1 Outils

- `grep`
- `zpipe`
- `hexdump`
- `ImageMagick`

8.2 congratulations.jpg

Une information est probablement cachée dans l'image. Pour ce format d'image, on peut chercher :
- De la stéganographie avec `stegdetect` / `outguess`
- Des commentaires avec `identify`
- Des informations rajoutées après le marqueur de fin "ff d9"

Les deux premiers tests ne donnant rien de concluant, on va tester la dernière :

```
$ grep -aobP '\xff\xd9' congratulations.jpg | cut -f1 -d':'
55246
127124
136547
```

Il n'est pas normal pour un fichier JPG d'avoir plusieurs marqueurs de fin, il y a donc des données rajoutées après le premier marqueur.

```
$ hexdump -s $((55246+2)) -n 16 -C congratulations.jpg
0000d7d0  42 5a 68 39 31 41 59 26  53 59 be ec b4 d2 00 92  |BZh91AY&SY......|
0000d7e0
```

On reconnait une signature bzip2. On peut donc extraire ce fichier :

```
$ dd if=congratulations.jpg of=jpg_file.tar.bz2 bs=1 skip=$((55246+2)) 2>/dev/null
damien@trantor:~/sstic15_sol/epilogue$ tar tvjf jpg_file.tar.bz2
-rw-r--r-- test/test    197557 2015-03-23 10:34 congratulations.png
```

On a de nouveau une image.

8.3 congratulations.png

Pour le PNG il n'existe pas beaucoup de solution de stéganographie. La commande `identify` ne remonte toujours rien d'intéressant, il faut donc aller chercher ailleurs.

La démarche va alors consister à parcourir les `chunks` du format PNG. On retrouve les `chunks` normaux : `IHDR`, `bKGD`, `pHYs`, `tIME`; puis un type de `chunk` non documenté : `sTic`. Sa taille est de 0x1337, ce qui n'est évidement pas un hasard. On peut déjà noter que le fichier caché par ce biais commence par les octets 78 9c, qui correspond à la sortie brute de zlib. Après avoir écrit un petit programme qui extrait l'ensemble des `chunks` "sTic", on peut passer le resultat directement dans `zpipe` :

```
$ make parse_png_sTic && ./parse_png_sTic congratulations.png | zpipe -d > png_file.tar.bz2
```

La sortie du fichier compressé par zlib donne un fichier au format bzip2, et même tar.bz2.
On obtient encore une image.

8.4 congratulations.tiff

En analysant rapidement l'image tiff, on peut s'apercevoir qu'elle n'est pas compressée. Cette fois ci la stéganographie a plus de chance de fonctionner. Le début de l'image est à l'offset 0x80, si on inspecte l'image on obtient :

```
$ hexdump -s $((0x80)) -n 128 -C congratulations.tiff
00000080  00 01 00 00 00 00 00 00  00 01 00 00 00 01 00 00  |................|
00000090  01 00 01 00 00 01 00 00  00 01 00 01 00 00 01 00  |................|
000000a0  00 00 00 00 00 00 00 01  01 00 01 00 00 00 01 00  |................|
000000b0  00 00 00 01 01 00 00 00  00 00 01 00 00 01 00 00  |................|
000000c0  00 00 00 00 00 00 01 00  00 01 00 00 01 00 01 00  |................|
000000d0  00 00 01 00 00 00 00 01  00 00 00 01 00 01 00 00  |................|
000000e0  00 01 00 00 01 00 00 00  00 01 01 00 00 01 00 00  |................|
000000f0  01 00 01 00 00 00 01 00  00 01 00 00 00 00 00 00  |................|
00000100
```

Les octets à 0x01 sont assez symptomatiques de la présence de stéganographie basique. Le format de l'image est RGB 24 bits. Il faut pouvoir identifier les canaux utilisés pour cacher de l'information :

```
$ hexdump -s $((0x80)) -vn 30 -e '3/1 "%02x " "\n"' congratulations.tiff
00 01 00
00 00 00
00 00 00
01 00 00
00 01 00
00 01 00
01 00 00
01 00 00
00 01 00
01 00 00
```

Même en cherchant plus loin, il semblerait que le canal bleu ne contienne pas d'information cachée. La démarche pour récupérer l'information cachée consiste alors à démarrer à l'offset 0x80 de l'image, puis de lire 3 caractères par 3 caractères et enfin d'extraire pour les 2 premiers octets le bit de poids faible pour générer un fichier valide :

```
$ make parse_tiff_stegano && ./parse_tiff_stegano congratulations.tiff > tiff_file.tar.bz2
```

Le résultat est au format tar / bzip2. Et pour changer, on obtient une image.

8.5 congratulations.gif

Pour le format GIF, le plus commun pour cacher des données est dans la palette de couleur. Après avoir testé toutes les combinaisons possibles pour cacher des informations directement dans la déclaration des couleurs de la palette, il faut se rendre à l'évidence. L'information n'est pas caché là. Si on extrait le flot LZMA de l'image pour le comparer à ce que donnerait la conversion du TIFF vers du GIF, on ne trouve pas d'information cohérente, ou du moins exploitable.

En inspectant cette fameuse palette de couleur, on peut noter une "anomalie" :

```
$ identify -verbose congratulations.gif | grep -A5 Colormap
  Colormap: 256
         0: (  0,  0,  0,255) #000000 black
         1: (  0,  0,  0,255) #000000 black
         2: (  0,  0,  0,255) #000000 black
         3: ( 11,  4,  2,255) #0B0402 srgba(11,4,2,1)
         4: (  7, 10,  6,255) #070A06 srgba(7,10,6,1)
```

La couleur noire est déclarée plusieurs fois. Techniquement, un GIF généré avec `gimp` pourrait avoir une palette ressemblant à ça. Cependant, on a déjà exclu toutes les autres possibilités. Donc on peut toujours tenter de creuser plus cette piste. Contrairement à ce que donnerait un GIF généré avec `gimp`, le nombre de couleurs déclarées à noire est tout de même très élevé :

```
$ identify -verbose congratulations.gif | grep -A256 Colormap | grep black | wc -l
54
```

C'est a peu près le nombre de caractères que contient une adresse email de challenge de SSTIC. On peut remarquer le bord noir autour de l'image, on va donc l'enlever pour voir ce qu'il se cache derrière. Il ne faudra pas longtemps pour le trouver, c'est la couleur 2. À l'aide d'un éditeur hexadécimal on peut la passer à blanc (ff ff ff), pour finalement obtenir :

1713e7c1d0b750ccd4e002bb957aa799@challenge.sstic.org

FIN.