Solution du challenge SSTIC 2013

par Raphaël Rigo

1 Introduction

Le challenge SSTIC 2013 ne faillit pas à la tradition et nous fournit un fichier à partir duquel il nous faudra retrouver une adresse mail, probablement après de nombreuses heures d'effort.

2 Une dissimulation un peu hasardeuse

2.1 Préliminaires

Une fois le fichier `dump.bin` téléchargé, les choses commencent classiquement par l'exécution de `file` pour identifier un potentiel format connu (heureusement que les concepteurs n'utilisent pas la fâcheuse tendance de `libmagic` à planter pour nous compliquer la tâche).

On constate donc tout de suite que le fichier est un pcap.

2.2 Canaux cachés

`Wireshark` va nous permettre de regarder ça de plus près. L'outil `Statistics -> Conversations` nous montre la présence de 3 connexions TCP différentes :

- une sur le port 1234 avec 884 octets échangés ;
- une connexion FTP ;
- une connexion sur des ports hauts, avec 849k échangés.

L'utilisation directe de la fonction *follow stream* nous permet de récupérer le contenu de la première connexion, qui est en texte :

Informations disponibles dans le pcap

```
Bonjour,
J'ai egare la cle pour dechiffrer mon carnet d'adresses.
Tu pourrais m'aider a la retrouver ? J'ai besoin de recuperer une adresse email
a l'interieur.
Pour t'aider, je t'envoie :
- une archive chiffree en AES par FTP
- la cle AES par canaux caches
voici l'iv utilise pour AES : 76C128D46A6C4B15B43016904BE176AC
voici le checksum de l'archive pour verifier le dechiffrement :
61c9392f617290642f9a12499de6b688
merci

PS :
Indication pour les canaux caches : 1 bit de canal cache temporel
concatene a 3 bits de canal cache non temporel.
```

Ce texte nous explique le contenu de la connexion FTP identifiée précédemment. L'extraction des données peut se faire également en utilisant `Wireshark` et permet de récupérer un fichier visiblement encodé en base 64, qui une fois décodé, semble chiffré.

Le texte nous indique également que la clé AES (quelle taille ? quel mode ?) est transmise par canal caché. Or, le pcap contient également 65 paquets ICMP, qui pourraient fort bien contenir les informations recherchées.

2.3 Analyse et divination

L'information "1 bit de canal temporel", conjuguée à la présence de 65 paquets, fait penser à l'utilisation de l'espacement entre deux paquets comme stockage d'information. Une passe de `tshark -r dump.bin icmp` permet de rapidement voir que les paquets sont distants de 1 ou 2 secondes, ce qui nous fournit bien 1 bit d'information. Nous ne savons néanmoins pas si un écart d'une seconde correspond à un bit à 0 ou à 1.

Cette même commande permet également de constater que les valeurs de TTL pour les 64 premiers paquets prennent 4 valeurs différentes : 10, 20, 30 et 40. Ce qui fournit 2 bits supplémentaires.

Reste donc 1 bit d'information à trouver dans les données. Une analyse des différences entre paquets montre qu'en dehors des champs évidents (*checksum* et données temporelles), seuls les champs DSCP IP et l'ID ICMP changent.

J'ai d'abord cherché toutes les combinaisons de 3 bits parmi les 24 de ID et TTL concaténés, mais cette solution se conclut par un échec.

Le champ DSCP variant entre seulement 2 possibilités, il semble finalement être une bonne option.

Reste maintenant le travail inintéressant de savoir comment concaténer le tout, et avec quelle correspondance entre les différences constatées et les bits utilisés.

Un petit programme en C va tester tout ça, afficher les clés potentielles sur la sortie standard qui seront utilisées pour appeler `openssl` :

Code de test des clés

```
#!/bin/bash
i=0
cat keys.txt | while read k ; do
        openssl aes-256-cbc -iv 76..AC -K $k -d -in file.gz -out /dev/null
        if [ $? -eq 0 ] ; then
          openssl aes-256-cbc -iv 76..AC -K $k -d -in file.gz -out "test$i"
        fi
done
```

Vu qu'openssl vérifie le *padding*, seules quelques clés vont rester, il suffit ensuite de vérifier le MD5 des fichiers produits.

Finalement, on trouve la clé grâce à la correspondance ($10 \rightarrow 1$, $20 \rightarrow 3$, $30 \rightarrow 2$, $40 \rightarrow 0$) et la concaténation "temps | ttl | dscp" :

`dd8cf2d52e69aafb734e3acd0e4a69e83ed93bc4870ecd0d5b6faad86a63ae94`

3 TEL THÉSÉE DANS SON LABYRINTHE

3.1 Identification

Le fichier déchiffré lors du premier niveau est une archive compressée *tar.gz* qui contient :

– 2 fichiers Python ;
– un fichier `data` apparemment chiffré ;
– un fichier texte `s.ngr` apparemment lié à une application Xilinx.

Une recherche rapide montre que le fichier NGR correspond à la représentation RTL (*Register-transfer level*) d'un circuit électronique. Celui-ci est un format propriétaire qu'il est possible d'ouvrir avec les outils Xilinx, téléchargeable sur leur site dans une version d'évaluation ou limitée en fonctionnalités. Le temps de télécharger les 7 Go nécessaires nous laisse le loisir d'étudier les fichiers Python.

Le fichier `decrypt.py` ne semble pas prévu pour s'exécuter, car il manque un module `dev` (peut-être fourni par Xilinx ?). Cependant, son code est très simple : il déchiffre le fichier `data` à l'aide d'un `dev`, initialisé avec un fichier NGR [1], auquel il envoie, en plus des données, un tableau de 231 octets, `smp`, et une clé de 16 octets. Il vérifie le clair obtenu à l'aide d'un MD5 puis décode la base 64 que le clair est censé contenir.

Il va donc falloir analyser `s.ngr` pour savoir comment retrouver la clé. Il est d'ailleurs dommage de ne pas disposer du fichier NGC correspondant, qui aurait pu permettre de simuler le circuit et de rentre la rétro conception plus facile.

3.2 Un circuit qui nous donne du fil à retordre

Xilinx ISE nous permet d'afficher le circuit correspondant et de l'analyser en parcourant les différentes briques qui le constituent. Ici, il est évidemment nécessaire de connaître ou d'apprendre les bases des circuits logiques, sur lesquelles je ne vais pas revenir.

L'éditeur présente l'avantage de pouvoir facilement sélectionner les liaisons afin de suivre les signaux, mais également de rajouter les étiquettes correspondant aux sources des signaux, ce qui permet une compréhension plus rapide. Une fonctionnalité également utile est l'affichage des signaux soit par bus soit bit à bit. Si la première rend l'affichage plus synthétique, la deuxième peut permettre de clarifier certaines ambiguïtés.

L'un des premiers constats est que le circuit `s` (voir figure 1) possède des entrées/sorties qui correspondent au code Python.

Ensuite, l'affichage du détail du circuit permet de voir quelques blocs aux noms intéressants :
– `ip` ;
– `r` ;
– `accu`.

Un *instruction pointer* ? Des registres ? Un accumulateur ? Effectivement, si l'on va voir le détail de ces circuits, `r` contient 8 registres, `accu` un seul et `ip` aussi. `s` serait donc un processeur, dont les registres et bus d'adressages fonctionneraient uniquement en 8 bits.

Si nous sommes bien en présence d'un processeur, alors nous devrions trouver une unité arithmétique (ALU, pour *Arithmetic Logic Unit*), de la mémoire et une logique permettant de décoder les instructions.

La principale difficulté ici est de ne pas tomber dans le piège de `smd` et `smp` et de les regarder trop dans le détail. Il s'agit en effet de simples mémoires ; leurs nombreuses portes sont simplement la logique d'adressage des 256 octets disponibles.

Pour analyser en détail le fonctionnement, j'ai simplement imprimé les circuits et étudié le tout sur papier.

3.2.1 Opcodes

La première chose à faire est de déterminer quels sont les *opcodes* de ce processeur. Vu le code Python, on se doute que le code du programme va être stocké dans `smp`, ce qui est confirmé par le fait que le flag d'écriture `smp_ctrl_w` n'est contrôlé que par l'extérieur du circuit, le code ne pourra donc pas être automodifiant et doit être chargé de l'extérieur.

Une fois `smp` identifié, on sait que les *opcodes* vont être portés par le signal `smp_data_r`. Il "suffit" donc de

1. ce qui est impossible, car il ne s'agit pas d'un *bitstream*.

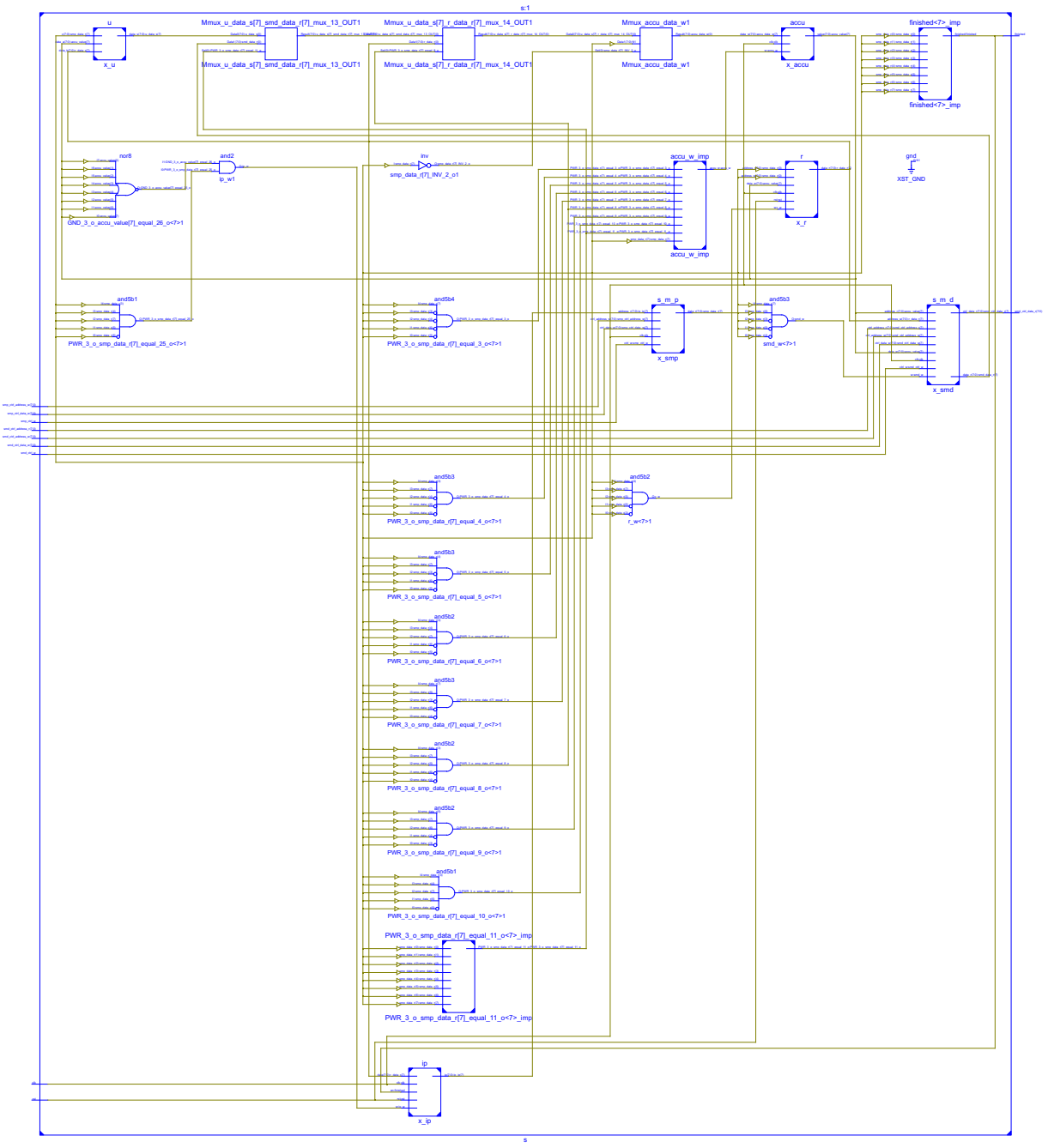

FIGURE 1 – Schéma RTL du fichier `s.ngr`

suivre les fils pour retrouver leur utilisation.

Je ne vais pas rentrer dans le détail de chaque fil, car la tâche n'est pas vraiment complexe, juste un peu fastidieuse. On constate néanmoins que les 5 bits de poids fort sont principalement utilisés dans le sous circuit `x_u` et dans des portes AND qui permettent de tester les *opcodes* afin d'activer ou non la mise à jour des registres ou de la mémoire. Les 3 bits de poids faible servent à adresser les 8 registres.

`x_u` est manifestement une unité arithmétique : elle calcule en fonction de l'accumulateur et du registre sélectionné par l'*opcode* et sa sortie est utilisée pour mettre à jour l'accumulateur.

Enfin, pour comprendre parfaitement les *opcodes*, il faut également déterminer lesquels activent l'écriture dans les différents éléments modifiables : registres, accumulateur et `smd`.

L'analyse donne finalement (en omettant les *opcodes* qui ne sont pas présents dans `smp.py`) :

Liste des opcodes

```
0x00 : mov acc, imm
0x88 : and acc, r
0x90 : or  acc, r
0xa0 : not acc
0xa8 : mov acc, r
0xb0 : mov r, acc
0xb8 : jz  r
0xc0 : mov [r], acc
0xc8 : end
0xd0 : mov acc, [acc]
0xd8 : shl r, 1
0xe0 : or  acc, 0x80
```

Quelques instructions sont légèrement plus difficiles à comprendre :
- `shl` et `or acc, 0x80` nécessitent d'activer la vue par bit dans l'ALU, car sinon il est très facile de se tromper ;
- les accès mémoire nécessitent de bien regarder la source et la destination des données.

Il ne reste plus qu'à implémenter tout ça dans un module Metasm pour analyser le code.

Il pourrait être intéressant d'utiliser la fonction de désobfuscation de Metasm pour simplifier le code. Certaines opérations sont peu lisibles du fait des limitations du processeur ; par exemple : `mov acc, 76h; or acc, 80h` qui peut être simplifié en `mov acc, 0F6h`. Ou encore `mov acc, 0; jz reg` qui donne `jmp reg`, `mov acc, reg1; XXX acc, reg2; mov reg1, acc` qui donne `XXX reg2, reg2`.

3.3 Crypto ou pas ?

Je ne vais pas recopier ici le code désassemblé, car, non optimisé, il ne présente que peu d'intérêt. Une nouvelle fois, la tâche n'est pas extrêmement difficile, mais un peu longue. Quelques petites astuces permettent de comprendre bien plus rapidement le programme.

3.3.1 loc_71h

Cette adresse est référencée à de nombreux endroits du programme. La compréhension de son action est donc essentielle. Un bon moyen est de traduire littéralement le code en C, afin de pouvoir l'appréhender dans sa globalité :

Traduction en C de loc_71h

```
loc_71h:
{
        r1 = r3 = 0;
```

```
        r2 = 1;
        while(r2) {
                if(r7 & r2) {
                        if(r6 & r2) {
                                if (r1) {
                                        r3 |= r2;
                                } else {
                                // loc_8fh:
                                        r1 = r2;
                                }
                        } else {
                                // loc_96h:
                                if (r1) {
                                        r1 = r2;
                                } else {
                                // loc_a2h:
                                        r3 |= r2;
                                        r1 = 0;
                                }
                        }
                } else {
                        // loc_ach:
                        if (r6 & r2) {
                                if (r1) {
                                        r1 = r2;
                                } else {
                                // loc_beh:
                                        r3 |= r2;
                                        r1 = 0;
                                }
                        } else {
                                // loc_c8h:
                                if (r1) {
                                        r3 |= r2;
                                        r1 = 0;
                                } else {
                                // loc_d7h:
                                        r1 = 0;
                                }
                        }
                }
                r1 <<= 1;
                r2 <<= 1;
        }
        r7 = r3;
}
```

Une fois traduit, le code est beaucoup plus simple :

- `r2` est utilisé pour tester les bits de `r7` et `r6` ;
- `r3` sert visiblement de résultat ;
- `r1` est mis à jour en fonction des bits de `r6` et `r7`.

Les conditions sur l'utilisation et la mise à jour de `r1` nous donnent la solution : loc_71h est une addition. `r6` et `r7` sont additionnés, `r1` sert de retenue et `r3` stocke temporairement le résultat.

Ceci nous donne une bonne partie de la solution.

3.3.2 Mais OU ET mon XOR ?

Une autre partie intéressante commence à l'adresse 24h : on y trouve des instructions permettant de lire la mémoire. Si l'on traduit une nouvelle fois le code en C :

Traduction de loc_24h

```
        r7 = 0x11+r0;
loc_24h:
        r6 = data[r7]|data[r0&0xF]
        acc = ~(data[r7]&data[r0&0xF])
        r7 = r6 & acc;
loc_53h:
        r4 = 0;
        r5 = 1;
        while(r5) {
                r4 <<= 1;
                if(r5 & r7) {
                        r4 |= 1;
                }
                r5 <<= 1;
        }
        r7 = r4;
        data[r1] = r7;
```

Cette partie n'est pas trop difficile à comprendre si on regarde bien les entrées : les données sont constituées de cette façon (`key` faisant 16 octets) :

```
   smd = d[i : (i + 224)]
smd = (key, len(smd), smd)
```

Or, le début de la fonction accède aux 16 premiers octets (`r0&0xF`), la clé donc. Mais également aux données, `r0` servant d'index.

Écrire la table de vérité des opérations binaires effectuées entre la clé et les données (voir figure 3.3.2) nous permet de constater qu'il s'agit en fait d'un `xor`.

a	b	r
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 2 – Table de vérité de l'opération $r = (a \vee b) \wedge (\neg(a \wedge b))$

La deuxième partie est assez simple : elle inverse l'ordre les bits de `r7`. Finalement, tout ceci se résume à : `data[r7] = reverse(data[r7] xor key[r0&0xF])`.

Il ne reste plus qu'à comprendre comment évoluent `r7` et `r0`.

3.3.3 Boucle principale

La boucle principale est assez simple une fois que l'addition et la "crypto" ont été identifiées. On constate que `r5` sert à stocker l'adresse où le code va atterrir après les appels à l'addition [2]. Donc, même méthode :

Pseudo code de la boucle principale

2. on peut donc considérer les `mov r5, next; mov acc, 0; jmp reg; next:` comme un `call reg`.

```
        r0 = 0;
        while(1) {
                // loc_2h
                r7 = data[0x10]; //longueur
                r6 = ~r0;
                r7 += r6;
                // loc_0eh
                r7++;
                // loc_16h
                if(r7 == 0)
                        return;
                // loc_1ah
                r7 = 0x11;
                r7 += r6;
                crypt();
                // loc_42h
                r7 = r0 + 1;
                // loc_4ch:
                r0 = r7;
        }
```

Seule petite subtilité ici : le `not` de `r0` suivi d'une addition à `r7` puis d'une incrémentation. Si l'on inverse l'ordre, on retrouve :

- `not` de `r0`
- incrémentation de `r0`

Ce qui correspond au calcul standard de l'opposé de `r0`. L'opération réalisée en pratique est : `r7 -= r0`.

Et on peut réécrire le code complet en C :

Code de déchiffrement

```
unsigned char decrypt(unsigned char *key, int len, unsigned char *data)
{
        int i;
        for (i=0; i < len; i++) {
                data[len-1-i] = reverse(data[len-1-i]^key[i&0xF]);
        }

}
```

Reste à "casser" la crypto et à retrouver la clé de chiffrement.

3.4 Pff, tout ça pour ça.

Donc, nous connaissons l'algorithme qui utilise la clé de 16 octets pour déchiffrer le fichier `data`. Mais disposer d'informations sur le texte clair permettrait de retrouver la clé plus efficacement. Or, nous avons vu dans `decrypt.py` que celui-ci tente de décoder le clair comme étant encodé en base 64 ; soit une belle condition d'arrêt du bruteforce. L'idée est donc de tester tous les octets possibles de la clé et de vérifier que le clair obtenu est toujours un caractère de la base 64 ou un retour chariot, en espérant obtenir le minimum de candidats.

Code de recherche de la clé

```
for (i=0; i<16; i++) {
        for(k=0; k<256; k++) {
                loop = 0;
                while(loop*16+i < 44053) {
                        x = reverse(file[loop*16+i] ^ k);
                        file2[loop*16+i] = x;
```

```
                        if(isgoodchar(x)) {
                                loop = 0;
                                break;
                        }
                        loop++;
                }
                if (loop) {
                        key[i] = k;
                        break;
                }
        }
}
printf("%s\n", file2);
```

Finalement, vu la simplicité de l'algorithme utilisé, je pense qu'il aurait été relativement facile de décrypter `data` en boîter noire, et probablement beaucoup plus rapide ! Mais bon, ça fait partie du jeu.

Le résultat, une fois la base 64 décodée, est un fichier PostScript...

4 PostScript

PostScript, peu utilisé de nos jours, est en fait un langage complet de description de document, qui embarque une machine virtuelle, basée sur la pile. Donc, ce niveau va être un bonheur pour ceux adepte de la notation polonaise inversée et un enfer pour les autres.

Une bonne documentation est nécessaire pour avoir la définition de toutes les opérations utilisées. Je me suis basé sur le *bluebook*[3].

4.1 Un code définitivement hideux

Histoire de compliquer les choses, le code du fichier est sur une seule ligne. Il faut donc commencer par le réindenter. J'ai opté pour une approche manuelle, même si l'opération était facilement automatisable, car je me suis dit que cela me permettrait de saisir un peu ce que faisait le code par la même occasion.

Lancer le PostScript avec *Ghostscript* donne le résultat suivant :

Lancement du stage3.ps

```
$ gs -q stage3.ps
missing '--' preceding script file
usage: gs -- script.ps key
$ gs -q stage3.ps -- test
missing '--' preceding script file
usage: gs -- script.ps key
```

La clé proposée n'est pas acceptée. Il va falloir regarder le code de plus près. D'abord, 4 variables sont définies : `I1` à `I4`, il s'agit de chaînes spécifiées en hexadécimal et référencées par la suite. Mais cherchons d'abord à obtenir une clé "valide". Le code de vérification est le suivant :

Premier test sur la clé

```
mark % définie une marque pour compter
shellarguments
{ % true
```

3. http://www-cdf.fnal.gov/offline/PostScript/BLUEBOOK.PDF

```
counttomark 1 eq
  {
    % true (1 argument)
    dup
    length
    exch
    /ReusableStreamDecode filter
    exch
    2 idiv
    string
    readhexstring
    pop  % la stack contient la clé
    dup length 16 eq  % check que la clé décodée fait 16 octets
...
```

La clé doit donc faire 16 octets encodés en hexadécimal. Si l'on passe une telle clé, le programme quitte sans message. Voyons donc la suite. Analyser du PostScript est une horreur, aucun débugger libre et fonctionnel ne semble exister. Le moyen le plus efficace reste de rajouter des opérateurs qui permettent d'afficher des éléments ou tout le contenu de la pile. Par exemple `qstack` affiche toute la pile en détaillant la valeur des chaînes de caractères[4].

4.1.1 Code de déchiffrement de `I2` et `I4`

Finalement, le code de déchiffrement de `I2` et `I4`, transcrit littéralement est le suivant :

Premier test sur la clé

```
void decrypt(unsigned char *key, unsigned char *out, unsigned char *data, int len)
{
        int i;

        key[2] = key[0];
        key[3] = key[1];
        for(i=0;i<len; i+=4) {
                out[i] = data[i]^key[0];
                out[i+1] = data[i+1]^key[1];
                out[i+2] = data[i+2]^key[2];
                out[i+3] = data[i+3]^key[3];
                key[0] = out[i];
                key[1] = out[i+1];
                key[2] = out[i+2];
                key[3] = out[i+3];
        }
}
```

La clé utilisée pour déchiffrer `I2` est spécifiée par les troisièmes et quatrièmes octets de la clé, par les deux premiers pour `I4`.

4.1.2 Bruteforce des deux clés

Nous savons que `I2` et `I4` sont du code PostScript car les mots clés `cvx exec` permettent de rendre la variable exécutable et d'interpréter ensuite leur contenu. La condition d'arrêt du bruteforce est donc l'obtention de PostScript, que l'on peut résumer en : obtenir de l'ASCII et avoir des mots clés PostScript.

Le résultat de ce bruteforce est : `bac9f7a8` pour les quatre premiers octets.

4. Technique élite également connue dans d'autres langages comme "foutre des `printf` partout".

$$lfsr = update\_lfsr(lfsr)$$

$$décalage = \begin{cases} 1 & \text{si } lfsr \leq 55555555_{16} \\ -1 & \text{si } lfsr \in]55555555_{16}, AAAAAAAA_{16}] \\ 0 & \text{si } lfsr > AAAAAAAA_{16} \end{cases}$$

$$curseur = curseur + décalage$$

FIGURE 3 – Valeur du décalage

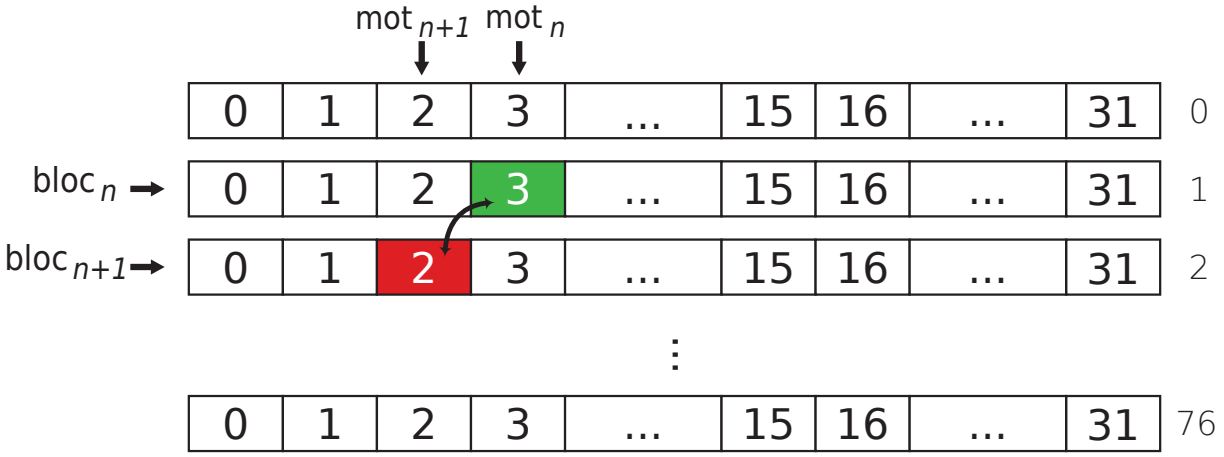

FIGURE 4 – Schéma de la partie d'échange des blocs

4.2 Encore plus de PostScript

Une fois I2 et I4 déchiffrés, le code obtenu laisse perplexe, particulièrement I2, dont la complexité rebute.

4.2.1 I2 : intuition

On notera certains noms de variables assez explicites : `F, a, b, c, d` mais surtout des constantes (ici en octal) bien connues. Tout ceci nous fait penser à MD5. Pour vérifier cette intuition, ce code a été rajouté : `(000000) calc pstack`, afin de comparer le résultat à celui de `md5sum`. Il s'agit bien d'un MD5.

4.2.2 I4 : Arg !

`I4` étant assez long à analyser, je ne vais pas rentrer dans le détail de chaque étape. Le code de déchiffrement de `I1` est en réalisé basé sur un LFSR (*Linear Feedback Shift Register*) qui sert de générateur de nombres pseudo aléatoires. Celui-ci est simple :

Générateur de pseudo aléa

```
uint32_t update_lfsr(uint32_t s)
{
        uint32_t s2;

        s2 = (s>>7)^(s>>3)^(s>>2)^s;
        s >>= 1;
        if (s2&1)
                s |= 0x80000000;
        return s;
}
```

Ce LFSR est utilisé pour deux choses :
- faire évoluer la clé de chiffrement ;
- déplacer des curseurs.

La phase d'initialisation est la suivante :

1. découper `I1` en 77 blocs de 128 octets (soit 32 mots de 4 octets) ;
2. initialiser un curseur de bloc à 0 ;
3. initialiser un curseur de mots à 0 ;
4. initialiser le LFSR avec la clé.

L'algorithme de déchiffrement est le suivant :

1. répéter 10240 fois :
   (a) itérer le LFSR et mettre à jour le curseur de bloc (voir figure 3),
   (b) itérer le LFSR et mettre à jour le curseur de mots (voir figure 3),
   (c) échanger le bloc pointé par les nouveaux curseurs avec celui pointé par les anciens (voir figure 4),
   (d) itérer le LFSR,
   (e) xorer le bloc pointé par les anciens curseurs avec l'état du LFSR ;
2. calculer le MD5 du résultat et comparer avec l'empreinte stockée dans `I3`.

Le tout pour les 6 fois 2 octets de clé restants, `I3` stockant six MD5.

Code C de déchiffrement

```
void decrypt(uint8_t *i1, uint8_t *out, uint32_t k, int iblk,
                          int ioff, int *outblk, int* outoff)
{
        int i;
        int blk, off, new_blk, new_off;

        blk = new_blk = iblk;
        off = new_off = ioff;
        memcpy(out, i1, 128*77);

        for(i=0; i<10240; i++) {
                k = update_lfsr(k);
                new_blk = mod(blk+next(k), 77);
                k = update_lfsr(k);
                new_off = mod(off+next(k), 32);
                k = update_lfsr(k);

                swap(out, blk, off, new_blk, new_off);
                xor(out, blk, off, k);

                blk = new_blk;
                off = new_off;
        }
        *outblk = blk;
        *outoff = off;
}
```

4.2.3 Bruteforce

La réimplémentation de ce code en C nous permet de bruteforcer les octets restants de la clé en 1 minute environ.

Bruteforce de la clé en C

```
c = 0;
sblk = soff = 0;
k = 0xf7a80000;
while(c<6 && low != 0x10000) {
        for(low=0x0000; low<0x10000; low++) {
                decrypt(i1, tmp, k|low, sblk, soff, &blk, &off);

                md5(tmp, 77*128, md5res);;
                if(!memcmp(md5res, md5ok+c*16, 16)) {
                        printf("Victory ! %x\n", k|low);
                        k=low<<16;
                        sblk = blk;
                        soff = off;
                        c++;
                        memcpy(i1, tmp, 128*77);
                        break;
                }
        }
}
```

Il est évident qu'il était possible de bruteforcer les octets de la clé en utilisant le code PostScript. Néanmoins, vu la lenteur du lancement de gs, du nombre de possibilités et du fait que seule une personne avait validé le challenge, je me suis dit que la compréhension serait probablement plus rapide. En effet, environ 4 ou 5 heures m'ont été nécessaires à la résolution de cette partie alors que le bruteforce aurait duré probablement au moins 12 heures sur ma machine.

5 Blague finale

Le fichier déchiffré au final est un fichier vCard. Malheureusement, celui-ci ne contient pas directement l'adresse mail, mais à la place une série de *syscall* Linux. Il suffit probablement de réencoder la chaîne en utilisant les numéros de *syscall*. Le fichier `arch_x86_syscalls_syscall_64.tbl` des sources du noyau contient les numéros pour l'architecture `amd64`. Quelques lignes de script nous donnent l'adresse mail :

`59575e0e71f1e3e9946bc307fc7a608d0b568458@challenge.sstic.org`.

6 Conclusion

Malgré un premier niveau un peu trop basé sur la devinette, la suite était particulièrement intéressante. La rétro conception d'un processeur à partir de sa logique était inattendue et le reverse du code exécuté également intéressant. Dommage que l'algorithme était en réalité si simple. La partie PostScript était un peu longue mais le fait que la machine virtuelle soit basée sur la pile uniquement rendait la chose assez exotique. Au final, un challenge de qualité, grâce à la diversité des sujets abordés, mais aussi par sa progressivité, aucun passage ne nécessitant des connaissances poussées. Encore une bonne année, merci aux concepteurs :)