Challenge SSTIC 2012 : solution

Julien Perrot

11 mai 2012

Résumé

Ce document présente une démarche possible pour résoudre le challenge SSTIC 2012. L'objectif de celui-ci est de déchiffrer un fichier « secret » afin d'y retrouver une adresse email en `@sstic.org`.

Deux étapes indépendantes, qui sont la cryptanalyse d'une implémentation de DES « white box » et l'analyse d'une ROM programmée sur une webcam, permettent de retrouver chacune une moitié de la clé requise pour le déchiffrement du « secret ».

Cependant, l'image disque fournie pour le challenge étant corrompue, il est nécessaire de reconstruire ce fichier avant de réussir à le déchiffrer correctement.

Le code source développé dans le cadre de ce challenge est disponible en annexe à la fin de ce document.

Table des matières

Table des figures

Liste des tableaux

1 Découverte du challenge

1.1 Identification du fichier challenge et montage de la partition

Le fichier challenge est téléchargeable à l'adresse http://static.sstic.org/challenge2012/challenge.

D'après la commande file, il s'agit d'un fichier au format gzip :

```
$ file challenge
challenge: gzip compressed data, was "dump.img", from Unix,\
    last modified: Fri Mar 23 10:11:37 2012
```

Il est donc possible de décompresser ce fichier :

```
$ mv challenge dump.img.gz
$ gunzip dump.img.gz
$ ls -al dump.img
-rw-r--r-- 1 jpe jpe 1073741824 2012-03-23 10:32 dump.img
```

Le fichier dump.img pèse environ 1go. Il s'agit d'une image disque :

```
$ file dump.img
dump.img: x86 boot sector; partition 1: ID=0x83, active, starthead 1, startsector 63,\
    2088387 sectors, code offset 0xb8
```

L'outil parted permet de consulter le contenu de la table de partitions :

```
$ parted dump.img
WARNING: You are not superuser.  Watch out for permissions.
GNU Parted 2.3
Using dump.img
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) unit
Unit?  [compact]? B
(parted) print
Model:  (file)
Disk dump.img: 1073741824B
Sector size (logical/physical): 512B/512B
Partition Table: msdos

Number  Start   End          Size         Type     File system  Flags
 1      32256B  1069286399B  1069254144B  primary  ext2         boot
```

L'extraction de la partition primaire au format ext2 est réalisée à l'aide de la commande dd :

```
$ dd if=dump.img of=dump.part bs=32256 skip=1
33287+1 enregistrements lus
33287+1 enregistrements ecrits
1073709568 octets (1,1 GB) copi\'es, 4,41739 s, 243 MB/s
$ file dump.part
```

```
dump.part: Linux rev 1.0 ext2 filesystem data, UUID=5712fd31-bf27-4376-bbf4-aabab500ba7b\
    (large files)
$ md5sum dump.part
f8afccd7d3b0dea1d10adb03045ca247  dump.part
```

La partition extraite est alors montée :

```
$ sudo mount -o loop dump.part /mnt/loop
$ ls /mnt/loop
bin  boot  dev  etc  home  lib  lost+found  media  mnt  opt  proc  root  sbin selinux  srv\
    sys  tmp  usr  var
$ ls /mnt/loop/home
sstic
$ ls /mnt/loop/home/sstic
irc.log  secret  ssticrypt
```

Le contenu du fichier `irc.log` est le suivant :

```
#sstic-challenge: <lobster_dog> I've a problem
#sstic-challenge: <lobster_dog> lobster cat is trying to steal one of my files
#sstic-challenge: <blue_footed_booby> lobster cat ?
#sstic-challenge: <lobster_dog> http://amazingdata.com/mediadata56/Image/2007_0921honeymoon0141_animal\
    _fun_weird_interesting_2009080319215810225.jpg
#sstic-challenge: <blue_footed_booby> k -_-
#sstic-challenge: <blue_footed_booby> gimme your file, I'll hide it
#sstic-challenge: <lobster_dog> k
#sstic-challenge: <blue_footed_booby> your data is secure ;) I just finished the encryption system
#sstic-challenge: <lobster_dog> ok, I secure erase it on my side!
#sstic-challenge: <blue_footed_booby> unfortunately my hard drive is making strange noises right now ... :(
-!- blue_footed_booby [~booby@galapagos] has quit [Read error: Connection reset by peer]
#sstic-challenge: <lobster_dog> ...
```

Il s'agit d'un extrait de conversation IRC où il est question de protéger un fichier à l'aide d'un système de chiffrement. Malheureusement, un des deux correspondants semble rencontrer des problèmes matériels avec son disque dur qui provoquent une déconnexion soudaine du serveur IRC.

La commande `file` nous précise que le fichier `ssticrypt` est un exécutable MIPS big-endian au format ELF :

```
$ file /mnt/loop/home/sstic/ssticrypt
/mnt/loop/home/sstic/ssticrypt: ELF 32-bit MSB executable, MIPS, MIPS-I version 1 (SYSV),\
    statically linked, stripped
```

La commande `readelf` doit permettre d'obtenir de l'information sur l'exécutable mais apparemment celui-ci semble corrompu :

```
$ readelf -a /mnt/loop/home/sstic/ssticrypt
readelf: Error: Unable to read in 0x28 bytes of section headers
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
```

```
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x400c40
  Start of program headers:          52 (bytes into file)
  Start of section headers:          305184 (bytes into file)
  Flags:                             0x1007, noreorder, pic, cpic, o32, mips1
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         39
  Section header string table index: 36
readelf: Error: Unable to read in 0x618 bytes of section headers
readelf: Error: Section headers are not available!
```

La corruption du fichier est sans doute lié aux problèmes de disque dur mentionnés dans l'extrait de conversation IRC.

Après démontage de la partition, l'appel au programme `fsck.ext2` permet de corriger les problèmes au niveau de la partition :

```
$ sudo umount loop
$ fsck.ext2 -f dump.part
e2fsck 1.41.14 (22-Dec-2010)
Pass 1: Checking inodes, blocks, and sizes
Inode 14, i_blocks is 2064, should be 24.  Fix<y>? yes

Inode 19, i_size is 128, should be 311296.  Fix<y>? yes

Inode 40820, i_size is 236, should be 40960.  Fix<y>? yes

Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 3A: Optimizing directories
Pass 4: Checking reference counts
Pass 5: Checking group summary information
Block bitmap differences:  -(26628--26882)
Fix<y>? yes

Free blocks count wrong for group #0 (31843, counted=32098).
Fix<y>? yes

Free blocks count wrong (232969, counted=233224).
Fix<y>? yes

dump.part: ***** FILE SYSTEM WAS MODIFIED *****
dump.part: 5469/65280 files (1.5% non-contiguous), 27824/261048 blocks
```

Après remontage de la partition corrigée, `readelf` arrive à correctement analyser le binaire `ssticrypt`, notamment au niveau des entêtes des sections (qui posaient problème auparavant) :

```
$ sudo mount -o loop dump.part /mnt/loop
$ md5sum /mnt/loop/home/sstic/ssticrypt
6f94e9a0739ee8950f88adb3d023bbaf  /mnt/loop/home/sstic/ssticrypt
```

```
$ readelf -S /mnt/loop/home/sstic/ssticrypt
There are 39 section headers, starting at offset 0x4a820:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        00400134 000134 00000d 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            00400144 000144 000020 00   A  0   0  4
```

Pour tester le programme, n'ayant pas à ma disposition une machine MIPS, j'ai utilisé l'émulateur qemu avec une image de Debian Lenny disponible à l'adresse http://people.debian.org/~aurel32/qemu/mips/ :

```
$ sudo apt-get install qemu-system
$ qemu-system-mips -M malta -kernel vmlinux-2.6.26-2-4kc-malta -hda debian_lenny_mips_standard.qcow2\
    -append "root=/dev/hda1 console=ttyS0" -nographic
$ QEMU 0.15.50 monitor - type 'help' for more information
(qemu) QEMU 0.15.50 monitor - type 'help' for more information
(qemu)
Could not open option rom 'pxe-pcnet.rom': No such file or directory
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 2.6.26-2-4kc-malta (Debian 2.6.26-26lenny1) (dannf@debian.org) (gcc version\
    4.1.3 20080704 (prerelease) (Debian 4.1.2-25)) #1 Sat Nov 27 11:32:29 UTC 2010
[...]
Debian GNU/Linux 5.0 debian-mips ttyS0

debian-mips login: user
Password:
Linux debian-mips 2.6.26-2-4kc-malta #1 Sat Nov 27 11:32:29 UTC 2010 mips

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
user@debian-mips:~$ scp jpe@192.168.0.1:/mnt/loop/home/sstic/secret .
secret                                        100% 1024KB   1.0MB/s   00:01
user@debian-mips:~$ scp jpe@192.168.0.1:/mnt/loop/home/sstic/ssticrypt .
ssticrypt                                     100%  304KB 304.0KB/s   00:00
user@debian-mips:~$ chmod +x ssticrypt
user@debian-mips:~$ ./ssticrypt
--> SSTICRYPT <--
usage: ./ssticrypt [-d|-e] <key> <secure container>
        -d: uncrypt
        -e: crypt
```

Le programme ssticrypt prend comme paramètres :
- un mode opératoire : -d pour déchiffrer, -e pour chiffrer ;
- une clé de chiffrement / déchiffrement ;
- un chemin vers un « container ».

Il est alors possible de réaliser un essai de chiffrement :

```
user@debian-mips:~$ cat > mysecret
hello world!
user@debian-mips:~$ ./ssticrypt -e aaa mysecret
```

```
--> SSTICRYPT <--
Error: key should be a 128-bits hexadecimal string
user@debian-mips:~$ ./ssticrypt -e aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa mysecret
--> SSTICRYPT <--
Using keys aaaaaaaaaaaaaaaa / aaaaaaaaaaaaaaaa ...
user@debian-mips:~$ ls
mysecret  mysecret.enc  secret  ssticrypt
user@debian-mips:~$ hexdump -C mysecret.enc
00000000  a1 12 83 b0 a1 2c 6d e7  fa a0 32 62 83 b4 25 a6  |.....,m...2b..%.|
00000010  e1 99 40 b4 c6 eb 12 e0  c4 98 99 50 56           |..@........PV|
0000001d
```

On imagine aisément que le but du challenge est d'arriver à passer en paramètre à `ssticrypt` la clé correcte pour déchiffrer le fichier `secret`.

Un essai de déchiffrement donne le résultat suivant :

```
user@debian-mips:~$ ./ssticrypt -d aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa secret
--> SSTICRYPT <--
Warning: MD5 mismatch for container
Using keys aaaaaaaaaaaaaaaa / aaaaaaaaaaaaaaaa ...
Traceback (most recent call last):
  File "check.py", line 50107, in <module>
AssertionError
Error: bad key1
```

1.2 Analyse rapide du programme `ssticrypt`

Des informations sur le binaire `ssticrypt` peuvent être obtenues simplement à l'aide de la commande `file` :

```
$ file ssticrypt
ssticrypt: ELF 32-bit MSB executable, MIPS, MIPS-I version 1 (SYSV), dynamically linked     \
  (uses shared libs), for GNU/Linux 2.6.8, with unknown capability 0x41000000 =  0xf676e75, \
  with unknown capability 0x10000 = 0x70401, not stripped
```

A ce stade, il faut retenir que le binaire est au format ELF sur une architecture MIPS big-endian et que les symboles de déboguage n'ont pas été supprimés. La commande `objdump` permet d'obtenir la liste des fonctions implémentées dans le binaire :

```
objdump -t ssticrypt| egrep 'g.*F \.text'
00401984 g     F .text  0000007c              swap_word
0040141c g     F .text  00000568              vicpwn_check
00402990 g     F .text  00000008              __libc_csu_fini
00400f04 g     F .text  0000005c              vicpwn_close
00400d90 g     F .text  00000174              vicpwn_init
00400f9c g     F .text  00000120              vicpwn_sendbuf
00401f7c g     F .text  00000198              check_key
00401174 g     F .text  000000c0              swap_key
00401234 g     F .text  000001e8              set_my_key
004010bc g     F .text  000000b8              load_layer
00400c40 g     F .text  00000000              __start
00401a00 g     F .text  0000049c              vicpwn_handle
```

```
00402998 g     F .text  000000b8              __libc_csu_init
00401e9c g     F .text  000000e0              extract_pyc
00400f60 g     F .text  0000003c              vicpwn_quit
00402114 g     F .text  00000874              main
```

`Metasm`[1] est capable de charger et de désassembler le binaire :

```
$ ruby -I. samples/disassemble-gui.rb --cpu Mips --fast ssticrypt
```

Le graphe d'appels du binaire est présenté sur la figure 1.

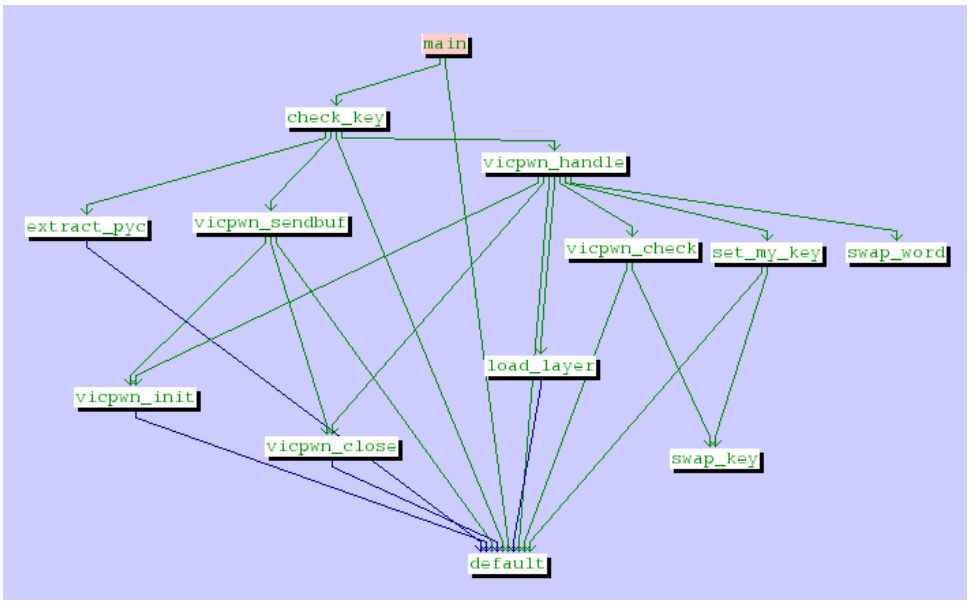

FIGURE 1 – Graphe d'appels du binaire `ssticrypt`

Il est possible d'instrumenter Metasm pour générer un fichier au format Graphviz qui représente ce graphe d'appels. Le résultat est présenté à la figure 2.

L'analyse par rétro-ingénierie du binaire permet de comprendre le fonctionnement du programme. Le fonctionnement de celui-ci peut être décrit ainsi :

- avant tout, la taille de la clé passée en argument est testée, celle-ci doit être de 32 caractères hexadécimaux (donc 16 octets) ;
- si le mode d'opération demandé est le déchiffrement, alors la vérification de la clé commence (sinon l'opération de chiffrement est réalisée puis le programme quitte) ;
- la clé est séparée en deux parties, chaque partie étant vérifiée séparément par la fonction `check_key` ;
- si la vérification de la clé réussit, alors l'opération de déchiffrement a lieu.

La fonction `check_key` accepte comme premier argument l'extrait de clé à vérifier (sur 8 octets). Le deuxième paramètre précise le type de vérification à effectuer :

- si ce paramètre est égal à 1 :
  - la fonction `extract_pyc` est appelée,
  - une chaîne de caractère est construite à l'aide de la fonction `sprintf` depuis la chaîne

1. http://code.google.com/p/metasm/

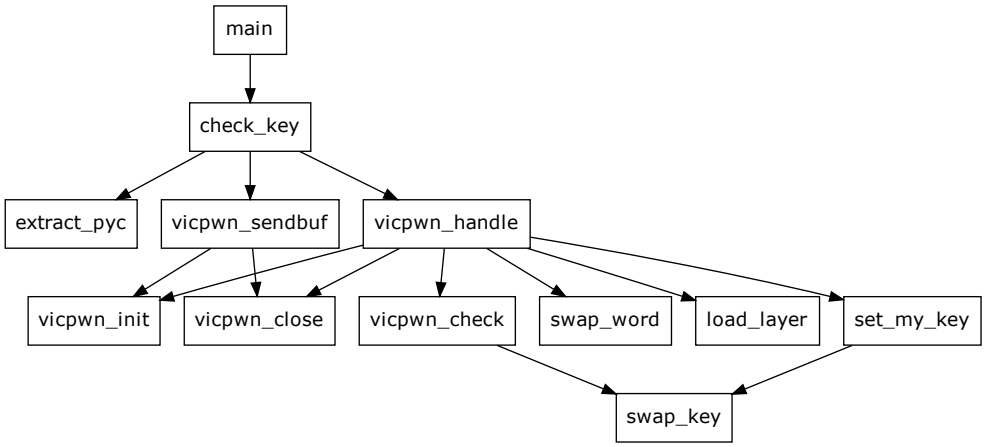

FIGURE 2 – Graphe d'appels du binaire `ssticrypt` (version graphviz)

de format `"python ./check.pyc %16s"` et la clé à vérifier,
  - un appel à la fonction `system` est réalisé sur la chaîne construite précédemment, puis le fichier `check.pyc` est supprimé à l'aide de la fonction `unlink`,
  - enfin, le code de retour de l'appel à `system` est testé : si celui-ci est égal à 0, alors la clé est valide, sinon le programme `ssticrypt` se termine ;
- sinon (paramètre différent de 1) :
  - la fonction `vicpwn_quit` est définie comme gestionnaire du signal `SIGINT`,
  - un tampon de données de taille `0x10000` est alloué avec la fonction `malloc`,
  - la fonction `vicpwn_sendbuf` est appelée deux fois avec comme paramètres un pointeur vers un tampon de données et la longueur de ces données,
  - enfin la fonction `vicpwn_handle` est appelée sur le morceau de clé à vérifier.

Le résultat de la rétro-conception de cette fonction depuis le code assembleur est présenté ci-dessous :

fonction check_key du binaire ssticrypt

```
int check_key(char *key, int id)
{
    int ret;
    char buf[255];

    if (id == 1) {
        extract_pyc();
        sprintf(buf, "python ./check.pyc %16s", key);
        ret = system(buf);
        unlink("./check.pyc");
        if (ret != 0) {
            puts("Error: bad key1");
            exit(EXIT_FAILURE);
        }
    } else {
        signal(SIGINT, vicpwn_quit);
        ram = malloc(0x10000);
        vicpwn_sendbuf(init_rom, init_rom_len);
        vicpwn_sendbuf(stage2_rom, stage2_rom_len);
```

```
        vicpwn_handle(key);
    }
}
```

L'étude de la première vérification fera l'objet du chapitre 2 tandis que la seconde partie sera examinée lors du chapitre 3.

2 DES « white-box »

2.1 Analyse du byte-code Python

D'après le paragraphe précédent, la vérification du début de la clé (les 8 premiers octets) est réalisée par un programme en Python compilé (format `pyc`). Il est donc nécessaire d'extraire le fichier `check.pyc` à partir du programme `ssticrypt`.

```
$ readelf -s ssticrypt|grep check_pyc
    88: 00413030 0x4457d OBJECT  GLOBAL DEFAULT   19 check_pyc
    99: 004575b0     4 OBJECT  GLOBAL DEFAULT   19 check_pyc_len
$ readelf -S ssticrypt|egrep '\.data'
  [19] .data             PROGBITS        00413020 003020 046dc0 00  WA  0   0 16
$ dd if=ssticrypt of=check.pyc bs=1 count=$((0x4457d)) skip=$((0x413030-0x413020+0x3020))
$ file check.pyc
check.pyc: python 2.5 byte-compiled
```

Le fichier `check.pyc` correspond donc à un programme compilé pour Python 2.5. On peut supposer que l'analyse de ce fichier permettra de comprendre la logique de vérification de la clé (et donc d'en déduire la clé attendue).

L'exécution du programme extrait permet de comprendre quels sont les arguments attendus.

```
$ python2.5 check.pyc
Usage: python check.pyc <key>
   - key: a 64 bits hexlify-ed string
Example: python check.pyc 0123456789abcdef
```

En l'occurence, le programme demande un seul argument qui est une chaîne de caractères représentant 64 bits en hexadécimal.

La page Internet [1] décrit précisément la structure d'un fichier Python compilé. On peut y apprendre qu'un tel fichier contient :

- un nombre « magique » sur 4 octets ;
- une date de modification sur 4 octets ;
- un objet de code sérialisé au format « marshal » [2].

De plus, le format du fichier est indépendant de l'architecture matérielle mais dépendant de la version de Python. Dans mon cas, j'ai du installer la version 2.5 de Python pour pouvoir continuer le challenge.

L'auteur de cet article propose également un désassembleur de byte-code utilisant des fonctionnalités natives de Python (`import dis, marshal`). Le code de ce désassembleur est présenté ci-dessous :

```
import dis, marshal, struct, sys, time, types

def show_file(fname):
    f = open(fname, "rb")
    magic = f.read(4)
    moddate = f.read(4)
    modtime = time.asctime(time.localtime(struct.unpack('<L', moddate)[0]))
    print "magic %s" % (magic.encode('hex'))
    print "moddate %s (%s)" % (moddate.encode('hex'), modtime)
    code = marshal.load(f)
    show_code(code, '->')

def show_code(code, indent=''):
    print "%scode" % indent
    indent += '   '
    print "%sargcount %d" % (indent, code.co_argcount)
    print "%snlocals %d" % (indent, code.co_nlocals)
    print "%sstacksize %d" % (indent, code.co_stacksize)
    print "%sflags %04x" % (indent, code.co_flags)
    show_hex("code", code.co_code, indent=indent)
    dis.disassemble(code)
    print "%sconsts" % indent
    for const in code.co_consts:
        if type(const) == types.CodeType:
            show_code(const, indent+'   ')
        else:
            print "   %s%r" % (indent, const)
    print "%snames %r" % (indent, code.co_names)
    print "%svarnames %r" % (indent, code.co_varnames)
    print "%sfreevars %r" % (indent, code.co_freevars)
    print "%scellvars %r" % (indent, code.co_cellvars)
    print "%sfilename %r" % (indent, code.co_filename)
    print "%sname %r" % (indent, code.co_name)
    print "%sfirstlineno %d" % (indent, code.co_firstlineno)
    show_hex("lnotab", code.co_lnotab, indent=indent)

def show_hex(label, h, indent):
    h = h.encode('hex')
    if len(h) < 60:
        print "%s%s %s" % (indent, label, h)
    else:
        print "%s%s" % (indent, label)
        for i in range(0, len(h), 60):
            print "%s   %s" % (indent, h[i:i+60])

show_file(sys.argv[1])
```

La sortie de l'exécution de ce programme sur le fichier `check.pyc` est présentée ci-dessous :

```
$ python2.5 disas.py check.pyc
magic b3f20d0a
moddate fd6f584f (Thu Mar  8 09:38:21 2012)
->code
->   argcount 0
```

```
->   nlocals 0
->   stacksize 11
->   flags 0040
->   code
->      6400006401006b00006c01005a01006c02005a020001640006402006b03
[...]
->      6a02006f0e0001652500641f00830100016e0b000165250064170083010 0
->      016e01000164030053
  3           0 LOAD_CONST               0 (-1)
              3 LOAD_CONST               1 (('floor', 'log'))
              6 IMPORT_NAME              0 (math)
              9 IMPORT_FROM              1 (floor)
             12 STORE_NAME               1 (floor)
             15 IMPORT_FROM              2 (log)
             18 STORE_NAME               2 (log)
             21 POP_TOP
[...]
```

Le fichier résultant pèse 4565 lignes et est disponible en annexe (cf. A.1.1). En parcourant ce fichier, on retrouve rapidement le point d'entrée du programme :

```
459         254 LOAD_NAME               25 (__name__)
            257 LOAD_CONST              21 ('__main__')
            260 COMPARE_OP               2 (==)
            263 JUMP_IF_FALSE          218 (to 484)
            266 POP_TOP

460         267 LOAD_NAME                9 (pickle)
            270 LOAD_ATTR               26 (loads)

50100         273 LOAD_CONST              22 ("ccopy_reg\n_reconstructor\np0\n(ccheck\nWhiteDES [...]
              276 CALL_FUNCTION            1
279           STORE_NAME              27 (WT)
```

Le test `__name__ == __main__` est caractéristique du début d'un programme Python. Si ce test est vérifié, alors la fonction `loads` du module `pickle` est appelée sur une constante et le résultat est stocké dans la variable `WT`. Ce module `pickle` permet de sérialiser et désérialiser un objet Python. La chaîne de caractère `WhiteDES` dans la constante chargée à l'aide de `pickle` laisse à penser que l'objet instancié est une implémentation de l'algorithme DES en mode « white-box ».

Ce mode d'utilisation de l'algorithme `DES` permet d'inclure au sein même de l'implémentation la clé secrète de chiffrement / déchiffrement et de rendre celle-ci difficilement extractible, même pour un attaquant qui aurait accès au code source de l'implémentation de l'algorithme. Le cas d'usage le plus courant sont les applications qui proposent des fonctionnalités de type DRM.

Pour rendre l'extraction de la clé secrète difficile, un certain nombre de transformations sont pré-calculées à partir de la clé à protéger et des `Sbox` pour pouvoir générer des tables de « look-up ». De plus, des opérations d'« encodage » sont appliquées en entrée et en sortie sur certaines primitives de DES. L'article de référence sur le sujet est *A White-Box DES Implementation for DRM Applications* [3].

Pour l'heure, l'objectif est de comprendre la suite du programme `check.pyc`. L'analyse de la suite du désassemblage permet d'identifier les manipulations effectuées sur la clé passée en

argument :

```
50106        323 LOAD_NAME               10 (Bits)
             326 LOAD_NAME                5 (a2b_hex)
             329 LOAD_NAME                6 (sys)
             332 LOAD_ATTR               29 (argv)
             335 LOAD_CONST              23 (1)
             338 BINARY_SUBSCR
             339 CALL_FUNCTION            1
             342 LOAD_CONST              27 (64)
             345 CALL_FUNCTION            2
             348 STORE_NAME              30 (K)
```

La clé est transformée en binaire par la méthode `a2b_hex` du module `Bits` puis le résultat est stocké dans la variable `K`. Un test est ensuite réalisé sur la valeur de `K` :

```
50107        351 LOAD_NAME               30 (K)
             354 LOAD_NAME               31 (range)
             357 LOAD_CONST              28 (7)
             360 LOAD_CONST              27 (64)
             363 LOAD_CONST              29 (8)
             366 CALL_FUNCTION            3
             369 BINARY_SUBSCR
             370 LOAD_CONST              30 (175)
             373 COMPARE_OP               2 (==)
             376 JUMP_IF_TRUE             7 (to 386)
             379 POP_TOP
             380 LOAD_GLOBAL             32 (AssertionError)
             383 RAISE_VARARGS            1
        >>   386 POP_TOP
```

La clé `K` est indexée par l'intervalle généré (7, 15, 23, 31, 39, 47, 55, 63), la valeur extraite est comparée à 175. Si cette condition n'est pas vérifiée, une erreur est déclenchée. Cette opération ressemble à une vérification d'une somme de contrôle sur les bits de parité de chaque octet de la clé.

La suite du programme est présentée ci-dessous :

```
50108        387 LOAD_NAME               10 (Bits)
             390 LOAD_NAME                8 (random)
             393 LOAD_ATTR               33 (getrandbits)
             396 LOAD_CONST              27 (64)
             399 CALL_FUNCTION            1
             402 LOAD_CONST              27 (64)
             405 CALL_FUNCTION            2
             408 STORE_NAME              34 (M)

50112        411 LOAD_NAME               35 (hex)
             414 LOAD_NAME               27 (WT)
             417 LOAD_ATTR               36 (_cipher)
             420 LOAD_NAME               34 (M)
             423 LOAD_CONST              23 (1)
             426 CALL_FUNCTION            2
             429 CALL_FUNCTION            1
             432 LOAD_NAME               35 (hex)
             435 LOAD_NAME               13 (enc)
```

```
438 LOAD_NAME                30 (K)
441 LOAD_NAME                34 (M)
444 CALL_FUNCTION             2
447 CALL_FUNCTION             1
450 COMPARE_OP                2 (==)
453 JUMP_IF_FALSE            14 (to 470)
```

Un bloc de 64 bits de données aléatoires est généré puis est chiffré de deux façons :
– en appelant la méthode `_cipher` de l'objet WT ;
– en appelant la fonction `enc` avec comme paramètre la clé `K` passée en ligne de commande.
Si le résultat des deux chiffrements est différent, alors le code saute à l'adresse 470 et sort du programme avec un code retour égal à 1. Sinon, le code retour vaut 0.

Au final, les opérations décrites précédemment peuvent être résumées par le code Python ci-dessous :

```
key_data = binascii.a2b_hex(sys.argv[1])
K = Bits(key_data, 64)
r = range(7, 64, 8)
t = K[r]

if t != 175:
    raise AssertionError

if hex(WT._cipher(M, 1)) == hex(enc(K, M)):
    exit(0)
else:
    exit(1)
```

La prochaine étape est d'arriver à implémenter en Python la méthode `_cipher` et la fonction `enc`.

Dans un premier temps, il est intéressant d'identifier quelles sont les fonctions du programme, ou plus généralement tous les objets de type « code » :

```
$ python2.5 disas.py check.pyc | grep "code object"
[...]
256         133 LOAD_CONST     8 (<code object enc at 0x7ff5c7ca57b0, file "check.py", line 256>)
272         142 LOAD_CONST     9 (<code object dec at 0x7ff5c7ca58a0, file "check.py", line 272>)
288         151 LOAD_CONST    10 (<code object subkey at 0x7ff5c7ca5918, file "check.py", line 288>)
297         160 LOAD_CONST    11 (<code object F at 0x7ff5c7ca5990, file "check.py", line 297>)
312         169 LOAD_CONST    12 (<code object IP at 0x7ff5c7ca5a08, file "check.py", line 312>)
324         178 LOAD_CONST    13 (<code object IPinv at 0x7ff5c7ca5a80, file "check.py", line 324>)
336         187 LOAD_CONST    14 (<code object PC1 at 0x7ff5c7ca5af8, file "check.py", line 336>)
347         196 LOAD_CONST    15 (<code object PC2 at 0x7ff5c7ca5b70, file "check.py", line 347>)
359         205 LOAD_CONST    16 (<code object E at 0x7ff5c7ca5be8, file "check.py", line 359>)
371         214 LOAD_CONST    17 (<code object P at 0x7ff5c7ca5c60, file "check.py", line 371>)
381         223 LOAD_CONST    18 (<code object S at 0x7ff5c7ca5cd8, file "check.py", line 381>)
[...]
```

Toutes ces fonctions devront donc être réimplémentées, en commençant par la fonction `enc` jusqu'aux fonctions terminales. Grâce au module `marshal`, il est possible d'instancier dynamiquement ces fonctions à partir du byte-code présent dans le fichier `check.pyc`.

```
def enc(): pass
def dec(): pass
def subkey(): pass
def F(): pass
def IP(): pass
def IPinv(): pass
def PC1(): pass
def PC2(): pass
def E(): pass
def P(): pass
def S(): pass
def tmp(): pass

f = open('check.pyc', 'rb')
code = marshal.load(f)
f.close

for const in code.co_consts:
    if type(const) == types.CodeType:
        if const.co_name == 'Bits':
            tmp.func_code = const
            Bits = type('Bits', (), tmp())
            continue

        if const.co_name == 'ECB':
            tmp.func_code = const
             ECB = type('ECB', (object, ), tmp())
             continue

        if const.co_name == 'WhiteDES':
            tmp.func_code = const
            WhiteDES = type('WhiteDES', (ECB,), tmp())
            continue

        tmpf = locals()[const.co_name]
        if tmpf:
            print "creating function %s" % (const.co_name)
            tmpf.func_code = const

WT = pickle.loads(code.co_consts[22])
```

De même, les classes `Bits`, `EBC`, `WhiteDES` et l'objet `WT` (instance de `WhiteDES`) sont instanciés à partir du byte-code.

Le code Python ci-dessous permet d'obtenir la description des méthodes de la classe `Bits`, qui représente un champ de bits indexable.

```
for name, method in inspect.getmembers(Bits, predicate=inspect.ismethod):
    print "%s: %s" % (name, method.func_doc)
```

Le résultat est présenté ci-dessous (les méthodes sans commentaires ont été retirées) :

```
__floordiv__: operator // is used for concatenation.
__getitem__: getitem defines b[i], b[i:j] and b[list] and returns a Bits instance
__hex__: byte string representation, bit0 first.
__iter__: bit iterator.
__setitem__: setitem defines
```

```
            b[i]=v with v in (0,1),
            b[i:j]=v
         and b[list]=v where
    v is iterable with range equals to that required by i:j or list,
    or v generates a Bits instance of desired length.
__str__: binary string representation, bit0 first.
bitlist: return a list of bits (bit0 first)
hd: hamming distance to another object of same length.
hw: hamming weight of the object (count of 1s).
```

A ce stade, nous avons donc un programme en Python fonctionnel, qui instancie au démarrage les fonctions, classes et méthodes nécessaires à partir des données du fichier `check.py`. J'ai donc ensuite procédé à la réimplémentation de la fonction `enc` en me basant sur le désassemblage du byte-code. La fonction réimplémentée se nomme `_enc` :

```python
def _enc(K, M):
    if M.size != 64:
        raise AssertionError
    k = PC1(K)
    blk = IP(M)
    L = blk[0:32]
    R = blk[32:64]
    for r in range(16):
        fout = F(R, k, r)
        L = L ^ fout
        L, R = R, L
    L, R = R, L
    C = Bits(0, 64)
    C[0:32] = L
    C[32:64] = R
    return IPinv(C)
```

Cette implémentation fait encore appel aux fonctions `PC1`, `IP`, `F` et `IPinv` instanciées depuis le byte-code. L'intérêt est de pouvoir valider la correction de la réimplémentation de la fonction `_enc` en comparant sa sortie (depuis une entrée de test) à celle obtenue par un appel à la fonction `enc`. Une fois cette réimplémentation validée, on peut alors s'attaquer à celle de `_PC1` par la même méthode. Au passage, on peut reconnaître que la fonction `enc` constitue la fonction principale d'un chiffrement DES, `F` étant la fonction Feistel. Le fonctionnement de l'algorithme est représenté à la figure 3.

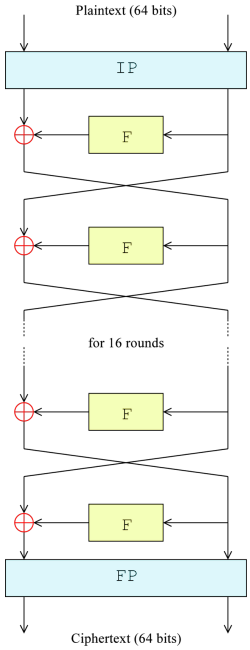

FIGURE 3 – Structure de l'algorithme DES

En suivant cette démarche, il est possible d'obtenir les fonctions `_PC1`, `_PC2`, `_IP`, `_IPinv`, `_E`, `_subkey`, `_S`, `_P`, `_F` et `_enc` qui sont les réimplémentations en Python des fonctions originales. De même, la classe `MyWhiteDES` réimplémente en Python la méthode `_cipher` :

```
class MyWhiteDES:
    def FX(self, v):
        res = Bits(0, 96)
        for b in range(96):
            t = v & self.tM2[b]
            # hw = hamming weight of the object (count of 1s)
            res[b] = t.hw() % 2
        return res

    def _cipher(self, M, d):
        if M.size != 64:
            raise AssertionError
        if d != 1:
            if d == -1:
                raise NotImplementedError
            else:
                return 0

        blk = M[self.tM1]
        for r in range(16):
            t = 0
            for n in range(12):
                nt = t + 8
                i = blk[t:nt].ival
                blk[t:nt] = self.KT[r][n][i]
                t = nt
            blk = self.FX(blk)
        return blk[self.tM3]
```

La table de « look-up » `KT` est directement copiée depuis l'objet `WT` après dé-sérialisation de celui-ci.

La méthode `check_reverse` présentée ci-dessous permet de vérifier globalement la correction de la réimplémentation en Python :

```
def check_reverse(WT):
    M = Bits(random.getrandbits(64), 64)
    K = Bits(random.getrandbits(64), 64)

    if hex(_enc(K, M)) == hex(enc(K, M)):
        print "_enc(K, M) == enc(K, M)"
    else:
        print "_enc(K, M) != enc(K, M)"
        exit(1)

    myWT = MyWhiteDES()
    myWT.KT = WT.KT
    myWT.tM1 = WT.tM1
    myWT.tM2 = WT.tM2
    myWT.tM3 = WT.tM3

    if hex(myWT._cipher(M,1)) == hex(WT._cipher(M, 1)):
        print "myWT._cipher(M,1) == WT._cipher(M, 1)"
    else:
        print "myWT._cipher(M,1) != WT._cipher(M, 1)"
        exit(1)
```

Il est alors possible de tester le résultat de la décompilation du byte-code de `check.pyc` :

```
$ ./reverse.py 0123456789abcdef
magic b3f20d0a
moddate fd6f584f (Thu Mar  8 09:38:21 2012)
creating function enc
creating function dec
creating function subkey
creating function F
creating function IP
creating function IPinv
creating function PC1
creating function PC2
creating function E
creating function P
creating function S
_enc(K, M) == enc(K, M)
myWT._cipher(M,1) == WT._cipher(M, 1)
Traceback (most recent call last):
  File "./reverse.py", line 319, in <module>
    test_key(sys.argv[1])
  File "./reverse.py", line 300, in test_key
    raise AssertionError
AssertionError
```

Une erreur est déclenchée car le calcul de la somme de contrôle échoue. Cependant, les implémentations de la fonction `_enc` et de la méthode `_cipher` de la classe `MyWhiteDES` semblent correctes.

Le résultat final `reverse.py` est disponible en annexe (cf. A.1.1).

2.2 Attaque par cryptanalyse et récupération de la clé

L'objectif à cet stade du challenge est d'arriver à extraire la clé secrète incorporée à l'implémentation de l'algorithme DES en mode « white-box ».

Les auteurs de l'article *A White-Box DES Implementation for DRM Applications* [3] sont conscients des limites de leur méthode de protection de la clé secrète et propose deux attaques par cryptanalyse nommées :

- *Jacob Attack on the Naked Variant* ;
- *Statistical Bucketing Attack on Naked Variant.*

La variante dite « naked » de l'implémentation « white-box » est fonctionnellement identique à l'algorithme DES original. Comme contre mesure aux deux attaques précédemment citées, les auteurs proposent une variante dite « non-standard » qui réalise des opérations de transformation des entrées et des sorties. On parle alors d'« external encoding ».

La présentation [5] explique de manière succincte les attaques possibles sur la version « naked » et la version « non-standard ». Dans notre cas, l'objectif est d'identifier quelle version est implémentée par l'objet Python `WT` afin de choisir l'attaque susceptible d'aboutir.

La fonction `_enc` est une implémentation DES standard, sans modification des données en entrée ou en sortie. De plus, les S-Box définies dans la fonction `_S` sont conformes aux S-Box standard (cf. [6]). Le résultat du chiffrement par la méthode `_cipher` de l'objet `WT` étant comparé au résultat de la fonction `enc` (ou `_enc` qui la version réimplémentée en Python), on peut supposer que la variante du challenge est standard (ou « naked »).

L'attaque choisie est alors décrite au chapitre 3 de l'article *Cryptanalysis of white box DES implementations* [4].

L'objectif de l'attaque est bien évidemment d'extraire la clé secrète de 56 bits masquée dans l'implémentation de la « white-box ». Plus précisément, l'attaque permet d'obtenir la valeur de la sous clé du premier tour qui est le résultat de la fonction de rotation de clé (`_subkey` du fichier `reverse.py`) appliquée à la clé principale. 48 bits de la clé principale se retrouvent dans la sous clé. Cette dernière est alors utilisée dans la fonction de Feistel (cf figure 4) pour chiffrer la partie droite du message.

Dans cette fonction, la sous clé est divisée en 8 parties de 6 bits, chacune n'intervenant conjointement qu'avec une seule S-Box.

L'attaque permet de retrouver, pour chaque S-Box, les 6 bits de la sous clé qui interviennent dans son évaluation. Elle doit donc être répétée 8 fois (une fois par S-Box) afin de retrouver les 48 bits de la sous clé.

Les détails de l'attaque sont relativement complexes et il est recommandé de se référer à l'article [4] pour en comprendre le fonctionnement. Cependant, le principe général peut être

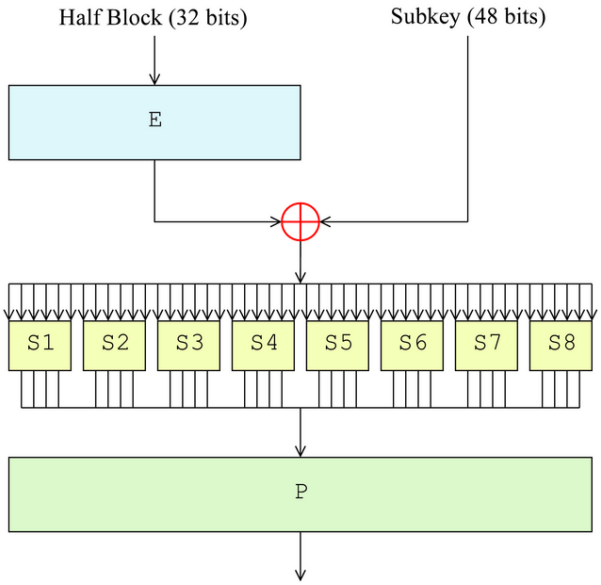

FIGURE 4 – Fonction de Feistel de DES

énoncé ainsi :

- on cible la S-Box qui correspond aux 6 bits de la sous clé que l'on cherche à obtenir ;
- on génère les 2^6 sous clés candidates pour cette S-Box ;
- des paires de messages sont construites sur la base de l'algorithme DES standard pour la sous clé candidate testée, de manière à minimiser les différences entre leurs chiffrés intermédiaires ;
- chaque paire de messages est alors évaluée au travers du DES « white-box ». Si les différences entre leurs chiffrés intermédiaires sont trop nombreuses, alors la sous clé candidate est éliminée.

Après itération du processus, on obtient pour chaque S-Box plusieurs variations potentiellement valables pour la sous clé du premier tour. L'attaque a été implémentée dans le fichier `reverse.py` A.1.1 (fonction `find_candidates`) et permet d'obtenir les candidats suivants pour chaque partie de la sous clé :

```
[   Set([33, 45]),
    Set([10, 6]),
    Set([35, 47]),
    Set([43, 39]),
    Set([43, 39]),
    Set([34, 46]),
    Set([9, 5]),
    Set([57, 53])]
```

Ainsi, les entiers 33 et 45 correspondent aux valeurs possibles pour les 6 premiers bits de la sous clé recherchée.

Chaque combinaison possible de ces candidats correspond à une sous clé candidate de 48 bits, l'une d'entre elles étant la sous clé correcte. Il y a donc 2^8 sous clés candidates.

Pour chaque sous clé candidate, en inversant la fonction de rotation, il ne reste plus que 8

bits de la clé initiale à retrouver. Au total, cela correspond à une recherche exhaustive réduite à 2^{16} possibilités. La clé correcte est retrouvée en comparant les chiffrés obtenus par les deux implémentations de DES qui seront nécessairement identiques pour un clair donné.

Le code Python de la fonction réalisant cette recherche exhaustive est présenté ci-dessous :

```
def bf(WT):
    sk = Bits(0, 48)
    M = Bits(random.getrandbits(64), 64)
    target = WT._cipher(M, 1)
    candidates = list(product([33,45], [10, 6], [35, 47], [43, 39], [43, 39],
     [34, 46], [9, 5], [57, 53]))
    for c in candidates:
        r = 0
    for x in c:
        nr = r + 6
        sk[r:nr] = x
        r += 6
    rsk = reverse_subkey(sk)
    for rest in range(256):
        key = Bits(0, 56)
        b = Bits(0, 8)
        b[0:8] = rest
        idx = 0
        for pos, item in enumerate(rsk):
        if item == None:
            key[pos] = b[idx].ival
            idx += 1
        else:
            key[pos] = item
        if target == _enc_no_pc1(key, M):
            print "%s -> %s" % (b.__str__(), key.__str__())
            print "key found !"
            exit(0)
```

La fonction `reverse_subkey` implémente la fonction inverse de la fonction de rotation. Elle prend donc en entrée une valeur sur 48 bits (la sous clé du tour) et retourne la clé principale sur 56 bits. En réalité, 8 bits de la clé principale sont alors inconnus, la valeur `None` est affectée aux positions qui correspondent à ces bits. Ces bits inconnus sont retrouvés par recherche exhaustive.

La recherche aboutit après environ une heure de calculs :

```
11000010 -> 10101101110110111011100110000101100010011101111010011001
key found !
```

Il ne reste plus alors qu'à inverser la fonction `PC1` pour obtenir les 64 bits attendus en entrée. De plus, le bit de poids faible de chaque octet de la clé doit être modifié pour vérifier le calcul de la somme de contrôle.

Le code ci-dessous permet de retrouver la clé finale : `fd4185ff66a94afd`.

```
    found_key = Bits(0, 56)
    finalkey = Bits(0, 64)
    found_key_data = [1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0,
        0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1]
```

```
    for pos, item in enumerate(found_key_data):
        found_key[pos] = item

    inv_found_key = _PC1inv(found_key)
    checksum = Bits(0, 8)
    checksum[0:8] = 175

    i = 0
    for pos, item in enumerate(inv_found_key):
    if item == None:
        finalkey[pos] = checksum[i].ival
        i += 1
    else:
        finalkey[pos] = item

    print "finalkey : %s" % (binascii.b2a_hex(finalkey.__hex__()))
```

```
$ python2.5 check.pyc fd4185ff66a94afd
$ echo $?
0
```

3 Analyse de la webcam

3.1 Analyse détaillée du programme `ssticrypt`

Comme précisé au chapitre 1.2, la deuxième partie de la vérification de la clé commence par deux appels à la fonction `vicpwn_sendbuf`. Cette dernière est appelée sur deux blocs de données qui sont référencés dans la table de symboles du programme :

```
$ objdump -t sticrypt|grep rom
004575f8 g     O .data  00000004              init_rom_len
004581a4 g     O .data  00000004              stage2_rom_len
0045772c g     O .data  00000a76              stage2_rom
004575b4 g     O .data  00000044              init_rom
```

D'après le paragraphe 1.2, le premier appel à `vicpwn_sendbuf` a pour paramètres `init_rom` et `init_rom_len`, tandis que le second appel a pour paramètres `stage2_rom` et `stage2_rom_len`.

Le résultat de la rétro-conception de la fonction `vicpwn_sendbuf` depuis le code assembleur MIPS est présenté ci-dessous :

fonction vicpwn_sendbuf du binaire ssticrypt

```
void vicpwn_sendbuf(char *data, int len) {
    vicpwn_init(0x9d);
    if (devCam == NULL) {
        devCam = vicpwn_init(0x9d);
    }
    if (devCam == NULL) {
        exit(EXIT_FAILURE);
    }
    usb_control_msg(devCam, USB_TYPE_VENDOR, 0xff, 0, 0, data, len, 1000);
    vicpwn_close();
```

```
}
```

La variable `devCam` est une variable globale au programme. Il s'agit d'un pointeur vers une structure `usb_dev_handle`.

La fonction `vicpwn_sendbuf` commence alors par réaliser un appel à la fonction `vicpwn_init`. Celle-ci effectue une énumération des bus et des périphériques (à l'aide des fonctions `usb_find_busses`[2] et `usb_find_devices`[3]). Spécifiquement, la fonction `vicpwn_init` recherche un périphérique ayant comme identifiant de vendeur `0x04c1` et comme identifiant de produit `0x9d`. Si un tel périphérique est présent, alors il est ouvert avec un appel à la fonction `usb_open` et un pointeur vers une structure `usb_dev_handle` est retourné. Dans le cas contraire, un pointeur NULL est retourné.

Le résultat de la rétro-conception de la fonction `vicpwn_init` depuis le code assembleur est présenté ci-dessous :

fonction vicpwn_init du binaire ssticrypt

```
usb_dev_handle *vicpwn_init(int prod_id) {
    struct usb_bus *buses;
    struct usb_device *dev;
    usb_init();
    usb_find_busses();
    usb_find_devices();
    buses = usb_get_busses();
    devCam = NULL;
    while (buses != NULL) {
        dev = buses->devices;
        while (dev != NULL) {
            if ((dev->descriptor.idVendor == 0x4c1) &&
                (dev->descriptor.idProduct == prod_id)) {
                devCam = usb_open(dev);
            }
            dev = dev->next;
        }
        buses = buses->next;
    }
    return devCam;
}
```

Après l'appel à `vicpwn_init`, la fonction `vicpwn_sendbuf` teste la valeur de retour et termine le programme si celle-ci est égale au pointeur NULL. Sinon, le périphérique recherché a été correctement identifié et ouvert. Dans ce cas, la fonction `usb_control_msg`[4] est appelée avec les paramètres suivants :

- `dev` : le pointeur retourné par `vicpwn_init` ;
- `requesttype` : 64 (`USB_TYPE_VENDOR`) ;
- `request` : `0xff` ;
- `value` : 0 ;
- `index` : 0 ;
- `bytes` : le premier paramètre de `vicpwn_sendbuf` ;
- `size` : le second paramètre de `vicpwn_sendbuf` ;

2. http://libusb.sourceforge.net/doc/function.usbfindbusses.html
3. http://libusb.sourceforge.net/doc/function.usbfinddevices.html
4. http://libusb.sourceforge.net/doc/function.usbcontrolmsg.html

– `timeout` : 1000.

Pour comprendre la signification de ces paramètres, il faut tout d'abord arriver à identifier quel est le type de périphérique recherché. La commande ci-dessous permet d'obtenir ce résultat :

```
$ grep -A 10 04c1 /usr/share/misc/usb.ids | grep 9d
    009d  HomeConnect Webcam [vicam]
```

La matériel en question est une webcam vendue par la société 3com. Une recherche sur Google permet de trouver la page du projet visant à fournir un pilote pour Linux : http://homeconnectusb.sourceforge.net/. La dernière mise à jour date d'octobre 2002. On peut y apprendre que la société `Vista Imaging` est responsable de la conception du matériel embarqué dans cette webcam. A ce stade, un des objectifs est d'obtenir les spécifications précises de la webcam pour arriver à interpréter les données envoyées par les appels à la fonction `usb_control_msg`. Cela fera l'objet du paragraphe suivant 3.2.

Pour l'instant, l'objectif est d'analyser la fonction `vicpwn_handle` qui est appelée par la fonction `check_key` après avoir envoyé à la webcam les deux blocs de données `init_rom` et `stage2_rom`.

Le résultat de la rétro-conception de cette fonction est présenté ci-dessous.

fonction vicpwn_handle du binaire ssticrypt

```
int vicpwn_handle(char *key) {
    char buf[20];
    uint8_t *layer;
    int i, ret;
    uint16_t addr_src, addr_dst;

    vicpwn_init(0x9d);
    if (devCam == NULL) {
        puts("No Dev found.");
        exit(EXIT_FAILURE);
    }

    addr_src = 0;
    layer = all_layers[0];

    for (i=0; i < 3; i++) {
        load_layer(layer, i, 0);
        set_my_key(key, i, 0xa000);

        while ( *(uint16_t *)(ram + 0x8000) == 0  || addr_src != 0 ) {
            addr_src &= 0xfff0;
            if (addr_src != 0xfff0) {
                memcpy(buf, ram + addr_src, 16);
                usb_control_msg(USB_TYPE_VENDOR | USB_ENDPOINT_OUT,
                        0x51, addr_src, 1, buf, 16, 1000);
            }

            ret = usb_control_msg(USB_TYPE_VENDOR | USB_ENDPOINT_IN,
                    0x56, 0, 0, buf, 20, 1000);

            addr_dst = (buf[ret-4] & 0xff) | ( (buf[ret-4+1] & 0xff) << 8);
            addr_src = (buf[ret-2] & 0xff) | ( (buf[ret-2+1] & 0xff) << 8);
```

```
            if ((addr_dst & 0xfff0) == 0xfff0)
                continue;

            if ((addr_dst & 1) == 0)
                continue;

            addr_dst &= 0xfff0;
            memcpy(ram + addr_dst, buf, 16);
        }
        layer = vicpwn_check(i, 0xa000, key);
        if (!layer)
            return -1;

        *(uint16_t *)(ram + 0x8000) = 0;
    }

    usb_release_interface(devCam, 0);
    vicpwn_close(devCam);
    return 0;
}
```

Cette fonction met en place un dialogue entre le programme `ssticrypt` et la webcam par l'intermédiaire d'appels à la fonction `usb_control_msg`. La clé passée en argument est vérifiée par trois niveaux appelés « layer » par la suite. Pour chaque niveau (ou couche), le principe de vérification est le suivant :

- les données du niveau en cours sont chargées en mémoire avec un appel à la fonction `load_layer` ;
- la clé à tester est chargée en mémoire à l'adresse 0xa000 avec un appel à `set_my_key` ;
- le dialogue entre le programme `ssticrypt` et la webcam s'établit. Ce dialogue prend fin quand le mot à l'adresse `0x8000` est différent de 0 et que la variable `off_src` est égale à 0 ;
- à la sortie du dialogue, la clé est vérifiée par la fonction `vicpwn_check`. Si la vérification réussit, alors la variable `layer` est mise à jour pour pointer vers les données du prochain niveau, sinon la fonction `vicpwn_handle` retourne un code d'erreur (-1) ;
- le mot à l'adresse `0x8000` est remis à 0 et la vérification du prochain niveau commence.

Au niveau de chaque étape du dialogue, deux phases sont à distinguer :

- une phase d'envoi de données (`ssticrypt` vers webcam) : 16 octets des données pointées par la variable `addr_src` sont copiés dans un tampon de données temporaire puis sont envoyés à la webcam via un appel à `usb_control_msg` ;
- une phase de réception de données (webcam vers `ssticrypt`) : après un appel à `usb_control_msg`, 20 octets de données sont reçus depuis la webcam, les quatre derniers sont utilisés pour mettre à jour les valeurs des variables `addr_dst` et `addr_src`. Si la variable `addr_dst` vérifie certaines conditions, alors les 16 premiers octets reçus sont écrits en mémoire à l'emplacement spécifié par `addr_dst`.

Trois fonctions restent à analyser pour comprendre intégralement (du moins du point de vue du binaire `ssticrypt`) le processus de vérification. Ces fonctions sont : `load_layer`, `set_my_key` et `vicpwn_check`.

La fonction `load_layer` réalise une simple copie mémoire du niveau spécifié à l'adresse indiquée :

fonction load_layer de ssticrypt
```
void load_layer(void *layer, int nlayer, int offset) {
    memcpy(ram + offset, layer, layers_len[nlayer]);
}
```

La fonction `set_my_key` copie en mémoire les données de la clé qui doivent être testées par le niveau en cours. En réalité, l'intégralité de la clé n'est pas testée à chaque niveau, seule une partie fait l'objet d'une validation :

- niveau 1 : `key[0..3]` ;
- niveau 2 : `key[2..5]` ;
- niveau 3 : `key[4..7]`.

Le code de la fonction `set_my_key` est présenté ci-dessous.

fonction set_my_key de ssticrypt
```
void set_my_key(char *key, int nlayer, int offset) {
    uint8_t buf[20];
    uint32_t ret;

    strcpy((char *) buf, key);

    /* 8, 12, 16 */
    buf[ 4 * (nlayer + 2) ] = 0;
    ret = htonl(strtoul( (char *) buf + 4 * nlayer , 0, 16));

    swap_key(&ret);
    memcpy(ram + offset, &ret, 4);
    if (nlayer == 1) {
        memcpy(ram + offset + 16, blob, 0x100);
    } else if (nlayer == 2) {
        memcpy(ram + offset + 16, blah, 0x20);
    }
}
```

On notera également que deux blocs de données sont copiés en mémoire pour les niveaux 2 et 3.

Pour finir, la fonction `vicpwn_check` effectue la validation de la clé après chaque niveau :

fonction vicpwn_check de ssticrypt
```
void *vicpwn_check(int nlayer, int addr, char *key) {
    int next_layer_id = nlayer + 1;
    int next_layer_len = layers_len[next_layer_id];
    uint8_t *orig_next_layer = all_layers[next_layer_id];
    uint32_t l, dwmem, dwkey;
    uint8_t i, j, k;
    uint8_t tmpbuf[16], S[256];

    uint8_t *next_layer = layers_buf[next_layer_id];
    uint32_t *next_layer32, *orig_next_layer32;

    memcpy(&dwmem, ram + addr, 4);
    swap_key(&dwmem);
```

```
    strncpy((char *) tmpbuf, key, 16);
    tmpbuf[8] = 0;

    dwkey = htonl(strtoul((char *) tmpbuf, 0, 16));

    switch(nlayer) {
        case 0:
            next_layer32 = (uint32_t *) next_layer;
            orig_next_layer32 = (uint32_t *) orig_next_layer;

            next_layer32[0] = orig_next_layer32[0] ^ dwmem;
            for (l=1; l < (next_layer_len/4) ; l++)
                next_layer32[l] = orig_next_layer32[l] ^ next_layer32[l-1];

            if (dwmem != dwkey) {
                fprintf(stderr, "[+] validation passed for layer 0, mem = %x, key = %x\n",
                        dwmem, dwkey);
                return next_layer;
            } else {
                fprintf(stderr, "[+] bad key for layer 0\n");
                exit(EXIT_FAILURE);
            }
        case 1:
            memcpy(S, ram + addr + 16, 256);
            if (S[254] == 0xff && S[255] == 0xff) {
                fprintf(stderr, "[+] bad key for layer 1\n");
                exit(EXIT_FAILURE);
            } else {
                fprintf(stderr, "[+] validation passed for layer 1, mem = %x, key = %x\n",
                        dwmem, dwkey);
            }
            i = 0; j = 0;
            /* RC4 */
            for (l = 0; l < next_layer_len; l++) {
                i = (i + 1) % 256;
                j = (j + S[i]) % 256;
                k = S[i];
                S[i] = S[j];
                S[j] = k;
                next_layer[l] = orig_next_layer[l] ^ S[ (S[i] + S[j]) % 256 ];
            }

            return next_layer;
        case 2:
            if (!strncmp((char *) ram + addr + 16, "V29vdCAhISBTbWVsbHMgZ29vZCA6KQ==", 0x20)) {
                fprintf(stderr, "[+] validation passed for layer 2, mem = %x, key = %x\n",
                        dwmem, dwkey);
                exit(EXIT_SUCCESS);
            } else {
                fprintf(stderr, "[+] bad key for layer 2\n");
                exit(EXIT_FAILURE);
            }
        default:
            return next_layer;
    }
}
```

On remarque que la méthode de validation de la clé est différente selon chaque niveau. On constatera également que les données d'un niveau dépendent directement de l'évaluation du

niveau précédent (sauf dans le cas du premier niveau). Par exemple, les données du niveau 3 sont déchiffrées à l'aide de l'algorithme RC4 à partir de la table d'état initialisée par le niveau 2.

Note : par rapport au code initial en assembleur, certaines simplifications ont été faites : des appels aux fonctions `swap_word` ont été supprimées car le résultat de la décompilation a été testé sur un système possédant une endianess identique à la webcam.

3.2 Identification de l'architecture matérielle

Une première piste est de regarder le code source du module Linux pour cette webcam. Ce dernier peut être consulté à l'adresse http://lxr.linux.no/linux+v2.6.26/drivers/media/video/usbvideo/vicam.c.

On peut constater la présence de cinq blocs de données dont la signification n'a pas l'air évidente, même pour les développeurs du module :

```
/* Not sure what all the bytes in these char
 * arrays do, but they're necessary to make
 * the camera work.
 */
```

La fonction `initialize_camera` présentée ci-dessous envoie ces cinqs blocs de données à la webcam à l'aide d'appels à la fonction `send_control_msg`.

```
static int
initialize_camera(struct vicam_camera *cam)
{
    const struct {
        u8 *data;
        u32 size;
    } firmware[] = {
        { .data = setup1, .size = sizeof(setup1) },
        { .data = setup2, .size = sizeof(setup2) },
        { .data = setup3, .size = sizeof(setup3) },
        { .data = setup4, .size = sizeof(setup4) },
        { .data = setup5, .size = sizeof(setup5) },
        { .data = setup3, .size = sizeof(setup3) },
        { .data = NULL, .size = 0 }
    };

    int err, i;

    for (i = 0, err = 0; firmware[i].data && !err; i++) {
        memcpy(cam->cntrlbuf, firmware[i].data, firmware[i].size);

        err = send_control_msg(cam, 0xff, 0, 0,
                       cam->cntrlbuf, firmware[i].size);
    }

    return err;
}
```

Les paramètres de la fonction `send_control_msg` sont similaires à ceux passés à la fonction `usb_control_msg` par la fonction `vicpwn_sendbuf`. On peut donc supposer que le programme `ssticrypt` chercher à initialiser (reprogrammer ?) la webcam avec les données contenues dans `init_rom` et `stage2_rom`.

Des recherches plus poussées sur Google (sur les mots clés `vicam imaging` et `processor`) permettent d'obtenir un document de spécification : http://www.vistaimaging.com/ViCAM-III%20Data%20Sheet.pdf. La lecture de ce document nous apprend que le processeur embarqué est un processeur RISC 16 bits. Cependant, le modèle précis du processeur reste inconnu.

Une piste alternative est de regarder directement la structure des données pour essayer d'y identifier des schémas répétitifs et, pourquoi pas, du code dans une architecture connue. Pour extraire les données de `init_rom` et `stage2_rom` (connaissant leurs adresses virtuelles dans le processus), il faut alors obtenir avec `objdump` l'adresse de la section `.data` (`0x413020`) ainsi que le décalage de cette section par rapport au début du fichier (`0x3020`).

```
$ objdump -t ssticrypt|grep init_rom
004575f8 g     O .data  00000004              init_rom_len
004575b4 g     O .data  00000044              init_rom
$ objdump -h ssticrypt|grep '\.data'
 18 .data         00046dc0  00413020  00413020  00003020  2**4
$ dd if=ssticrypt of=init_rom bs=1 count=$((0x44)) skip=$((0x4575b4-0x413020+0x3020))
$ md5sum init_rom
61b2ac364ee18f2ff97661d15e946565  init_rom
```

Le bloc de données `stage2_rom` peut être extrait de la même manière :

```
$ objdump -t ssticrypt |grep stage2_rom
004581a4 g     O .data  00000004              stage2_rom_len
0045772c g     O .data  00000a76              stage2_rom
$ dd if=ssticrypt of=stage2_rom bs=1 count=$((0xa76)) skip=$((0x45772c-0x413020+0x3020))
$ md5sum stage2_rom
943eadad8613ce27ca17bb58902f9984  stage2_rom
```

Une fois extrait le bloc de données `init_rom`, il est intéressant d'en examiner le contenu :

```
$ hexdump -C init_rom
00000000  b6 c3 31 00 02 64 e7 07  00 00 08 c0 e7 07 00 00  |..1..d..........|
00000010  3e c0 e7 67 fd ff 0e c0  e7 09 de 00 8e 00 c0 09  |>..g............|
00000020  40 03 c0 17 44 03 4b af  c0 07 44 03 4b af c0 07  |@...D.K...D.K...|
00000030  00 00 4b af 97 cf b6 c3  03 00 03 64 2a 00 b6 c3  |..K........d*...|
00000040  01 00 06 64                                       |...d|
00000044
```

Un motif revient régulièrement : les octets `b6 c3` suivis de ce qui semble être une indication de taille (`0x31`, `0x3`, `0x1`) et enfin un code sur un octet (`0x2`, `0x3`, `0x6`) suivi de l'octet `0x64`. Si les octets `31 00` sont effectivement une indication de taille, on peut alors supposer que le reste des données est codé en little-endian.

La recherche sur Google des mots clés « risc processor 16 bits 0xc3b6 » nous amène vers des documents de spécification du contrôleur USB SL11R, produit de la société Cypress Semiconductor. Le manuel du BIOS [8] est particulièrement intéressant. On peut y apprendre, page

21, que la structure identifiée précédemment correspond à une signature utilisée pour interagir avec le BIOS du système.

Cette structure est décrite au tableau 1.

Type de données	**Champ**	**Signification**
mot de 16 bits	`0xc3b6`	début de la signature
mot de 16 bits	longueur	longueur des données à suivre (sans prendre en compte l'opcode)
octet	opcode	type d'action à effectuer au niveau du BIOS
octets	données	1 à n octets de données (selon la valeur de l'opcode)

TABLE 1 – Structure des signatures

Une étude plus approfondie du document de spécification du BIOS permet de trouver l'information suivante page 48 : « The SUSBx_LOADER_INT will be called if bmRequest = 0xFF ». La valeur `0xff` est identique au paramètre passé à la fonction `usb_control_msg` par la fonction `vicpwn_sendbuf` (cf. 3.1). Cette interruption permet d'activer le mode de déboguage du processeur : « These interrupts vectors are designed to support the debugger and should not be modified by the user. BIOS uses the USB idle task to monitor the Vendor Command Class packet with the bRequest value equal to 0xff (i.e. debugger command). When this command is detected, it will call these interrupts. »

Une seconde recherche avec les mots clés « vicam cypress » nous amène vers un message [5] sur un forum de développement du pilote de la webcam : « I have been able to download a copy of the cypress sl11r manual. This will allow the decode of the firmware. I have looked at it but it give me a headache. Email me if you want me to send you a copy... ». Ce résultat permet de confirmer l'hypothèse que le processeur embarqué sur la webcam serait un processeur Cypress RISC 16 Bits.

Une dernière recherche avec les mots clés « Cypress RISC 16 Bits » permet d'obtenir un nouveau document de spécification intitulé « CY16 USB Host/Slave Controller/16-Bit RISC Processor Programmers Guide » [7]. Ce document détaille le jeu d'instruction supporté par le processeur.

Le décodage des structures de signature dans les blocs de données `init_rom` et `stage2_rom` est présenté au tableau 2.

Bloc	**Longueur**	**Opcode**	**Signification**
`init_rom`	49	2	Write Interrupt Service Routine (int 100)
`init_rom`	3	3	Fix-up (relocate) ISR Code (int 100)
`init_rom`	1	6	Call Interrupt (int 100)
`stage2_rom`	2217	2	Write Interrupt Service Routine (int 100)
`stage2_rom`	445	3	Fix-up (relocate) ISR Code (int 100)
`stage2_rom`	1	6	Call Interrupt (int 100)

TABLE 2 – Signatures décodées

5. http://sourceforge.net/projects/homeconnectusb/forums/forum/69637/topic/1208527

Le décodage des signatures permet de mieux comprendre le processus de l'initialisation de la webcam. Pour chaque bloc de données, les étapes suivantes sont réalisées :

- chargement en mémoire d'une routine de traitement d'interruption (pour l'interruption 100 dans notre cas) ;
- traitement des relocations sur la routine précédemment chargée ;
- déclenchement de l'interruption.

Le script Ruby présenté ci-dessous permet d'extraire les données de chaque signature :

```ruby
#!/usr/bin/env ruby

input = ARGV.shift

basename = File.basename(input)
dirname = File.dirname(input)

count = 0
File.open(input, "rb") do |fi|
  while s = fi.read(2)
    magic = s.unpack('v').first
    raise unless magic == 0xc3b6
    size = fi.read(2).unpack('v').first
    opcode = fi.read(1).unpack('c').first
    interrupt = fi.read(1).unpack('c').first
    data = fi.read(size-1)
    puts "scan record found: #{opcode}, #{size}"
    output_basename = "#{basename}_scan_record_#{count}"
    File.open(File.join(dirname, output_basename), "wb") do |fo|
      fo.write(data)
    end
    count += 1
  end
end
```

L'exécution de ce script sur le bloc `stage2_rom` extrait correctement les données des trois signatures :

```
$ ruby scan_signature.rb stage2_rom
scan record found: 2, 2217
scan record found: 3, 445
scan record found: 6, 1
$ ls -al stage2_rom*
-rw-rw-r-- 1 jpe jpe 2678 mars  27 13:58 stage2_rom
-rw-rw-r-- 1 jpe jpe 2216 mai    8 23:33 stage2_rom_scan_record_0
-rw-rw-r-- 1 jpe jpe  444 mai    8 23:33 stage2_rom_scan_record_1
-rw-rw-r-- 1 jpe jpe    0 mai    8 23:33 stage2_rom_scan_record_2
```

Maintenant que le jeu d'instructions du processeur est connu, il est possible de de désassembler les routines d'interruption chargées en mémoire afin de comprendre la logique programmée au niveau de la webcam.

3.3 Désassemblage de la ROM

Ma première approche fut de commencer à développer un désassembleur en Ruby en me basant sur les spécifications du processeur CY16 [7]. Cependant, en surveillant les changements au niveau du code source de Metasm, je découvris un message daté du 28 mars qui attira mon attention : « add cy16 cpu (not working yet) ». Après prise en compte de correctifs par Yoann Guillot, le support du processeur CY16 se stabilise le 01/04/2012. Je décide donc d'utiliser Metasm pour la suite du challenge.

3.3.1 Présentation de l'architecture du processeur CY16

Le document [7] détaille les spécifications du processeur CY16. Les caractéristiques présentées ci-dessous sont importantes pour le reste du challenge :

- le processeur CY16 est un processeur RISC 16 bits avec un adressage par octets ;
- il utilise un espace mémoire commun pour les données et le code ;
- il possède deux jeux de 16 registres, les registres courants sont sélectionnés selon la valeur du registre `REGBANK` ;
- un registre `FLAGS` représente l'état des drapeaux de condition à un instant donné ;
- les registres `r0` à `r7` sont des registres généralistes ;
- les registres `r8` à `r14` peuvent être utilisés comme des registres généralistes mais servent principalement à stocker des adresses ;
- le registre `r15` sert de pointeur de pile ;
- six modes d'adressage différents peuvent être utilisés.

3.3.2 Identification des points d'entrée de la ROM

Le désassemblage sur les données de la première signature du bloc de données `init_rom` fonctionne correctement.

```
$ ruby -I. samples/disassemble.rb --cpu CY16 init_rom_scan_record_0
entrypoint_0:
// function binding: sp -> sp+2
// function ends at 2eh
    mov word ptr [0c008h], 0                    ; @0  e707000008c0  w2:0c008h
    mov word ptr [0c03eh], 0                    ; @6  e70700003ec0  w2:0c03eh
    and word ptr [0c00eh], 0fffdh               ; @0ch  e767fdff0ec0  r2:0c00eh w2:0c00eh
    mov word ptr [8eh], word ptr [0deh]         ; @12h  e709de008e00  r2:0deh w2:8eh
    mov r0, word ptr [340h]                     ; @18h  c0094003  r2:340h
    add r0, 344h                                ; @1ch  c0174403
    int 4bh                                     ; @20h  4baf
    mov r0, 344h                                ; @22h  c0074403
    int 4bh                                     ; @26h  4baf
    mov r0, 0                                   ; @28h  c0070000
    int 4bh                                     ; @2ch  4baf
    ret                                         ; @2eh  97cf  endsub entrypoint_0
```

Le code initialise certaines adresses en mémoire puis appelle l'interruption `0x4b` qui est utilisée pour libérer des zones mémoires (d'après le manuel du BIOS).

Le désassemblage sur les données de la première signature du bloc de données `stage2_rom` donne le résultat suivant :

```
$ ruby -I. samples/disassemble.rb --fast --cpu CY16 stage2_rom_scan_record_0

    mov word ptr [0aah], 76h                    ; @0  e7077600aa00
    mov word ptr [0b4h], 0f756h                 ; @6  e70756f7b400
    mov r8, 11ah                                ; @0ch  c8071a01
    mov word ptr [r8++], 4000h                  ; @10h  e0070040
    mov word ptr [r8++], 0                      ; @14h  e0070000
    mov word ptr [r8++], 0                      ; @18h  e0070000
    ret                                         ; @1ch  97cf
db 54 dup(0)                                    ; @1eh
```

Le désassemblage s'arrête assez rapidement, le reste des données n'étant pas décodé. Le manuel du BIOS [8] précise à la page 19 que l'adresse `0xaa` contient l'adresse de la routine d'interruption 85 `SUSB1_VENDOR_INT`. La webcam va donc exécuter le code à l'adresse 0x76 pour traiter la réception d'un message USB de type `VENDOR`. Cette adresse `0x76` constitue donc un point d'entrée supplémentaire dans le code, ce point d'entrée devant être passé en paramètre à Metasm lors du désassemblage.

```
$ ruby -I. samples/disassemble.rb --fast --cpu CY16 stage2_rom_scan_record_0 0 0x76

    mov word ptr [0aah], 76h                    ; @0  e7077600aa00
    mov word ptr [0b4h], 0f756h                 ; @6  e70756f7b400
    mov r8, 11ah                                ; @0ch  c8071a01
    mov word ptr [r8++], 4000h                  ; @10h  e0070040
    mov word ptr [r8++], 0                      ; @14h  e0070000
    mov word ptr [r8++], 0                      ; @18h  e0070000
    ret                                         ; @1ch  97cf
db 54 dup(0)                                    ; @1eh
db 0f1h, 0ffh, 0f2h, 0ffh, 0f3h, 0ffh, 0f4h, 0ffh, 0f5h, 0ffh, 0ffh, 0ffh ; @54h
db 0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0, 0, 54h, 0, 0, 0, 0, 0 ; @60h
db 0, 0, 0, 0, 0, 0                             ; @70h
    mov r0, byte ptr [r8 + 1]                   ; @76h  000e0100
    cmp r0, 51h                                 ; @7ah  c0575100
    jz loc_8ah                                  ; @7eh  05c0  x:loc_8ah

    cmp r0, 56h                                 ; @80h  c0575600
    jz loc_0aeh                                 ; @84h  14c0  x:loc_0aeh

    jmp loc_0e8h                                ; @86h  9fcfe800  x:loc_0e8h

[...]
```

Le code n'est toujours pas décodé dans son intégralité. On remarque cependant qu'à l'adresse `0x7a`, le registre `r0` est comparé aux valeur `0x51` puis `0x56`. On retrouve les valeurs passées en paramètre de la fonction `usb_control_msg` par la fonction `vicpwn_handle` (cf. 3.1). On peut donc supposer que le registre `r8` pointe vers une structure décrivant la requête en cours de traitement. Cette structure est décrite page 59 du manuel du BIOS [8] :

```
bmRequest equ 0
bRequest equ 1
wValue equ 2
wIndex equ 4
```

```
wLength equ 6
VND_VEC equ (SUSB1_VENDOR_INT*2)
```

Le code à l'adresse `0x8a` sert donc à traiter les requêtes de type `0x51` (les demandes de réception de données en provenance du programme `ssticrypt`). Cette routine de traitement est présentée ci-dessous :

```
// Xrefs: 7eh
loc_8ah:
    mov r9, 68h                                 ; @8ah  c9076800
    mov r1, word ptr [r9++]                     ; @8eh  4108
    mov r10, word ptr [r9++]                    ; @90h  4a08
    addi r8, 2                                  ; @92h  48d8
    mov word ptr [r10++], word ptr [r8++]       ; @94h  2208
    mov word ptr [r9++], word ptr [r8++]        ; @96h  2108
    mov r8, r9                                  ; @98h  4802
    xor word ptr [r9++], word ptr [r9]          ; @9ah  6194
    mov word ptr [r9++], r1                     ; @9ch  6100
    mov word ptr [r9++], 10h                    ; @9eh  e1071000
    mov word ptr [r9++], 0f6h                   ; @0a2h  e107f600
    mov r1, 8000h                               ; @0a6h  c1070080
    int 51h                                     ; @0aah  51af
    ret                                         ; @0ach  97cf
```

Le code en question initialise une structure de données puis déclenche l'interruption `0x51` (81 en décimal). Cette interruption est décrite page 44 du manuel du BIOS [8], il s'agit de l'interruption nommée `SUSBx_RECEIVE_INT`. La structure de données y est également décrite :

```
R8: points at an 8-byte control header block structure defined as follows:
dw next_link: pointer (used by this routine, input must be 0x0000)
dw address: pointer to the address of the device that is sending data.
dw length: length of data to send
dw call_back: pointer of the "call back" subroutine.
```

Le registre `r1` pointe donc vers l'adresse destination, la valeur `0x10` précise la longueur de données à recevoir et l'adresse `0xf6` est l'adresse de la routine de traitement des données après réception qui constitue donc un point d'entrée supplémentaire dans le code de la ROM.

De façon analogue à la routine précédente, le code à l'adresse `0xae` traite les requêtes de type `0x56` (les demandes d'envois de données de la webcam vers le programme `ssticrypt`). Le code de cette routine est présenté ci-dessous :

```
// Xrefs: 84h
loc_0aeh:
    mov r9, 68h                                 ; @0aeh  c9076800
    mov r1, word ptr [r9++]                     ; @0b2h  4108
    mov r12, r1                                 ; @0b4h  4c00
    add r12, 10h                                ; @0b6h  cc171000
    mov r10, 120h                               ; @0bah  ca072001
    mov word ptr [r10++], r12                   ; @0beh  2203
    mov word ptr [r10++], word ptr [r12]        ; @0c0h  2205
    mov word ptr [r10++], word ptr [r12 + 2]    ; @0c2h  220d0200
    mov r8, word ptr [r9++]                     ; @0c6h  4808
    mov r3, 14h                                 ; @0c8h  c3071400
    mov word ptr [r12++], word ptr [r8]         ; @0cch  2404
```

```
    mov r9, 6eh                                 ; @0ceh  c9076e00
    mov word ptr [r12], word ptr [r9]           ; @0d2h  5404
    mov r8, r9                                  ; @0d4h  4802
    xor word ptr [r9++], word ptr [r9]          ; @0d6h  6194
    mov word ptr [r9++], r1                     ; @0d8h  6100
    mov word ptr [r9++], r3                     ; @0dah  e100
    mov word ptr [r9++], 0e8h                   ; @0dch  e107e800
    mov r1, 8000h                               ; @0e0h  c1070080
    int 50h                                     ; @0e4h  50af
    ret                                         ; @0e6h  97cf
```

L'interruption `0x50` (`SUSBx_SEND_INT`) est déclenchée à la fin de la routine. La structure initialisée pointe vers une routine à l'adresse `0xe8` (qui est la routine de traitement des requêtes par défaut).

Pour tous les autres types de requête, le code à l'adresse `0xe8` est exécuté.

```
// Xrefs: 86h
loc_0e8h:
    int 59h                                     ; @0e8h  59af
    mov r10, 120h                               ; @0eah  ca072001
    mov r12, word ptr [r10++]                   ; @0eeh  8c08
    mov word ptr [r12++], word ptr [r10++]      ; @0f0h  a408
    mov word ptr [r12++], word ptr [r10++]      ; @0f2h  a408
    ret                                         ; @0f4h  97cf
```

En conclusion, en analysant la mise en place des différentes interruptions, il a été possible d'identifier les points d'entrée suivant :

- `0` : définition de la routine de traitement de l'interruption `SUSBx_VENDOR_INT` ;
- `0x76` : routine d'« aiguillage » des requêtes USB ;
- `0xf6` : routine de traitement des données reçues ;
- `0xe8` : routine de traitement par défaut des requêtes.

Ces points d'entrées doivent être spécifiés à Metasm pour désassembler l'intégralité du code de la ROM.

3.3.3 Identification de l'adresse de base

Lors de l'initialisation de la ROM par les deux appels à la fonction `vicpwn_sendbuf`, la mémoire externe de la webcam est reprogrammée. Le schéma page 6 du manuel du BIOS [8] détaille l'organisation de la mémoire et précise que la mémoire externe est accessible à l'adresse `0x4000`. Cette adresse est donc choisie comme adresse de base pour désassembler le code de la ROM.

3.3.4 Gestion des relocations

L'étude des structure de signature contenues dans le bloc de données `stage2_rom` (cf. tableau 2) a mis en évidence la nécessité d'effectuer des relocations sur le code de la ROM.

Le script Ruby ci-dessous permet d'effectuer ces relocations :

```ruby
#!/usr/bin/env ruby

MAP_ADDR = 0x4000

binary = File.open(ARGV.shift, "rb").read.unpack('S*')
relocs = File.open(ARGV.shift, "rb").read.unpack('S*')
out = File.open(ARGV.shift, "wb")

relocs.each do |r|
  raise unless (r % 2) == 0
  before = binary[r / 2]
  binary[r / 2] = before + MAP_ADDR
end

out.write( binary.pack('S*'))
out.close
```

Il faut donc exécuter ce script sur les données de la première signature en utilisant les informations de relocation de la deuxième signature :

```
$ ruby reloc.rb stage2_rom_scan_record_0 stage2_rom_scan_record_1 \
    stage2_rom_scan_record_0.relocated
```

Metasm peut alors désassembler ces données en spécifiant l'adresse de base `0x4000` (les points d'entrée doivent être ajustés en conséquence) :

```
$ ruby -I. samples/disassemble.rb --fast --rebase 0x4000 --cpu CY16 \
    stage2_rom_scan_record_0.relocated 0x4000 0x4076 0x40f6 0x40e8
```

3.3.5 Instructions non documentées

Après avoir découvert les différents points d'entrée de la ROM, identifié l'adresse de la base et appliqué les relocations, le désassemblage de la ROM devrait être complet (à l'exception des segments de données). En réalité, Metasm échoue à désassembler certaines parties du code :

```
$ ruby -I. samples/disassemble.rb --fast --rebase 0x4000 --cpu CY16 \
    stage2_rom_scan_record_0.relocated 0x4000 0x4076 0x40f6 0x40e8 | grep -C 3 "^db"

[...]

// Xrefs: 4290h 432ch 45f6h
--
    mov r5, word ptr [411ch]                    ; @42f0h  c5091c41
    add r5, r14                                 ; @42f4h  8513
    xor r1, r1                                  ; @42f6h  4190
db 0c6h, 0dfh, 0ah, 0c2h, 40h, 1, 9fh, 0afh     ; @42f8h
db 62h, 41h, 0ch, 80h, 0eh, 0d8h, 0cah, 17h, 0f8h, 0ffh, 0, 90h, 0c6h, 0dfh, 6ch, 0c3h ; @4300h


// Xrefs: 42ech
--
```

```
    add r9, r0                                  ; @4472h  0910
    add word ptr [412ah], word ptr [r9 + 40feh] ; @4474h  671cfe402a41
    add r1, 13h                                 ; @447ah  c1171300
db 0c7h, 0dfh                                   ; @447eh
db 62h, 0c9h                                    ; @4480h


// Xrefs: 443eh
--
loc_46a6h:
    addi word ptr [411ah], 2                    ; @46a6h  67d81a41
    clc                                         ; @46aah  c3df
db 0c6h, 0dfh, 2, 0c2h                          ; @46ach
db 9fh, 0cfh, 0aeh, 45h                         ; @46b0h
```

Metasm semble avoir des difficultés pour désassembler les valeurs suivantes : `0xdfc6` et `0xdfc7`. Le codage en binaire de ces deux valeurs est la suivante :

```
ruby-1.9.3-p0 :001 > "%016b" % 0xdfc6
 => "1101111111000110"
ruby-1.9.3-p0 :002 > "%016b" % 0xdfc7
 => "1101111111000111"
```

En comparant ce codage binaire avec les instructions existantes telles que décrites dans le document [7], on remarque que les instructions `stc` (1101111111000010) et `clc` (110111111100011) en sont très proches. On peut alors supposer que les instructions inconnues de Metasm (pour les valeurs `0xdfc6` et `0xdfc7`) sont des équivalents de `stc` et `clc` mais pour un autre drapeau. Il est alors nécessaire de modifier le code de Metasm pour ajouter le support de ces instructions :

```
diff --git a/metasm/cpu/cy16/opcodes.rb b/metasm/cpu/cy16/opcodes.rb
index 230a7ec..a323318 100644
--- a/metasm/cpu/cy16/opcodes.rb
+++ b/metasm/cpu/cy16/opcodes.rb
@@ -71,6 +71,8 @@ class CY16
                addop 'cli', (13<<12) | (7<<9) | (7<<6) | 1
                addop 'stc', (13<<12) | (7<<9) | (7<<6) | 2
                addop 'clc', (13<<12) | (7<<9) | (7<<6) | 3
+               addop 'stx', (13<<12) | (7<<9) | (7<<6) | 6
+               addop 'clx', (13<<12) | (7<<9) | (7<<6) | 7
        end

        alias init_latest init_cy16
```

Grâce à cette modification, le désassemblage de la ROM est alors complet :

```
$ ruby -I. samples/disassemble.rb --fast --rebase 0x4000 --cpu CY16 \
    stage2_rom_scan_record_0.relocated 0x4000 0x4076 0x40f6 0x40e8 | grep -C 3 "^db"
    mov word ptr [r8++], 0                      ; @4014h  e0070000
    mov word ptr [r8++], 0                      ; @4018h  e0070000
    ret                                         ; @401ch  97cf
db 54 dup(0)                                    ; @401eh
db 0f1h, 0ffh, 0f2h, 0ffh, 0f3h, 0ffh, 0f4h, 0ffh, 0f5h, 0ffh, 0ffh, 0ffh ; @4054h
db 0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0, "@T@", 0, 0, 0, 0 ; @4060h
db 0, 0, 0, 0, 0, 0                             ; @4070h
    mov r0, byte ptr [r8 + 1]                   ; @4076h  000e0100
```

```
    cmp r0, 51h                             ; @407ah  c0575100
    jz loc_408ah                            ; @407eh  05c0  x:loc_408ah
--
    call sub_44dah                          ; @40f6h  9fafda44  x:sub_44dah
    int 59h                                 ; @40fah  59af
    ret                                     ; @40fch  97cf
db 72 dup(0)                                ; @40feh

// Xrefs: 4290h 432ch 45f6h
```

Les données restantes qui ne sont pas désassemblées ne sont pas du code mais constituent les segments de données du programme.

3.3.6 Définition des segments

Le résultat du désassemblage tel que présenté au paragraphe précédent nous permet d'identifier rapidement les segments de code et de données :

- `0x4000` à `0x401c` (30 octets) : code ;
- `0x401e` à `0x4074` (88 octets) : données ;
- `0x4076` à `0x40fc` (136 octets) : code ;
- `0x40fe` à `0x4144` (72 octets) : données ;
- `0x4146` à `0x48a6` (1890 octets) : code.

En réalité, le premier segment code sert également de segment de données après la première exécution. La figure 5 montre qu'avant le déclenchement de l'interruption `0x51` (`SUSBx_RECEIVE_INT`), l'adresse mémoire de destination des données est stockée dans le registre `r1` (instruction à l'adresse `0x409c`). Or `r1` est initialisé avec la valeur stockée à l'adresse `0x4068` qui s'avère être `0x4000`. En définitive, les 16 octets reçus sont stockés à l'adresse `0x4000`.

```
405eh    db 0ah dup(0ffh)
4068h    dw 4000h
406ah    dw 4054h
406ch    db 0ah dup(0)

entrypoint_4076h:
4076h    mov r0, byte ptr [r8 + 1]
407ah    cmp r0, 51h
407eh    jz loc_408ah                   ; x:loc_408ah
4080h    cmp r0, 56h
4084h    jz loc_40aeh                   ; x:loc_40aeh
4086h    jmp loc_40e8h                  ; x:loc_40e8h

// Xrefs: 407eh
loc_408ah:
408ah    mov r9, 4068h
408eh    mov r1, word ptr [r9++]
4090h    mov r10, word ptr [r9++]
4092h    addi r8, 2
4094h    mov word ptr [r10++], word ptr [r8++]
4096h    mov word ptr [r9++], word ptr [r8++]
4098h    mov r8, r9
409ah    xor word ptr [r9++], word ptr [r9]
409ch    mov word ptr [r9++], r1
409eh    mov word ptr [r9++], 10h
40a2h    mov word ptr [r9++], 40f6h
40a6h    mov r1, 8000h
40aah    int 51h
40ach    ret
```

FIGURE 5 – Écriture à l'adresse `0x4000`

3.4 Point d'étape

Arrivé à cette étape du challenge, il est nécessaire de faire un point sur les résultats obtenus ainsi que la démarche à adopter pour la suite.

Nous sommes arrivés à :
- comprendre le fonctionnement global du programme `ssticrypt` : principalement l'établissement de l'échange avec la webcam et les mécanismes de vérification des données renvoyées par la webcam ;
- identifier l'architecture matérielle de la webcam et obtenir les documents de spécification du BIOS [8] et du processeur [7] ;
- désassembler intégralement la ROM programmée au niveau de la webcam ;
- identifier les différents points d'entrée dans cette ROM, notamment au niveau des routines déclenchées lors de la réception ou de l'envoi de données ;
- identifier les segments de code et de données de la ROM.

Pour finir le challenge, le but recherché est : arriver à comprendre la logique implémentée dans la ROM pour déterminer quelles sont les entrées (en l'occurence la clé) qui déclenchent les conditions recherchées dans la fonction `vicpwn_check` du programme `ssticrypt`. Deux approches sont alors envisageables :
- une approche dite dynamique : il s'agit alors d'obtenir une trace d'exécution de la logique programmée dans la ROM sur une entrée de test (une clé choisie au hasard par exemple) puis d'analyser la trace produite ;
- une approche dite statique : le code assembleur obtenu à l'aide de Metasm est alors commenté pour en comprendre la logique.

Dans un premier temps, j'ai choisi de m'intéresser à l'approche dynamique qui me semblait plus simple à mettre en œuvre. Cependant, une telle approche nécessite une implémentation de référence pour pouvoir générer des traces d'exécution fiables à partir d'entrées de test. Ne disposant physiquement pas de la webcam HomeConnect [6], j'ai alors entrepris de développer un émulateur à partir des spécifications du processeur (chapitre 3.5).

Dans un second temps, j'ai décidé de compléter les résultats obtenus par la démarche exposée précédemment en analysant le code assembleur produit et en ré-implémentant la logique du programme en langage C (chapitre 3.6).

3.5 Développement d'un émulateur

L'objectif du développement d'un émulateur est de pouvoir disposer d'une implémentation de référence côté webcam pour le reste du challenge. Le processeur de la webcam étant un processeur RISC, le jeu d'instruction reste relativement simple. Cependant, quelques subtilités sont à prendre en compte pour obtenir un émulateur fiable :
- les registres du processeur sont « mappés » en mémoire en fonction de la valeur du registre `REGBANK` (cf. page 3 du manuel du processeur [7]) ;
- de même, les drapeaux de condition sont également « mappés » en mémoire à l'adresse

6. qui se négocie actuellement autour d'une centaine d'euros sur Ebay

`0xc000` (cf. page 4 du même manuel) ;
- le registre `r15` est utilisé comme pointeur de pile : dans certains modes d'adressage, ce registre doit être pré-décrementé lors d'un accès en écriture (pour simuler l'instruction `push`) et post-incrémenté lors d'un accès en lecture (pour simuler l'instruction `pop`). Ce comportement est décrit page 8 du même manuel ;
- l'utilisation d'un mode d'adressage en particulier nécessite d'incrémenter automatiquement les valeurs des registres utilisés (cf. page 8 du même manuel).

De plus, certaines parties du code de la ROM sont auto-modifiantes (cf. paragraphe 4.1.3) : l'émulateur doit donc décoder les instructions dynamiquement lors de l'exécution.

Pour implémenter l'émulateur, j'ai décidé de me baser sur Metasm. Le traitement principal de l'émulateur est relativement simple [7] :

méthodes principales de l'émulateur

```ruby
class Emulator

  def initialize
    @cpu = Metasm::CY16.new
    @mem = Memory.new(0x10000)
    [...]
  end

  def decode_next_ins(addr)
    @mem.ptr = addr
    di = @cpu.decode_instruction(@mem, addr)
    puts "invalid instruction at 0x%04x : 0x%04x" % [addr, rw(addr)] unless di
    return di
  end

  def execute(addr)
    @pc = verbose

    while @pc
      di = decode_next_ins(@pc)
      raise "invalid instruction at 0x#{@pc.to_s(16)}" unless di
      interpret(di)
      di.instruction.args.each { |a| handle_autoinc(a) }
    end
    return true
  end

  def interpret(di)
    [...]
  end
end
```

La méthode `interpret` (non détaillée) effectue les traitements spécifiques à chaque instruction. Le code source complet de l'émulateur est disponible en annexe (fichier `lib/cy16/metaemul.rb` A.2.2).

De plus, l'unité arithmétique et logique est également émulée pour permettre la prise en compte des drapeaux `carry` et `overflow` lors des opérations d'addition et de soustraction. Cela implique une certaine lenteur au niveau de l'émulateur mais permet de rester au plus proche

7. le code a été volontairement simplifié dans un but pédagogique

des spécifications du processeur.

Des tests unitaires ont été développés pour s'assurer de la bonne implémentation de chaque instruction :

```
$ rake
ruby -I"lib" "test/tc_alu.rb" "test/tc_ins.rb"
Run options:

# Running tests:

..........

Finished tests in 0.006555s, 1525.4609 tests/s, 12203.6869 assertions/s.

10 tests, 80 assertions, 0 failures, 0 errors, 0 skips
```

Une interface graphique a été également développée pour faciliter l'analyse des traces d'exécution. Celle-ci est présentée à la figure 6.

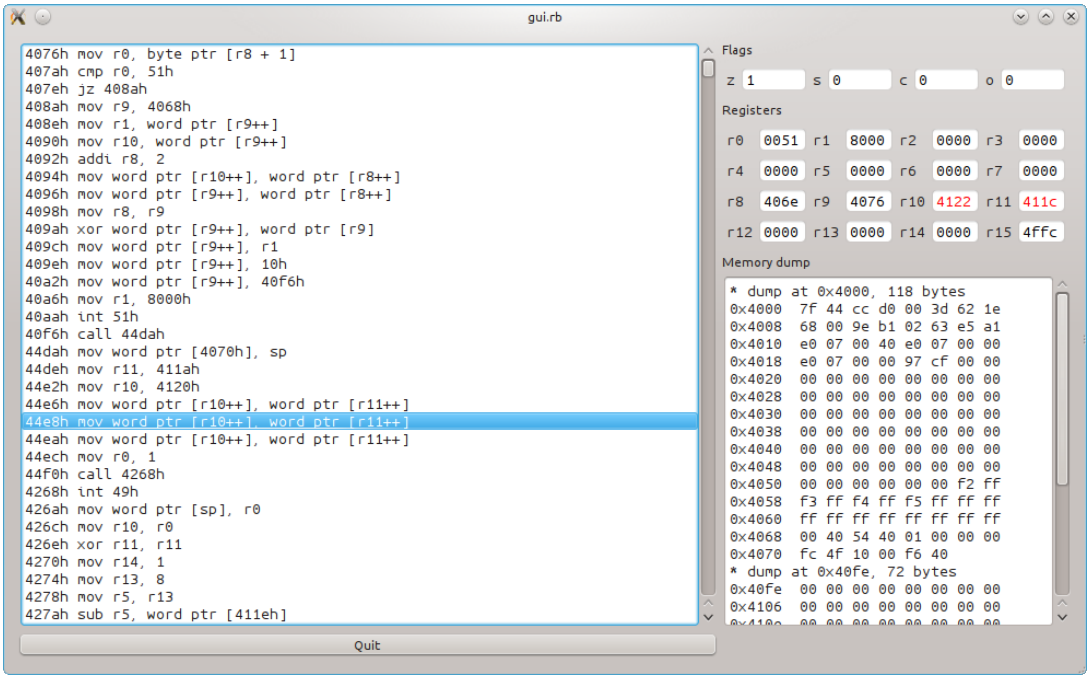

FIGURE 6 – Interface graphique de l'émulateur

La prochaine étape est de pouvoir réaliser un échange entre le programme `ssticrypt` et l'émulateur, ce qui revient à tester le début de la clé. Pour cela, il m'a semblé pertinent de faire appel au binaire original MIPS plutôt que d'utiliser le résultat de la rétro-conception (afin de se prémunir de potentiels défauts introduits pendant la réécriture en C depuis l'assembleur MIPS).

Cependant, le binaire MIPS cherche à communiquer en USB avec la webcam. Pour arriver à faire communiquer ce binaire avec l'émulateur, j'ai décidé de développer une bibliothèque partagée `usb-hook.so` (code A.2.2) qui intercepte les appels aux fonctions USB et utilise une socket TCP pour interagir avec la webcam. Cette bibliothèque partagée peut être alors chargée avec la variable d'environnement `LD_PRELOAD`.

La fonction qui nous intéresse principalement est la fonction `usb_control_msg` dont le code est présenté ci-dessous :

redéfinition de la fonction usb_control_msg

```
int usb_control_msg(usb_dev_handle *dev, int requesttype, int request, int value, int index,
    char *bytes, int size, int timeout) {
    char buf[24];
    int ret;

    printf("usb_control_msg(%p, %x, %x, %x, %x, %p, %x, %x): ",
            dev, requesttype, request, value, index, bytes, size, timeout);

    buf[1] = value & 0xff;
    buf[2] = (value >> 8) & 0xff;
    buf[3] = index & 0xff;
    buf[4] = (index >> 8) & 0xff;
    buf[5] = size & 0xff;
    buf[6] = (size >> 8) & 0xff;

    switch(request) {
        case 0x51:
            print_bytes(bytes, size);
            buf[0] = 0x51;
            memcpy(buf + 7, bytes, size);
            write(sockfd, buf, 7 + size);
            ret = 0;
            break;
        case 0x56:
            buf[0] = 0x56;
            write(sockfd, buf, 7);
            ret = read(sockfd, bytes, size);
            print_bytes(bytes, ret);
            break;
        default:
            printf("unhandled request: 0x%x\n", request);

    };

    return ret;
}
```

Les fonctions `usb_open` et `usb_close` sont respectivement chargées d'établir et de fermer la session TCP avec la webcam (dans notre cas, l'émulateur CY16).

L'émulateur est invoqué par le script ci-dessous :

lancement de l'émulateur avec support réseau

```
#!/usr/bin/env ruby

BIN_FILE = "~/git/sstic2012/input/ssticrypt"
STAGE2_ROM_OFFSET = 0x4772c
REMAP_ADDR        = 0x4000
STACK_BASE        = 0x5000
ENTRY_POINT       = REMAP_ADDR + 0x76

emul = CY16::Emulator.new(STACK_BASE, REMAP_ADDR)
emul.load_file(File.expand_path(BIN_FILE), STAGE2_ROM_OFFSET)

server = TCPServer.new(31337)
```

```
loop do
  client = server.accept
  emul.client = client

  while h = emul.client.read(7) do
    a = h.bytes.to_a
    request = a[0]
    value   = a[1] | (a[2] << 8)
    index   = a[3] | (a[4] << 8)
    size    = a[5] | (a[6] << 8)
    puts "request = 0x%x, value = 0x%x, index = 0x%x, size = 0x%x" % [request, value, index, size]
    case request
    when 0x51
      emul.gpr[8] = 0x300
      emul.gpr[9] = 0x200
      emul.wb(0x301, request)
      emul.ww(0x302, value)
      emul.ww(0x304, index)
      emul.ww(0x306, size)
      emul.execute(ENTRY_POINT)
    when 0x56
      emul.gpr[8] = 0x300
      emul.gpr[9] = 0x200
      emul.wb(0x301, request)
      emul.ww(0x302, value)
      emul.ww(0x304, index)
      emul.ww(0x306, size)
      emul.execute(ENTRY_POINT)
    end
  end

  client.close
end
```

Pour tester l'émulateur, il suffit alors de lancer le binaire `ssticrypt` depuis un environnement `qemu` MIPS en chargeant la bibliothèque partagée `usb-hook.so`.

trace d'exécution de ssticrypt sous qemu

```
$ TARGET="192.168.0.1" LD_PRELOAD="./usb-hook.so" ./ssticrypt -d \
    fd4185ff66a94afda1a2a3a4a5a6a7a8 secret
--> SSTICRYPT <--
Warning: MD5 mismatch for container
Using keys fd4185ff66a94afd / a1a2a3a4a5a6a7a8 ...
usb_init() called
usb_find_busses() called
usb_find_devices() called
usb_get_busses() called
usb_open() called
usb_control_msg(0x1, 40, ff, 0, 0, 0x4575b4, 44, 3e8): unhandled request: 0xff
usb_close() called
usb_init() called
usb_find_busses() called
usb_find_devices() called
usb_get_busses() called
usb_open() called
usb_control_msg(0x1, 40, ff, 0, 0, 0x45772c, a76, 3e8): unhandled request: 0xff
usb_close() called
usb_init() called
usb_find_busses() called
usb_find_devices() called
```

```
usb_get_busses() called
usb_open() called
usb_control_msg(0x1, 40, 51, 0, 1, 0x46a418, 10, 3e8): 7f44ccd0003d621e68009eb10263e5a1
usb_control_msg(0x1, c0, 56, 0, 0, 0x46a418, 14, 3e8): 00000000000000000000000000000000f3ff00a0
usb_control_msg(0x1, 40, 51, a000, 1, 0x46a418, 10, 3e8): a2a1a4a300000000000000000000000
usb_control_msg(0x1, c0, 56, 0, 0, 0x46a418, 14, 3e8): 00000000000000000000000000000000f5ff1000
[...]
usb_control_msg(0x1, 40, 51, 650, 1, 0x46a418, 10, 3e8): 0a0154f3e0050000ff80000000000000
usb_control_msg(0x1, c0, 56, 0, 0, 0x46a418, 14, 3e8): 20500aa78c40a0154f9881402a9ef10210060000
Error: bad key2
```

119 échanges (envoi / réception de données) sont effectués entre le binaire `ssticrypt` et l'émulateur. Finalement, la validation échoue et le message `Error: bad key2` est affiché. On remarquera que lors du second échange, les quatre premiers octets (`a2a1a4a3`) de la seconde partie de la clé sont envoyés à l'émulateur. Enfin, les dernières données envoyées à l'émulateur correspondent à la fin du bloc de données `layer1` :

```
$ hexdump -C layer1|tail -n 3
00000640  88 81 40 2a 9f 91 02 80  55 3d e2 05 00 aa 78 c4  |..@*....U=....x.|
00000650  0a 01 54 f3 e0 05 00 00  ff 80                    |..T.......|
0000065a
```

On peut donc supposer que l'intégralité du bloc `layer1` a été envoyé par le programme `ssticrypt`. La trace complète est disponible en annexe (fichier `trace-20120425.txt` A.2.2).

L'approche dynamique atteint ses limites : il est possible de tester une clé depuis le binaire `ssticrypt` mais la trace d'exécution au niveau de l'émulateur est bien trop complexe pour permettre d'en tirer des conclusions sur la logique programmée au niveau de la webcam. L'analyse du code assembleur fonction par fonction semble alors inévitable.

3.6 Analyse du code assembleur et réimplémentation en C

L'objectif de cette étape est d'implémenter (à partir du code assembleur) la logique programmée dans la webcam vers le langage C pour augmenter le niveau d'abstraction et faciliter la compréhension.

Pour tenter de minimiser les erreurs d'implémentation, la démarche globale que j'ai adoptée est la suivante :

- parcours linéaire du code assembleur en commentant ligne par ligne ;
- implémentation en C de chaque fonction en restant très proche de l'assembleur (garder les registres comme nom de variable, remplacer les instructions `jmp` par des `goto`, etc.) ;
- instrumentation de l'émulateur pour générer des tests unitaires à chaque appel de fonction ;
- utilisation d'un « débogueur » en cas d'échec d'un test unitaire pour analyser instruction par instruction le comportement de l'émulateur ;
- dès que tous les tests unitaires sont validés, réalisation d'une opération de raffinement du code C :
  - renommage des fonctions et des variables locales,
  - remplacement des `goto` par des boucles `for` ou `while`,
  - utilisation de directives `define` pour remplacer les adresses mémoires par des noms symboliques ;

– après chaque étape de raffinement ou de simplification du code C, les tests unitaires sont déroulés pour détecter la moindre régression.

Cette démarche, bien que fastidieuse, permet de s'assurer que le code C obtenu est conforme au fonctionnement de l'émulateur (et donc de la webcam).

3.6.1 Identification des fonctions

Une fois le code désassemblé, l'interface graphique de Metasm permet de générer le graphe d'appels des fonctions (figure 7).

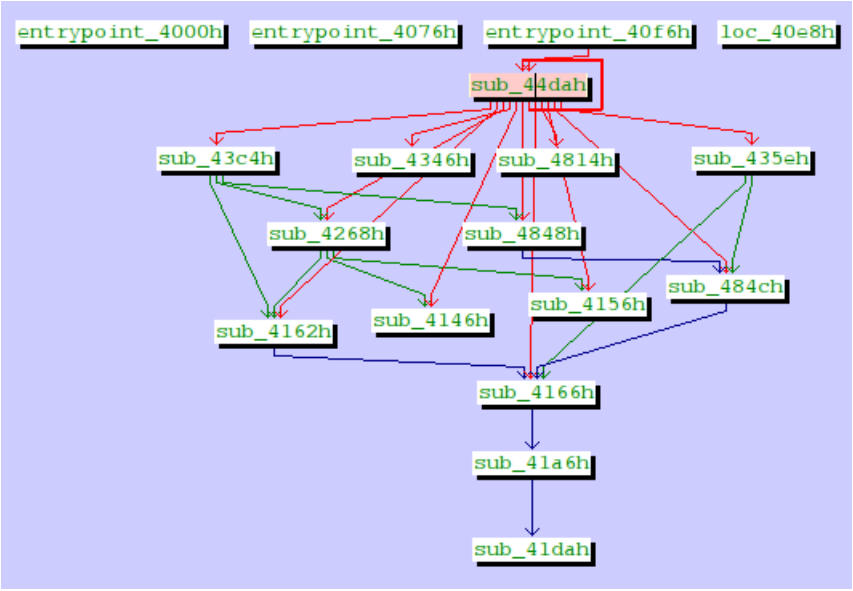

FIGURE 7 – Graphe d'appels de la ROM

Comme cela a été fait précédemment pour le binaire `ssticrypt` MIPS, il est possible de générer un graphe au format Graphviz (figure 8).

L'analyse du graphe d'appels permet de tirer certaines conclusions :
– le point d'entrée de traitement des données reçus semble être la fonction `sub_44da`. De plus, cette fonction implémente une boucle ;
– les fonctions `sub_4146`, `sub_4156`, `sub_41da`, `sub_4346` et `sub_4814` sont terminales (elles n'appellent pas d'autres fonctions) ;
– la fonction `sub_4166` est appelée par quatre autres fonctions.

3.6.2 Génération des tests unitaires

L'émulateur présenté précédemment peut être très facilement instrumenté pour générer automatiquement des tests unitaires afin de valider l'implémentation des différentes fonctions en C.

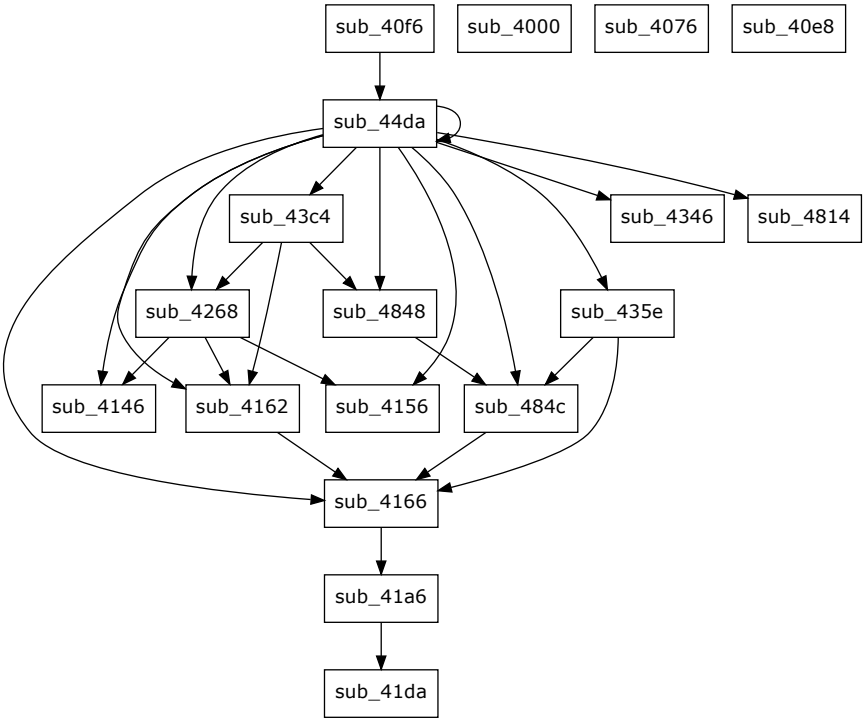

FIGURE 8 – Graphe d'appels de la ROM (version graphviz)

Le principe est le suivante :
- au démarrage de l'exécution, l'émulateur initialise une pile d'état (un état est défini par la valeur des registres, des drapeaux de condition et des segments de données à un instant précis) ;
- lors du traitement d'une instruction `call`, un état est sauvegardé puis empilé ;
- lors du traitement d'une instruction `ret`, le dernier état empilé est dépilé et un nouvel état est sauvegardé ;
- à partir de l'état avant `call` et de l'état `ret`, il est possible de connaître les valeurs en entrée de la fonction ainsi que les valeurs attendues en sortie.

A titre d'exemple, un test unitaire est présenté ci-dessous :

```
[...]
sw(0x4134, 0x56fe);
sw(0x4136, 0x4);
sw(0x4138, 0xc56a);
sw(0x413a, 0xc);
sw(0x413c, 0x1);
sw(0x413e, 0x3);
sw(0x4144, 0x11);
sw(0xc000, 0x1);

ret = sub_4162(0x5e, 0x0, 0x4054);
printf("ret: 0x%04x == 0xe9\n", ret);
assert(ret == 0xe9);
```

```
    printf("w[0x4072]: 0x%04x == 0x10 ?\n", rw(0x4072));
    assert(rw(0x4072) == 0x10);
    printf("w[0x4054]: 0x%04x == 0x60 ?\n", rw(0x4054));
    assert(rw(0x4054) == 0x60);
    printf("w[0x4056]: 0x%04x == 0x40 ?\n", rw(0x4056));
    assert(rw(0x4056) == 0x40);
    [...]
```

La macro `sw` écrit une valeur en mémoire tandis que la macro `rw` retourne la valeur spécifiée par l'adresse en paramètre. La valeur de retour de la fonction ainsi que les valeurs à différents emplacements mémoires sont testées avec la directive `assert`.

Le code du générateur de test est disponible en annexe (fichier `lib/testgenerator.rb` A.2.2).

3.6.3 Utilisation du débogueur

Lorsqu'un test unitaire n'est pas validé, il est parfois complexe d'en comprendre la cause. L'implémentation au sein de l'émulateur d'un débogueur permet d'examiner instruction par instruction le comportement attendu d'une fonction et de comparer les différents états intermédiaires avec l'implémentation en C de cette même fonction.

Par défaut, le débogueur arrête l'exécution du programme à l'adresse `0x4076` :

```
$ ruby bin/cli.rb
[+] write interrupt
[+] fix-up (relocate)
[+] call interrupt
[!] breaking at 0x4076: 4076h mov r0, byte ptr [r8 + 1]
> help
break addr: add a breakpoint at given address
del addr: remove breakpoint at given address
continue: continue the program execution until next breakpoint
step: break at next instruction
step: break at next instruction (stepping over function calls)
show (breakpoints|regs|mem): show the specified object
trace: enable / disable call tracing
exit: exit the debugger
```

Il est alors possible d'ajouter ou de supprimer des points d'arrêt, de consulter l'état des registres ou des segments de données :

```
> show regs

[!] registers :
r0 = 0x0000 r1 = 0x0000  r2 = 0x0000  r3 = 0x0000  r4 = 0x0000  r5 = 0x0000  r6 = 0x0000  r7 = 0x0000
r8 = 0x0300 r9 = 0x0200 r10 = 0x0000 r11 = 0x0000 r12 = 0x0000 r13 = 0x0000 r14 = 0x0000 r15 = 0x5000

> show mem

[!] dump at 0x4000, 118 bytes
0x4000  e7 07 76 40 aa 00 e7 07
0x4008  56 f7 b4 00 c8 07 1a 41
```

```
0x4010  e0 07 00 40 e0 07 00 00
0x4018  e0 07 00 00 97 cf 00 00
0x4020  00 00 00 00 00 00 00 00
[...]
```

Une dernière fonctionnalité intéressante est la possibilité de tracer les appels de fonction jusqu'au prochain point d'arrêt :

```
> trace
> cont
[+] 0x4000: recv(7f44ccd0003d621e68009eb10263e5a1) (len = 16)
0x40f6 call 0x44da, r0 = 0x51, r1 = 0x8000, r2 = 0x0, r3 = 0x0, r4 = 0x0, r5 = 0x0, r6 = 0x0
    0x44f0 call 0x4268, r0 = 0x1, r1 = 0x8000, r2 = 0x0, r3 = 0x0, r4 = 0x0, r5 = 0x0, r6 = 0x0
        0x4288 call 0x4162, r0 = 0x0, r1 = 0x0, r2 = 0x0, r3 = 0x0, r4 = 0x0, r5 = 0x8, r6 = 0x0
            0x41a8 call 0x41da, r0 = 0x0, r1 = 0x2, r2 = 0x0, r3 = 0x0, r4 = 0x0, r5 = 0x8, r6 = 0x0
            0x422c ret, r0 = 0x4000, r2 = 0x4054
        0x41d8 ret, r0 = 0x7f, r2 = 0x0
        0x4290 call 0x4146, r0 = 0x7f, r1 = 0x7, r2 = 0x0, r3 = 0x0, r4 = 0x0, r5 = 0x8, r6 = 0x0
        0x4154 ret, r0 = 0x0, r2 = 0x0
        0x429c call 0x4156, r0 = 0xffff, r1 = 0x1, r2 = 0x0, r3 = 0x0, r4 = 0x0, r5 = 0x8, r6 = 0x0
        0x4160 ret, r0 = 0xfffe, r2 = 0x0
    0x42c6 ret, r0 = 0x0, r2 = 0x0
    [...]
```

L'utilisation de la bibliothèque `readline` permet de fournir très facilement des fonctionnalités d'historique (avec recherche) et de complétion des commandes.

Le code du débogueur est disponible en annexe (fichier `lib/cy16/debugger.rb` A.2.2).

3.6.4 Sémantique des instructions `stx` et `clx`

Comme présenté au paragraphe 3.3.5, la ROM utilise des instructions non documentées dont la sémantique est inconnue.

En étudiant le positionnement de ces instructions dans le code assembleur, on remarque qu'elles précédent toujours des sauts conditionnels qui testent la valeur du drapeau `carry`. Par exemple,

```
42f8h stx
42fah jc loc_4310h ; x:loc_4310h     ; saute si carry <=> word_411c + r14 > 0xffff
42fch mov r0, r5
42feh call sub_4162h ; x:sub_4162h   ; r0 = sub_4162h(word_411c + r14, 0, v2)
4302h or r12, r0                     ; r12 = (sub_4162h(w11c, 0, v2) << 8) |
                                     ;     sub_4162h(word_411c + 1, 0, v2)
4304h addi r14, 1                    ; r14 += 1
4306h add r10, 0fff8h                ; r10 += 0fff8h
430ah xor r0, r0                     ; r0 = 0
430ch stx                            ;
430eh jnc loc_42e8h ; x:loc_42e8h    ; saute si pas carry <=> r10 + 0xfff8 <= 0xffff
[...]
447ah add r1, 13h                    ; r1 = 0xffff + 0x13 = 0x12 => z = 0, c = 1, o = 0, s = 0
447eh clX                            ; clear c ?
4480h jbe save_exit ; x:loc_4446h    ; saute si z == 1 ou c == 1
```

Deux interprétations sont alors possibles :
– les instructions `stx` et `clx` sont en fait équivalentes à `stc` et `clc` ;
– les instructions `stx` et `clx` sont équivalentes à une instruction `nop`.

Pour déterminer quelle interprétation est la bonne, il faut alors essayer de comprendre le sens global de la fonction qui utilise ces instructions. En examinant le code ci-dessous (extrait de la fonction `sub_43c4`), on se rend compte que si l'instruction `clx` est équivalente à `clc`, alors deux chemins de code sont identiques. L'instruction `clx` doit donc être interprétée comme une instruction `nop`.

```
switch(r0) {
    case 0: // loc_4482h
        reg = read_nbits(4);
        sw(SRC_REG, reg);
        sw(SRC_MEM_ADDR, READ_REG(reg));
        break;
    case 1: // loc_449ah
        r0 = read_nbits(16);
        sw(SRC_MEM_ADDR, r0);
        break;
    case 2: // loc_4454h
        r0 = read_nbits(16);
        sw(SRC_MEM_ADDR, r0);
        reg = read_nbits(4);
        sw(SRC_REG, reg);
        addw(SRC_MEM_ADDR, READ_REG(reg)); // si clc fait rien
        // sw(OP_MEM_ADDR, rw(0x40fe + 2 * r0); clX <=> clc, equivalent case 0
        break;
}
```

Une démarche similaire sur les cas d'utilisation de l'instruction `stx` permet de conclure que cette instruction doit également être interprétée comme uen instruction `nop`.

3.6.5 Qualification des différentes fonctions

Les différentes étapes de raffinement effectuées sur le code C permettent de comprendre au fur et à mesure la logique programmée au niveau de la webcam. Une approche ascendante (« bottom-up ») permet de comprendre l'utilité de chaque fonction. Le principe de cette démarche est de commencer à s'intéresser aux fonctions terminales (qui ne dépendent donc pas d'autres fonctions) puis de remonter le graphe d'appels jusqu'à la fonction principale, `sub_44da`.

En procédant ainsi, il est possible de remonter rapidement jusqu'à la fonction `sub_4268` (cf. figure 8). On remarque rapidement que cette fonction prend en entrée un entier (qui représente une taille) et retourne des données du bloc `layer1`. Les appels successifs à cette fonction permettent d'avancer dans le contenu de ce bloc de données.

La fonction appelante, `sub_44da`, semble effectuer un décodage des données du bloc `layer1` par des appels successifs à `sub_4268` puis un traitement spécifique est réalisé selon les valeurs d'un des champs décodés. Une fois ce traitement terminé, un saut est réalisé au début de la fonction `sub_44da` pour continuer à traiter le reste des données de `layer1`.

Ce comportement ressemble fortement à celui d'une machine virtuelle, le bloc `layer1` représentant alors une suite d'instruction et d'opérandes (autrement dit, du « bytecode » pour la machine virtuelle).

Par déduction, il est possible de comprendre le rôle des autres fonctions :
- `sub_44da` : décodage et interprétation d'une instruction (fonction principale) ;
- `sub_4146` : décalage à droite d'une valeur entière ;
- `sub_4156` : décalage à gauche d'une valeur entière ;
- `sub_4162` : lecture d'une valeur à l'adresse passée en paramètre ;
- `sub_4166` : lecture ou écriture d'une valeur à l'adresse passée en argument ;
- `sub_41da` : implémente un mécanisme de translation d'adresses et de cache de données entre la mémoire du client (`ssticrypt`) et celle de la webcam ;
- `sub_4268` : lecture de n bits de données du bloc `layer1` ;
- `sub_4346` : sauvegarde le résultat du décodage d'une opérande avant un nouveau décodage ;
- `sub_435e` : sauvegarde le résultat de l'évaluation d'une instruction à l'emplacement spécifié par l'opérande de destination ;
- `sub_43c4` : réalise le décodage d'une opérande.

Le fichier `server.c` constitue l'implémentation en C de la ROM de la webcam et est disponible en annexe (cf. A.2.3).

3.6.6 Conclusion intermédiaire

Une fois la partie webcam implémentée en C, il est alors possible de recoder en C la logique du programme `ssticrypt` (fichier `client.c` A.2.3) et de vérifier que les deux implémentations en C arrivent à simuler un échange identique à celui observé entre le binaire `ssticrypt` sous qemu et l'émulateur (cf. 3.5).

Au final, la logique côté client (`ssticrypt`) et celle côté serveur (`webcam`) est implémentée au sein d'un même binaire, `test_key`, dont le code source est présenté ci-dessous :

```
#include "server.h"
#include "client.h"
#include "utils.h"
#include <stdio.h>
#include <stdlib.h>

void test_key(char *k) {
    do_client_init();
    do_server_init();

    check_key(k, 1);
}

int main(int argc, char **argv) {
    verbose_level = INFO;
    test_key("a1a2a3a4a5a6a7a8");
    exit(EXIT_SUCCESS);
}
```

L'exécution de ce programme permet alors de tester une clé :

```
$ make test_key
gcc -O2 -W -Wall -Wno-unused -std=c99    -c -o test_key.o test_key.c
gcc -O2 -W -Wall -Wno-unused -std=c99    -c -o client.o client.c
gcc -O2 -W -Wall -Wno-unused -std=c99    -c -o server.o server.c
gcc -O2 -W -Wall -Wno-unused -std=c99    -c -o utils.o utils.c
gcc -o test_key test_key.o client.o server.o utils.o -O2 -W -Wall -Wno-unused -std=c99
$ ./test_key
[+] server.c:do_susb1_receive_int:301 0x4000 <- 7f44ccd0003d621e68009eb10263e5a1
[+] server.c:do_instruction:1423
-> PC = (0, 0), cond = 0, cond_bits = f:e, flags = 0:0, ins = mov, uf = 0
[+] server.c:decode_operands:985 REG r12 = 0x0
[+] server.c:decode_operands:1007 MEMREF IMM 0xa000
[...]
-> PC = (1614, 0), cond = 0, cond_bits = f:1, flags = 2:3, ins = mov, uf = 0
[+] server.c:decode_operands:985 REG r0 = 0xaa
[+] server.c:decode_operands:990 IMM 0xaa
[+] server.c:do_instruction:1423
-> PC = (1618, 7), cond = 0, cond_bits = f:3, flags = 2:3, ins = jmp7, uf = 0
[+] server.c:decode_operands:990 IMM 0x0
[+] server.c:sub_41da:639 needs more bytes, exiting...
[+] server.c:do_susb1_send_int:310 0x4010 -> 20500aa78c40a0154f9881402a9ef10210060000
[+] bad key for layer 0
```

La prochaine étape est de réaliser l'analyse de la machine virtuelle implémentée par la ROM de la webcam. Cette analyse est l'objet de la suite de ce document.

4 Analyse de la machine virtuelle

4.1 Détermination des spécifications

L'étude du fonctionnement de la machine virtuelle, en particulier de la fonction principale `sub_44da` (cf. fonction `do_instruction` du fichier `server.c` A.2.3), permet de mieux appréhender le jeu d'instruction accepté par celle-ci.

Une instruction de la machine virtuelle est composée de 13 bits comme présenté sur la figure 9.

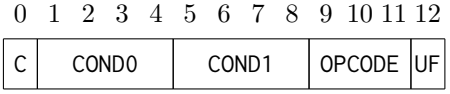

FIGURE 9 – Format d'une instruction de la machine virtuelle

Les différents champs qui constituent une instruction peuvent donc être décrits ainsi :
- `C` : 0 ou 1, précise quels sont les bits de condition qui vont être testés ;
- `COND0` : de 0 à 15, bits de condition testés si `C` vaut 0 ;
- `COND1` : de 0 à 15, bits de condition testés si `C` vaut 1 ;
- `OPCODE` : de 0 à 7, code de l'instruction ;
- `UF` : 0 ou 1, précise si les drapeaux de condition vont être mis à jour (`UF=1`) ou non (`UF=0`) suite à l'exécution de l'instruction.

Le décodage des instructions se fait à partir d'un flux de données. La position courante dans le programme est exprimée à l'aide de deux valeurs :

- une position en octets (stockée à l'adresse `0x411c`) ;
- un décalage exprimé en bits et relatif à la position en octets (stocké à l'adresse `0x411e`).

4.1.1 Instructions supportées

La machine virtuelle sait gérer 6 instructions différentes spécifiées par la valeur du champ `OPCODE` :

`OPCODE`	Instruction	Opérandes	Drapeaux affectés
0	`and`	source, destination	zero
1	`or`	source, destination	zero
2	`neg`	destination	zero, sign
3	`shift`	direction (1 bit), décalage (8 bits), destination	zero, sign, carry
4	`mov`	source, destination	aucun
5, 6, 7	`jmp`	décalage en octets (6 bits), décalage en bits (3 bits), destination si le décalage en octets est nul	aucun

Table 3 – Liste des instructions supportées par la machine virtuelle

Le traitement des opcodes 5, 6 et 7 commencent tous à l'adresse `0x4702`. Bien qu'un traitement spécifique soit présent à l'adresse `0x47ae` dans le cas où l'opcode est égal à 6, cette situation ne s'est jamais présentée dans le cadre de ce challenge. On peut imaginer que les auteurs de ce challenge avaient prévu d'utiliser l'instruction ayant pour opcode 6 mais n'ont pas pu le faire par manque de temps. De même, la présence d'un opcode égal à 5 n'a jamais été observée.

Au final, Les trois programmes (chaque « layer » correspondant à un programme) n'utilisent que les opcodes 0, 1, 2, 3, 4 et 7 (jamais 5 et 6).

Instructions `and` et `or`

Ces deux instructions sont traitées de manière similaire à l'adresse `0x4574`. Deux opérandes sont décodées, le calcul sur les valeurs des opérandes est effectué par le processeur CY16. Le drapeau `zero` de la machine virtuelle est mis à jour en fonction du résultat du calcul précédent. Enfin, le résultat est sauvegardé par la fonction `sub_435e` à l'emplacement spécifié par l'opérande destination.

Instruction `neg`

Cette instruction est traitée à l'adresse `0x45be`. L'opérande destination est décodée, le calcul est effectué par le processeur CY16, les drapeaux `zero` et `sign` sont mis à jour. Enfin, le résultat est sauvegardé par la fonction `sub_435e` à l'emplacement spécifié par l'opérande.

Instruction `shift`

Cette instruction est traitée à l'adresse `0x45d0`. Dans un premier temps, un bit est décodé et

spécifie la direction (0 pour gauche, 1 pour droite) du décalage. Ensuite la valeur du décalage est décodée sur 8 bits. L'opérande destination est décodée, le calcul est effectué par les fonctions `sub_4156` (décalage à gauche) et `sub_4146` (décalage à droite). Les drapeaux `zero`, `sign` et `carry` sont mis à jour. La valeur du drapeau `carry` est la valeur du dernier bit qui a disparu suite au décalage. Le résultat est sauvegardé par la fonction `sub_435e` à l'emplacement spécifié par l'opérande.

Instruction `mov`

Cette instruction est traitée à l'adresse `0x460a`. Les deux opérandes destination et source sont décodées. La valeur de l'opérande source est copiée à l'emplacement spécifié par l'opération destination.

Instruction `jmp`

Cette instruction est traitée à l'adresse `0x4702`. Deux décalages sont décodés, le premier sur 6 bits est un décalage en octet, le deuxième sur 3 bits est en décalage en bits. Si le premier décalage est nul, alors une opérande est décodée qui spécifie la destination du saut. La position courante (adresses `0x411c` et `0x411e`) est alors mise à jour en fonction de ces deux décalages.

4.1.2 Opérandes

Le décodage des opérandes est réalisé par la fonction `sub_43c4`. Il s'effectue en deux phases. La première est le décodage des informations communes à toutes les opérandes, à savoir :

- un bit précisant la taille des données (octet ou mot de 16 bits) adressées par l'opérande (champ nommé `S` par la suite) ;
- 2 bits précisant le type de l'opérande (champ nommé `T` par la suite).

La seconde phase est un décodage spécifique en fonction du type l'opérande. 4 types différents sont gérés par la machine virtuelle :

- registre (`T=0`) ;
- immédiat (`T=1`) ;
- référence mémoire (`T=2`) ;
- opérande dite « obfusquée » (`T=3`).

A la fin du décodage d'une opérande, les données suivantes sont disponibles (en fonction du type d'opérande) :

- `0x4126` : valeur de l'opérande ;
- `0x4128` : registre utilisé par l'opérande ;
- `0x412a` : adresse mémoire référencée par l'opérande ;
- `0x412c` : valeur utilisée pour l'obfuscation (champ `OPV`, cf 4.1.2) ;
- `0x412e` : taille (octet ou mot) de la donnée référencée par l'opérande ;
- `0x4130` : type de l'opérande.

Dans le cas des instruction manipulant plusieurs opérandes, il est nécessaire de pouvoir sauvegarder ces données avant de décoder une nouvelle opérande : c'est le rôle de la fonction `sub_4346` ci-dessous :

sauvegarde d'une opérande

```
// Xrefs: 4578h 45c2h 45e8h 464ch
sub_4346h:
// function binding: sp -> sp+2

// function ends at 435ch
    int 49h                                 ; @4346h  49af
    mov r11, word_4126h                     ; @4348h  cb072641
    mov r10, word_4134h                     ; @434ch  ca073441
    mov r0, 6                               ; @4350h  c0070600

// Xrefs: 4358h
loc_4354h:
    mov word ptr [r10++], word ptr [r11++]  ; @4354h  e208  r2:word_4126h w2:word_4134h
    subi r0, 1                              ; @4356h  00da
    jnz loc_4354h                           ; @4358h  7dc1  x:loc_4354h

    int 4ah                                 ; @435ah  4aaf
    ret
```

Opérande de type registre

Le format d'une opérande de type registre est présenté à la figure 10.

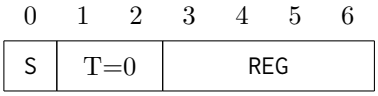

FIGURE 10 – Format d'une opérande de type registre

Lors du décodage de l'opérande, le numéro de registre est stocké à l'adresse `0x4128`, la valeur du registre est stockée à l'adresse `0x4126`.

Opérande de type immédiat

Le format d'une opérande de type immédiat dans le cas $S == 0$ est présenté à la figure 11.

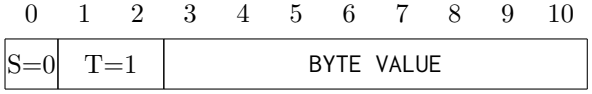

FIGURE 11 – Format d'une opérande de type immédiat (octet)

Le format d'une opérande de type immédiat dans le cas $S == 1$ est présenté à la figure 12.

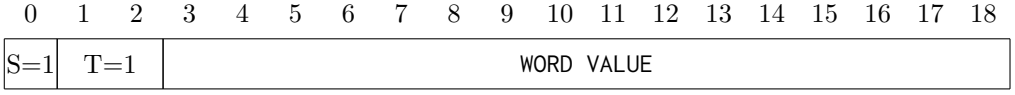

FIGURE 12 – Format d'une opérande de type immédiat (mot)

Dans les deux cas d'adressage, la valeur décodée est sauvegardée à l'adresse `0x4126`

Opérande de type référence mémoire

Par rapport aux autres types d'opérandes, un champ supplémentaire (`MT`) existe pour préciser le mode d'adressage utilisé (qui sont au nombre de trois).

Ces trois modes d'adressage sont :
- adressage direct par registre (`MT=0`) ;
- adressage direct par valeur immédiate sur 16 bits (`MT=1`) ;
- adressage indirect avec une adresse de base sur 16 bits et un décalage spécifié par un registre (`MT=2`).

Le mode d'adressage est précisé par un champ de deux bits.

Le format d'une opérande référence mémoire par registre est présenté à la figure 13.

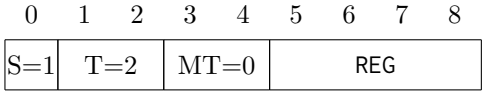

FIGURE 13 – Format d'une opérande référence mémoire par registre

Le format d'une opérande référence mémoire immédiate est présenté à la figure 14.

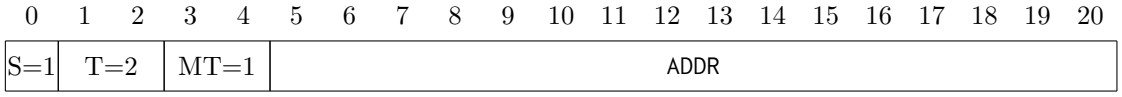

FIGURE 14 – Format d'une opérande référence mémoire immédiate

Le format d'une opérande référence mémoire immédiate est présenté à la figure 15.

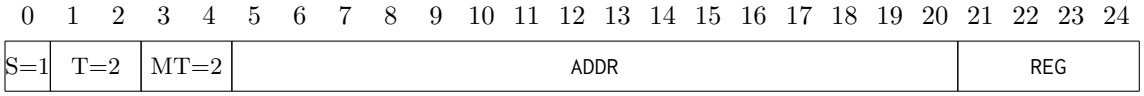

FIGURE 15 – Format d'une opérande référence mémoire indirecte

Opérande dite « obfusquée »

Le format d'une opérande dite de type « obfusquée » est présenté à la figure 16.

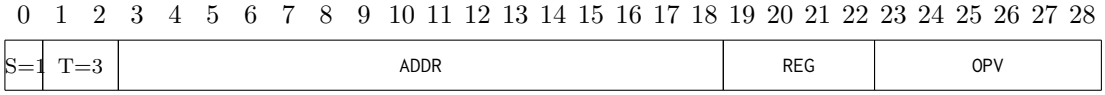

FIGURE 16 – Format d'une opérande dite « obfusquée »

Lors du décodage d'une telle instruction, la fonction `sub_4848` est appelée avec les valeurs des champs décodés comme paramètres.

fonctions sub_4848 et sub_484c

```
/* get a byte (size == 0) or word (size == 1) at address addr */
uint16_t sub_4848(uint16_t addr, uint16_t regv, uint16_t opv, uint16_t size) {
    return sub_484c(addr, regv, opv, (size + 1) << 1, 0);
}

/* get or set a byte / word at address addr */
uint16_t sub_484c(uint16_t addr, uint16_t regv, uint16_t opv, uint16_t op, uint16_t value) {
    uint16_t ret, r2, r5, r6 = 0, r7, r8;
    int i;

    r5 = (( ( (opv << 6) + opv) << 4 ) + opv) ^ 0x464d;
```

```
    r2 = addr ^ 0x6c38;
    for (i = 0; i < regv + 2; i++) {
        r7 = r6;
        r5 = ROL(r5, 1);
        r2 = ROR(r2, 2);
        r2 += r5;
        r5 += 2;
        r6 = r2 ^ r5;
        r8 = ( (r2 ^ r5) >> 8 ) & 0xff;
        r6 = (r6 & 0xff) ^ r8;
    }

    r6 = (r6 << 8) | r7;

    ret = sub_4166(addr + regv, op, value ^ r6) ^ r6;

    if (op != DO_GET_WORD)
        ret &= 0xff;

    return ret;
}
```

Ces deux fonctions sont chargées d'appliquer une routine d'obfuscation sur les valeurs à écrire ou à lire en mémoire : un certain entier est calculé à partir des paramètres `opv` et `regv` (qui détermine le nombre d'itérations). Le résultat du calcul est stocké dans le registre `r6` puis appliqué sur la valeur à sauvegarder ou à retourner à l'aide de l'opération `xor`. La fonction `sub_4166` est responsable, en fonction de la valeur du paramètre `op`, de la lecture ou de l'écriture en mémoire à l'adresse spécifiée par `addr + regv`.

Lors du décodage d'une telle opérande, la valeur retournée par la fonction `sub_4848` est sauvegardée à l'adresse `0x4126`.

4.1.3 Gestion des conditions

Chaque instruction possède deux jeux de bits de condition. L'exécution d'une instruction est donc conditionnée par le jeu de bits de condition qui va être retenu en fonction de la valeur du champ `C`.

Ces bits de condition sont codés de façon identique à ceux du processeur CY16 et sont présentés au tableau 4.

Au niveau de la gestion des conditions, les principales différences entre la machine virtuelle et le processeur CY16 sont :

- chaque instruction possède deux jeux de bits de condition ;
- la machine virtuelle possède deux jeux de drapeaux de condition (dont les valeurs sont stockées sur 8 bits à l'adresse 0x4144) ;
- chaque instruction de la machine virtuelle est conditionnelle alors que seules les instructions `jmp`, `call` et `ret` du processeur CY16 le sont.

Condition	Mnemonic	cccc Bits	Description
Z	Zero	0000	Z=1
NZ	Not Zero	0001	Z=0
C / B	Carry / Borrow	0010	C=1
NC / AE	Not Carry / Above or Equal	0011	C=0
S	Sign	0100	S=1
NS	Not Sign	0101	S=0
O	Overflow	0110	O=1
NO	Not Overflow	0111	O=0
A / NBE	Above / Not Below or Equal	1000	(Z=0 AND C=0)
BE / NA	Below or Equal / Not Above	1001	(Z=1 OR C=1)
G / NLE	Greater Than / Not Less Than or Equal	1010	(O=S AND Z=0)
GE / NL	Greater or Equal / Not Less Than	1011	(O=S)
L / NGE	Less Than / Not Greater or Equal	1100	(O !=S)
LE / NG	Less Than or Equal / Not Greater Than	1101	(O !=S OR Z=1)
(not used)		1110	
Unconditional	Unconditionally	1111	Unconditional

TABLE 4 – Bits de condition du processeur CY16

Évaluation d'une condition

Lors du décodage d'une instruction, le test sur les conditions est effectué à partir de l'adresse `0x4502`. A l'entrée de cette routine, le registre `r0` contient les bits de conditions de l'instruction (`COND0` et `COND`) et le registre `r14` contient la valeur du champ `C`.

L'extrait de code ci-dessous est particulièrement intéressant car il repose sur deux propriétés :
- la présence de code auto modifiant ;
- le fait que les drapeaux de condition du processeur CY16 soient accessibles en lecture / écriture à l'adresse `0xc000`.

Cette façon de faire laisse à penser que le code CY16 de la Webcam a été directement écrit en assembleur car il est peu probable qu'un compilateur puisse produire un tel code.

Le codage des drapeaux à l'adresse `0xc000` est décrit dans le manuel du processeur au paragraphe 1.3.4.

traitement des bits de condition de l'instruction

```
    cmp r0, 0ffh                            ; @4502h  c057ff00
    jz loc_4812h                            ; @4506h  9fc01248  x:loc_4812h

    mov r11, r0                             ; @450ah  0b00
    mov r12, word ptr [word_4144h]          ; @450ch  cc094441  r2:word_4144h
    and r14, r14                            ; @4510h  8e63
    jnz loc_4518h                           ; @4512h  02c1  x:loc_4518h

    shr r11, 4                              ; @4514h  cbd0
    shr r12, 4                              ; @4516h  ccd0

// Xrefs: 4512h
loc_4518h:
    and r11, 0fh                            ; @4518h  cb670f00
    add r11, 0c0h                           ; @451ch  cb17c000
```

```
    and r12, 0fh                            ; @4520h  cc670f00
    mov byte ptr [byte_4537h], r11          ; @4524h  ef023745  w1:byte_4537h
    mov r0, word ptr [0c000h]               ; @4528h  c00900c0  r2:0c000h
    and r0, 0fff0h                          ; @452ch  c067f0ff
    or r0, r12                              ; @4530h  0083
    mov word ptr [0c000h], r0               ; @4532h  270000c0  w2:0c000h
    jae loc_453ah                           ; @4536h  01c3  x:loc_453ah

    addi r14, 2                             ; @4538h  4ed8

[...]

// Xrefs: 4506h
loc_4812h:
    ret                                     ; @4812h  97cf  endsub sub_4502h
```

En premier lieu, les bits de condition sont testés par rapport à la valeur `0xff`. En cas d'égalité, un saut est effectué à l'adresse `0x4812` pour effectuer un `ret` qui déclenche la sortie de la fonction de décodage des instructions : le programme de la machine virtuelle est donc terminé.

Dans le cas contraire, l'exécution continue normalement. Les bits de condition sont copiés dans le registre `r11` et les drapeaux de condition (`0x4144`) de la machine virtuelle sont copiés dans le registre `r12`. Ensuite, selon la valeur du registre `r14` (qui contient la valeur du champ `C`), les deux registres `r11` et `r12` sont (ou ne sont pas) décalés de 4 bits à droite pour ne sélectionner que les bits à tester.

Les deux registres `r11` et `r12` sont masqués avec la valeur `0xf` pour ne garder que les 4 bits de poids faible. La valeur `0xc` est ajoutée au registre `r11`, la valeur de ce dernier étant écrite en mémoire à l'adresse `0x4537`. Cette opération permet de réécrire les bits de condition de l'instruction `jae loc_453A` à l'adresse `0x4536`. Enfin, la valeur des drapeaux du processeur CY16 est copiée dans le registre `r0` pour être ensuite masquée avec la valeur `0xfff0`. Le résultat est ajoutée à `r12` à l'aide d'une instruction `or` puis les drapeaux du processeur sont mis à jour avec la nouvelle valeur ainsi calculée.

Si l'instruction à l'adresse `0x4536` est exécutée, alors le test conditionnel (de la machine virtuelle) est vérifié. Sinon, le test échoue et la valeur du registre `r14` est incrémentée de 2. Cette valeur est alors testée plus loin dans le code pour savoir si l'instruction de la machine virtuelle doit être exécutée ou non, par exemple à l'adresse `0x45a` :

```
loc_45aeh:
    mov r12, r14                            ; @45aeh  8c03
    and r12, 2                              ; @45b0h  cc670200
    jnz loc_45bah                           ; @45b4h  02c1  x:loc_45bah

    call sub_435eh                          ; @45b6h  9faf5e43  x:sub_435eh

// Xrefs: 45b4h
loc_45bah:
    jmp loc_44dah                           ; @45bah  9fcfda44  x:loc_44dah
```

Si le registre `r14` n'a pas été incrémenté de 2 lors de l'évaluation du test, alors le saut à l'adresse `0x45b4` est effectué, l'appel à la fonction `sub_453e` n'est pas réalisé. Cette fonction

est responsable de la sauvegarde du résultat de l'instruction à l'adresse mémoire précisée par l'opérande de destination.

Dans tous les cas, un saut est finalement effectué à l'adresse `0x44da` pour décoder et exécuter la prochaine instruction de la machine virtuelle.

La prise en compte du résultat du test conditionnel arrive assez tardivement dans le traitement de l'instruction en cours. En effet, la machine virtuelle travaille sur un flux de données : pour que ce flux soit correctement positionné lors du traitement d'une nouvelle instruction, il est nécessaire que l'instruction précédente demande à décoder toutes ses opérandes. Cela explique qu'en cas d'échec du test conditionnel, le saut à l'adresse `0x44da` n'est pas réalisé immédiatement mais après le décodage des opérandes de l'instruction en cours d'évaluation.

Mise à jour des drapeaux de condition

La function `sub_4814` à l'adresse `0x4814` est chargée de mettre à jour les drapeaux de condition de la machine virtuelle à partir des drapeaux du processeur CY16. A l'entrée de cette fonction, le registre `r6` contient la valeur du champ `UF`, le registre `r14` contient le résultat de l'évaluation des bits de condition (telle que décrite dans le paragraphe précédent).

gestion des drapeaux de condition

```
// Xrefs: 4598h 45cah 45fah 4604h
sub_4814h:
    int 49h                                 ; @4814h  49af
    mov r0, word ptr [0c000h]               ; @4816h  c00900c0  r2:0c000h
    and r0, 0fh                             ; @481ah  c0670f00
    and r6, r6                              ; @481eh  8661
    jz loc_4844h                            ; @4820h  11c0  x:loc_4844h

    mov r12, r14                            ; @4822h  8c03
    and r12, 2                              ; @4824h  cc670200
    jnz loc_4844h                           ; @4828h  0dc1  x:loc_4844h

    mov r12, word ptr [word_4144h]          ; @482ah  cc094441  r2:word_4144h
    and r14, r14                            ; @482eh  8e63
    jz loc_4838h                            ; @4830h  03c0  x:loc_4838h

    and r12, 0f0h                           ; @4832h  cc67f000
    jmp loc_483eh                           ; @4836h  03cf  x:loc_483eh

// Xrefs: 4830h
loc_4838h:
    shl r0, 4                               ; @4838h  c0d2
    and r12, 0fh                            ; @483ah  cc670f00

// Xrefs: 4836h
loc_483eh:
    or r12, r0                              ; @483eh  0c80
    mov word ptr [word_4144h], r12          ; @4840h  27034441  w2:word_4144h

// Xrefs: 4820h 4828h
loc_4844h:
    int 4ah                                 ; @4844h  4aaf
    ret
```

Cet extrait de code commence par charger dans le registre `r0` la valeur des drapeaux de condition du processeur CY16 et masque cette valeur avec `0x0f` pour ne garder que les 4 premiers bits de poids faible.

Si la valeur du champ `UF` est égale à 0, alors le code saute directement à l'adresse `0x4844`. De façon similaire, si la valeur du registre `r14` masquée avec 2 est nulle, alors cela signifie que les bits de condition de l'instruction n'ont pas été vérifiés : le code sort de la fonction.

Dans le cas contraire, la valeur actuelle des drapeaux de la machine virtuelle est stockée dans le registre `r12`. Selon la valeur du registre `r14` (qui contient la valeur du champ `C` de l'instruction), les opérations suivante sont réalisées :

- si `r14` est nul (`C=0`), alors `r0` est décalé de 4 à gauche et `r12` est masqué avec `0x0f` ;
- si `r14` est non-nul (`C=1`), alors `r12` est simplement masqué avec `0xf0`.

Finalement, les valeurs des registres `r12` et `r0` sont combinées à l'aide de l'instruction `or` pour former la nouvelle valeur des drapeaux de condition de la machine virtuelle.

4.1.4 Synthèse des adresses utilisées par la machine virtuelle

Le tableau 5 présente les adresses mémoires utilisées par la machine virtuelle pour stocker le résultat du décodage des opérandes.

Nom symbolique	Adresse	Signification
SRC_VALUE	0x4126	valeur de l'opérande source
SRC_REG	0x4128	registre référencé par l'opérande source
SRC_MEM_ADDR	0x412a	adresse mémoire référencée par l'opérande source
SRC_OBF_VALUE	0x412c	valeur utilisée pour l'obfuscation de l'opérande source
SRC_SIZE	0x412e	taille (octet ou mot) de l'opérande source
SRC_TYPE	0x4130	type de l'opérande source
DST_VALUE	0x4134	valeur de l'opérande destination
DST_REG	0x4136	registre référencé par l'opérande destination
DST_MEM_ADDR	0x4138	adresse mémoire référencée par l'opérande destination
DST_OBF_VALUE	0x413a	valeur utilisée pour l'obfuscation de l'opérande destination
DST_SIZE	0x413c	taille (octet ou mot) de l'opérande destination
DST_TYPE	0x413e	type de l'opérande destination

Table 5 – Adresses utilisées pour la gestion des opérandes

Le tableau 6 présente les adresses mémoires utilisées par la machine virtuelle pour stocker la position dans le programme, l'état des drapeaux de condition ainsi que les valeurs des registres.

La valeur du registre x est stockée à l'adresse `REG_BASE + 2 * x`.

Nom symbolique	Adresse	Signification
CURRENT_BYTE	0x411c	position dans le programme (en octets)
CURRENT_BITS	0x411e	position dans le programme (décalage en bits par rapport à la position en octets)
FLAGS	0x4144	valeur des drapeaux de condition
REG_BASE	0x40fe	adresse de base des registres

Table 6 – Adresses utilisées par les registres et les drapeaux de condition

4.2 Développement de l'outillage nécessaire

Une fois la spécification de la machine virtuelle comprise, il m'a semblé nécessaire de disposer de deux outils pour continuer l'analyse :

- un désassembleur pour obtenir depuis les trois « layers » les instructions exécutées par la machine ;
- un programme capable de traduire un ensemble d'instructions de la machine virtuelle vers un programme en C.

La démarche globale est de pouvoir générer le code assembleur correspondant à un « layer », de générer le code C depuis le code assembleur afin de pouvoir mettre en place une attaque par force brute.

4.2.1 Développement d'un désassembleur

Fort de la spécification de la machine virtuelle, il m'a alors été possible de développer un désassembleur pour tenter de comprendre la logique programmée dans les trois « layers », le but étant bien évidemment d'arriver à déterminer la clé permettant de passer chaque « layer ».

Le désassembleur est développé en Ruby et s'exécute en ligne de commande en spécifiant un fichier en entrée (le binaire à désassembler) et un fichier de sortie.

```
$ ./vmdisas.rb -d -i vm/layer1/layer1 -o layer1.asm
```

Le fichier résultant contient alors la liste d'instructions du programme :

```
[0000:0][c=0][uf=0] mov r12, w[0xa000]
[0005:1][c=0][uf=0] mov r3, w[0xa002]
[0010:2][c=0][uf=0] mov r0, r12
[0013:5][c=0][uf=0] shift right r0, 8
[0017:2][c=0][uf=0] mov r13, r0
[0020:5][c=0][uf=0] and r13, r12
[0024:0][c=0][uf=0] mov w[0xe3b4], r0
[0029:1][c=0][uf=0] or w[0xe3b4], r12
[0034:2][c=0][uf=0] neg w[0xe3b4]
[0038:4][c=0][uf=0] or r13, w[0xe3b4]
[0043:5][c=0][uf=0] neg r13
[0046:1][c=0][uf=0] mov r6, r13
[0049:4][c=0][uf=0] mov r5, r6
[0052:7][c=0][uf=0] mov w[0xc7f2], 0x9577
```

```
[0059:4][c=0][uf=0] neg w[0xc7f2]
[0063:6][c=0][uf=0] or r5, w[0xc7f2]
[0068:7][c=0][uf=0] neg r5
[0071:3][c=0][uf=0] mov w[0xc882], r6
[0076:4][c=0][uf=0] mov w[0xf400], 0x9577
[0083:1][c=0][uf=0] neg w[0xf400]
[..]
```

Le code source du désassembleur est disponible en annexe : A.3.1. Au niveau de l'implémentation, le programme définit 3 classes principales :

- la classe `Instruction` représentant une instruction du programme. Une classe fille d'`Instruction` est définie pour chaque type d'instruction. Chacune de ces classes doit implémenter la méthode `to_s` qui donne le résultat de l'instruction désassemblée ;
- la classe `Operand` représentant une opérande d'une instruction. Une classe fille est définie pour chaque type d'opérande. Chacune de ces classes doit implémenter deux méthodes, `to_dest` qui représente l'opérande quand elle sert de destination et `to_src` qui représente l'opérande quand elle sert de source ;
- la classe `Decoder`, chargée du décodage des données binaires et d'instancier les objets de type `Instruction` et `Operand`.

4.2.2 Développement d'un compilateur

Pour résoudre les trois « layers », deux méthodes sont envisageables :

- partir du fichier source résultant du désassemblage, analyser l'assembleur produit pour comprendre la logique de vérification des données en entrée puis inverser cette logique pour arriver à retrouver les données attendues ;
- traduire l'assembleur produit dans un autre langage (par exemple en C) pour obtenir un nouveau programme dont la logique est conforme à celle du « layer » désassemblé puis réaliser une attaque par force brute sur les données en entrées jusqu'à trouver les valeurs attendues.

Dans un premier temps, j'ai adopté la première approche pour tenter de résoudre le `layer1`. Cette méthode ne s'est pas montrée concluante, les équations booléennes dérivées des entrées devenant rapidement très complexes [8]. L'espace des données en entrée étant relativement faible (32 bits pour le « layer » 1, 16 bits pour les autres), une attaque par force brute sur une réimplémentation en C de chaque couche me semblait réalisable.

Pour simplifier ce processus de réimplémentation (et éviter les erreurs de retranscription de l'assembleur vers C), j'ai décidé d'ajouter au désassembleur précédemment développé une fonctionnalité de traduction d'une instruction vers l'équivalent exprimé en C. Pour cela, chaque type d'instruction doit implémenter la méthode `to_c`. Un « template » ERB [9] définit le corps du programme C. Le code source du programme est disponible en annexe : A.3.1.

L'option `-h` en ligne de commande permet de préciser quelles sont les options attendues par le programme :

8. un solver SAT aurait pu se révéler utile à ce point
9. http://ruby-doc.org/stdlib-1.9.3/libdoc/erb/rdoc/ERB.html

```
$ vmdisas.rb -h
Usage: vmdisas [options]
    -h, --help                       show help
    -i, --input FILE                 Input file
    -o, --output FILE                Output file
    -d, --disas                      Disassemble specified input file
    -c, --compile                    Disassemble then compile specified input file
```

Le code source C produit définit une fonction principale `f`. Avant l'appel à cette fonction, l'extrait de clé à tester (deux mots de 16 bits) doit être écrit en mémoire aux adresses `0xa000` et `0xa002`. La fonction retourne 0 si les valeurs définies à ces adresses ne permettent pas de vérifier une certaine condition (à définir manuellement dans le code) ou 1 si les valeurs définies sont celles attendues.

4.3 Résolution des trois couches

4.3.1 Première couche

Comme présenté au paragraphe 4.2.1, il est possible à ce stade de désassembler le fichier binaire `layer1` qui a été directement extrait du programme `ssticrypt`. Le fichier source assembleur produit contient 306 instructions.

Il est alors possible de générer automatiquement la retranscription en C :

```
$ md5sum layer1
683ed5bc1073fddda05fc7005df4d7fd  layer1
$ vmdisas.rb -i layer1 -o bflayer1.orig.c -c
$ gcc  -Wall -Wno-unused -o bflayer1 bflayer1.orig.c
```

Le fichier produit doit être modifié pour préciser quelle est la condition que l'on cherche à atteindre. Pour connaître celle-ci précisément, il faut se référer au résultat de la rétro-conception du binaire `ssticrypt`, et précisément à la fonction `vicpwn_check`. Dans le cas de la première couche, la condition à vérifier est la suivante : le mot en mémoire stocké à l'adresse `0xa000` doit être différent des quatre premiers octets de la clé.

L'exécution des deux instructions ci-dessous permettent de vérifier cette condition.

```
[1316:4][c=0][uf=0] mov z w[0xa000], w[0xc1a4]
[...]
[1369:1][c=0][uf=0] mov z w[0xa002], w[0xdfd0]
```

Il faut donc modifier le code C produit en conséquence ainsi que commenter le saut vers le début de programme (qui correspond à une fin d'exécution pour la machine virtuelle mais qui provoque une boucle infinie dans notre cas).

```
--- bflayer1.orig.c     2012-05-08 13:50:44.258539908 +0200
+++ bflayer1.c  2012-05-08 14:01:00.154532064 +0200
@@ -682,8 +682,10 @@
        // [1311:1][c=1][uf=1] shift left w[0xfd20], 1
```

```
        shiftw(LEFT, COND1, DO_UF, 0xfd20, 1);
        // [1316:4][c=0][uf=0] mov z w[0xa000], w[0xc1a4]
-       if (z0 == 1)
+       if (z0 == 1) {
+               ret = 1;
                sw(0xa000, rw(0xc1a4));
+       }
        // [1323:3][c=0][uf=0] mov z w[0xdfd0], r6
        if (z0 == 1)
                sw(0xdfd0, r6);
@@ -812,7 +814,7 @@
        // [1614:0][c=0][uf=0] mov r0, 0x00aa
        r0 = 0x00aa;
        // [1618:7][c=0][uf=0] jmp 0000:0
-       goto loc_0000_0;
+       //goto loc_0000_0;
        // [1624:0][c=1][uf=0] and r0, r0
        r0 = and(COND1, NO_UF, r0, r0);
```

Le résultat est obtenu en un peu moins de six minutes :

```
$ time ./bflayer1
e5df 94e3
./bflayer1  342,65s user 0,10s system 99% cpu 5:43,05 total
```

Il est intéressant de remarquer que tout l'espace possible a été couvert et qu'un seul candidat a été trouvé.

4.3.2 Seconde couche

La démarche permettant d'identifier l'extrait de clé correcte pour valider la seconde couche reste globalement très similaire à ce qui été réalisé pour la première couche. Néanmoins, quelques différences sont présentes :

- les données correspondant au « layer2 » dans le binaire `ssticrypt` sont obfusquées et le processus de dé-obfuscation dépend du résultat de l'évaluation de la première couche (cf fonction `vic_pwncheck`) ;
- la seconde couche utilise massivement les opérandes dite `obfusquées` (cf. 4.1.2) alors que ces dernières n'étaient pas du tout présentes dans la première couche ;
- les octets 3 à 6 de la clé sont copiés aux adresses `0xa000` et `0xa002` ;
- la fonction `set_my_key` du binaire `ssticrypt` copie le contenu du tampon de données `blob` à l'adresse `0xa000 + 16`.

Pour extraire les données de la seconde couche, il est possible de modifier la fonction `vicpwn_check` du fichier `client.c` pour enregistrer dans un fichier temporaire les données décodées dans le cas où la validation de la première couche est positive.

Comme pour la première couche, il est alors possible de désassembler les données décodées :

```
$ md5sum layer2.decoded
64bb1d6b3afbd30d29e8a43c8a335461  layer2.decoded
$ vmdisas.rb -i layer2.decoded -o layer2.decoded.asm -d
```

```
$ vmdisas.rb -i layer2.decoded -o bflayer2.orig.c -c
```

Le fichier `layer2.decoded.asm` contient 519 instructions. La compilation du fichier `bflayer2.orig.c` émets quelques avertissements :

```
bflayer2.orig.c: In function 'f':
bflayer2.orig.c:152:2: attention : 'r10' is used uninitialized in this function [-Wuninitialized]
bflayer2.orig.c:160:2: attention : 'r6' is used uninitialized in this function [-Wuninitialized]
bflayer2.orig.c:182:2: attention : 'r12' is used uninitialized in this function [-Wuninitialized]
bflayer2.orig.c:206:2: attention : 'r2' is used uninitialized in this function [-Wuninitialized]
bflayer2.orig.c:208:2: attention : 'r9' is used uninitialized in this function [-Wuninitialized]
bflayer2.orig.c:284:2: attention : 'r13' is used uninitialized in this function [-Wuninitialized]
```

La pertinence de ces avertissements est confirmée par l'analyse des premières instructions :

```
[0000:0][c=0][uf=0] mov r4, 0x0000
[0004:7][c=0][uf=0] mov obf(0xca5c, r4, 0xf), r11
[0011:0][c=0][uf=0] or obf(0xca5c, r4, 0xf), r10
[0017:1][c=0][uf=0] mov obf(0xca5c, r4, 0xf), 0x56de
[0024:6][c=0][uf=0] mov obf(0xe4ac, r4, 0x24), 0x40b7
[0032:3][c=0][uf=0] mov r11, w[0xa000]
[0037:4][c=0][uf=0] mov obf(0xeafe, r4, 0x29), r6
[...]
```

Les registres `r6`, `r10` et `r11` sont effectivement utilisés sans initialisation. En réalisé, la valeur de ces registres est déterminée par l'exécution du la première couche. Il faut alors analyser une trace d'exécution de la machine virtuelle (avec le programme `test_key` A.2.3) pour connaître les valeurs de ces registres, par exemple dans le cas de `r6` :

```
[+] server.c:do_instruction:1423
-> PC = (37, 4), cond = 0, cond_bits = f:5, flags = 1:1, ins = mov, uf = 0
[+] server.c:decode_operands:1031 OBF OP: SRC_MEM_ADDR = eafe, r4 = 0, v = 29
[+] server.c:decode_operands:985 REG r6 = 0x6b14
[+] server.c:do_instruction:1423
```

Le registre `r6` doit donc être initialisé à la valeur `0x6b14`. L'opération doit être répétée pour initialiser tous les registres correctement.

Pour valider la clé, la fonction `vicpwn_check` vérifie que la valeur à l'adresse `0xa000 + 16` est différente de `0xffff`. L'instruction ci-dessous affecte précisément cette valeur en mémoire :

```
[3161:0][c = 1][uf = 0] mov nz w[0x9fc0] + r8, 0xffff
```

Il faut donc partir du principe que la clé testée est bonne, sauf dans le cas où l'instruction précédente est exécutée. Les modifications à apporter sur le code C généré sont présentées ci-dessous :

```
--- bflayer2.orig.c     2012-05-08 14:39:04.810502967 +0200
+++ bflayer2.c  2012-05-08 14:54:36.934491098 +0200
@@ -142,6 +142,9 @@

        c0 = 0; c1 = 0; z0 = 0; z1 = 0;
```

```
+       r0 = 0;       r1 = 0; r2 = 0; r3 = 0; r4 = 0; r5 = 0;
+       r6 = 0x6b14; r7 = 0; r8 = 0; r9 = 0; r10 = 0; r11 = 0;
+       r12 = 0xe5df; r13 = 0; ret = 1;

 loc_0000_0:
        // [0000:0][c=0][uf=0] mov r4, 0x0000
@@ -1042,8 +1045,10 @@
        // [3152:1][c=1][uf=1] and obf(0xfeda, r4, 0x35), obf(0xfeda, r4, 0x35)
        ando(COND1, DO_UF, 0xfeda, r4, 0x35, rwo(0xfeda, r4, 0x35));
        // [3161:0][c=1][uf=0] mov nz w[0x9fc0 + r8], 0xffff
-       if (z1 == 0)
+       if (z1 == 0) {
+               ret = 0;
                sw(0x9fc0 + r8, 0xffff);
+       }

 loc_3168_1:
        // [3168:1][c=0][uf=0] mov obf(0xd7ac, r4, 0x21), obf(0xdf44, r4, 0x37)
@@ -1254,32 +1259,45 @@
        // [3797:6][c=0][uf=0] mov r0, 0x00aa
        r0 = 0x00aa;
        // [3802:5][c=0][uf=0] jmp 0000:0
-       goto loc_0000_0;
+       //goto loc_0000_0;
        // [3807:6][c=1][uf=1] and r1, r0
        r1 = and(COND1, DO_UF, r1, r0);

        return ret;
 }

-void do_init(void) {
-       /* TODO : code me ! */
-}
-
 int main(int argc, char **argv) {
        uint32_t k1, k2;
+       char *addr;
+       int fd;
+       struct stat sb;
+
+       if (argc != 2) {
+           fprintf(stderr, "usage: %s <path to blob>\n", argv[0]);
+           exit(EXIT_FAILURE);
+       }

        m = calloc(MEM_SIZE, 1);

-       do_init();
-
-       for (k1 = 0; k1 < 65536; k1++) {
-               for (k2 = 0; k2 < 65536; k2++) {
-                       sw(0xa000, k1);
-                       sw(0xa002, k2);
-                       if (f()) {
-                               printf("%x %x\n", k1, k2);
-                       }
-               }
+       fd = open(argv[1], O_RDONLY);
+       if (fd == -1)
+           handle_error("open");
```

```
+
+       if (fstat(fd, &sb) == -1)
+           handle_error("fstat");
+
+       addr = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
+       if (addr == MAP_FAILED)
+           handle_error("mmap");
+       close(fd);
+
+       for (k2 = 0; k2 < 65536; k2++) {
+           sw(0xa000, 0x94e3);
+           sw(0xa002, k2);
+           memcpy(m + 0xa000 + 16, addr, sb.st_size);
+           if (f()) {
+               printf("%x\n",k2);
+           }
        }
        exit(EXIT_SUCCESS);
 }
```

Le résultat est obtenu en cinq secondes :

```
$ time ./bflayer2 blob
f63d
./bflayer2 blob  5,42s user 0,00s system 99% cpu 5,428 total
```

4.3.3 Troisième couche

De façon similaire à la seconde couche, les données de la troisième couche dans le binaire `ssticrypt` sont chiffrées en fonction du résultat de la couche précédente. Contrairement aux données de la seconde couche qui étaient protégées par une routine d'obfuscation, un chiffrement RC4 est utilisé pour la troisième couche. L'évaluation de la seconde couche sur la clé attendue permet d'initialiser la table RC4 correcte. Il est donc possible de modifier le code de la fonction `vicpwn_check` du fichier `client.c` pour enregistrer dans un fichier les données déchiffrées de la troisième couche après validation de la couche précédente.

Pour le reste, la démarche est identique à celle adoptée précédemment. Le code déchiffré est désassemblé puis traduit en C :

```
$ md5sum layer3.decoded
$ vmdisas.rb -i layer3.decoded -o layer3.decoded.asm -d
$ vmdisas.rb -i layer3.decoded -o bflayer3.orig.c -c
```

Comme précédemment, la compilation du fichier C produit des avertissements :

```
$ gcc -Wall -o bflayer3 bflayer3.orig.c
bflayer3.orig.c: In function 'f':
bflayer3.orig.c:154:2: attention : 'r2' is used uninitialized in this function [-Wuninitialized]
bflayer3.orig.c:174:2: attention : 'r4' is used uninitialized in this function [-Wuninitialized]
bflayer3.orig.c:224:2: attention : 'r8' is used uninitialized in this function [-Wuninitialized]
bflayer3.orig.c:232:2: attention : 'r11' is used uninitialized in this function [-Wuninitialized]
bflayer3.orig.c:434:2: attention : 'r3' is used uninitialized in this function [-Wuninitialized]
```

```
bflayer3.orig.c:436:2: attention : 'r5' is used uninitialized in this function [-Wuninitialized]
bflayer3.orig.c:599:2: attention : 'r1' is used uninitialized in this function [-Wuninitialized]
```

Il faut alors étudier une trace d'exécution de la machine virtuelle pour identifier les valeurs d'initialisation des registres lors du début du traitement de la troisième couche.

Le test effectué par la fonction `vicpwn_check` à la sortie de la troisième couche est de comparer la chaîne de caractères à l'adresse `0xa000 + 16` avec `V29vdCAhISBTbWVsbHMgZ29vZCA6KQ==` (`Woot!! Smells good :)` codé en base 64. Avant chaque test de clé, le contenu du buffer `blah` doit être copié à l'adresse `0xa000 + 16`.

Les modifications à apporter sur le code C généré sont présentées ci-dessous :

```
--- bflayer3.orig.c     2012-05-08 15:11:14.206478397 +0200
+++ bflayer3.c  2012-05-08 15:17:13.834473818 +0200
@@ -141,7 +141,9 @@
        uint16_t r8, r9, r10, r11, r12, r13, r14, r15;

        c0 = 0; c1 = 0; z0 = 0; z1 = 0;
-
+
+       r7 = 0; r9 = 0; r2 = 0; r0 = 0xaa; r13 = 0;
+       r4 = 0; r8 = 0x14e; r11 = 0x9cc3;

 loc_0000_0:
        // [0000:0][c=0][uf=0] mov r13, r7
@@ -692,32 +694,46 @@
        // [1750:4][c=0][uf=0] mov r0, 0x00aa
        r0 = 0x00aa;
        // [1755:3][c=0][uf=0] jmp 0000:0
-       goto loc_0000_0;
+       //goto loc_0000_0;
        // [1760:4][c=1][uf=0] and r0, r0
        r0 = and(COND1, NO_UF, r0, r0);

        return ret;
 }

-void do_init(void) {
-       /* TODO : code me ! */
-}
-
 int main(int argc, char **argv) {
        uint32_t k1, k2;
+       char *addr;
+       int fd;
+       struct stat sb;
+
+       if (argc != 2) {
+           fprintf(stderr, "usage: %s <path to blah>\n", argv[0]);
+           exit(EXIT_FAILURE);
+       }

-       m = calloc(MEM_SIZE, 1);
+       fd = open(argv[1], O_RDONLY);
+       if (fd == -1)
+           handle_error("open");
```

```
+
+       if (fstat(fd, &sb) == -1)
+           handle_error("fstat");
+
+       addr = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
+       if (addr == MAP_FAILED)
+           handle_error("mmap");
+       close(fd);

-       do_init();
+       m = calloc(MEM_SIZE, 1);

-       for (k1 = 0; k1 < 65536; k1++) {
-               for (k2 = 0; k2 < 65536; k2++) {
-                       sw(0xa000, k1);
-                       sw(0xa002, k2);
-                       if (f()) {
-                               printf("%x %x\n", k1, k2);
-                       }
-               }
+       for (k2 = 0; k2 < 65536; k2++) {
+           memcpy(m + 0xa000 + 16, addr, sb.st_size);
+           sw(0xa000, 0xf63d);
+           sw(0xa002, k2);
+           f();
+           if (!strncmp((char *) m + 0xa000 + 16, "V29vdCAhISBTbWVsbHMgZ29vZCA6KQ==", 0x20)) {
+               printf("%x\n", k2);
+           }
        }
        exit(EXIT_SUCCESS);
 }
```

Le résultat est obtenu quasi instantanément :

```
$ time ./bflayer3 blah
8937
./bflayer3 blah  0,28s user 0,00s system 98% cpu 0,287 total
```

4.4 Conclusion

Les attaques par force brute menées sur les trois couches permettent d'obtenir la seconde partie de la clée recherchée :

- couche 1 : `e5df 94e3` ;
- couche 2 : `f63d` ;
- couche 3 : `8937`.

En combinant le début de la clé extrait de l'implémentation DES « Whitebox », on obtient alors la clé complète : `fd4185ff66a94afd / e5df94e3f63d8937`. Il est donc possible de passer à la suite du challenge.

5 Déchiffrement du fichier `secret`

5.1 Premier essai de déchiffrement

Une fois la clé obtenue dans son intégralité, il est alors possible de tenter de déchiffrer le fichier `secret`. Pour être certain de ne pas avoir fait d'erreur dans l'implémentation de la routine de déchiffrement à partir de l'assembleur, il m'a semblé judicieux d'utiliser le binaire `ssticrypt` original pour effectuer cette opération. La vérification de la clé pouvant se révéler assez longue, il est possible de contourner celle-ci en exécutant le programme avec GDB sous environnement qemu et en posant un point d'arrêt lors du test du mode d'opération demandé (la vérification de la clé n'étant effectuée que pour un déchiffrement et non un chiffrement de données).

Ce test est effectué dans le binaire `ssticrypt` à l'adresse `0x4025c4`. En particulier, si le registre `v0` n'est pas nul à l'adresse `0x4025cc`, alors les deux appels à `check_key` ne sont pas réalisés.

```
_____________________ test du mode d'opération (chiffrement / déchiffrement) _____________________
// Xrefs: 402560h 40258ch
loc_4025c4h:
    lw $v0, 20h($fp)            ; @4025c4h  8fc20020
    nop                         ; @4025c8h  00000000
    bnz $v0, loc_402604h        ; @4025cch  1440000d  x:loc_402604h
    nop                         ; @4025d0h  00000000

    addiu $v0, $fp, 58h         ; @4025d4h  27c20058
    addu $a0, $v0, $zero        ; @4025d8h  00402021
    li $a1, 1                   ; @4025dch  24050001
    jal check_key               ; @4025e0h  0c1007df  x:check_key
    nop                         ; @4025e4h  00000000
    lw $gp, 10h($fp)            ; @4025e8h  8fdc0010
    addiu $v0, $fp, 6ch         ; @4025ech  27c2006c
    addu $a0, $v0, $zero        ; @4025f0h  00402021
    li $a1, 2                   ; @4025f4h  24050002
    jal check_key               ; @4025f8h  0c1007df  x:check_key
    nop                         ; @4025fch  00000000
    lw $gp, 10h($fp)            ; @402600h  8fdc0010
```

Pour déchiffrer le fichier `secret`, il suffit donc de lancer le binaire à l'aide de GDB, de poser un point d'arrêt à l'adresse `0x4025cc` et de modifier la valeur du registre `v0`.

```
$ gdb -q ssticrypt
(gdb) break *0x4025cc
Breakpoint 1 at 0x4025cc
(gdb) run -d fd4185ff66a94afde5df94e3f63d8937 secret
Starting program: /home/user/ssticrypt -d fd4185ff66a94afde5df94e3f63d8937 secret
--> SSTICRYPT <--
Warning: MD5 mismatch for container
Using keys fd4185ff66a94afd / e5df94e3f63d8937 ...

Breakpoint 1, 0x004025cc in main ()
Current language:  auto; currently asm
(gdb) p $v0
$1 = 0
(gdb) set $v0 = 1
(gdb) cont
```

```
Continuing.

Program exited normally.
```

Malgré l'avertissement « `Warning: MD5 mismatch for container` », un fichier `secret.dec` est créé. J'ai donc ensuite procédé à l'implémentation (cf. fichier `decrypt.c` A.4) en C de l'algorithme de déchiffrement pour ensuite pouvoir comparer les empreintes MD5 des fichiers obtenus avec ce programme et avec `ssticrypt` et ainsi valider la correction de mon implémentation.

Le fichier `secret.dec` se révèle être un système de fichiers `ext2` qu'il est possible de monter. Cependant, la commande `ls` sur la racine retourne une erreur d'entrée / sortie.

```
$ file secret.dec
secret.dec: Linux rev 1.0 ext2 filesystem data, UUID=ace27cef-9d4-4d79-ad64-42597535b42e
$ md5sum secret.dec
9a5976264e9d6d13848a83bd513611b5  secret.dec
$ sudo mount -o loop secret.dec /mnt/loop
$ ls /mnt/loop
ls: reading directory /mnt/loop: Input/output error
```

Il semble donc nécessaire de reconstruire le fichier chiffré `secret` pour que le test sur l'empreinte MD5 des données soit vérifié. L'empreinte correcte est obtenue avec la ligne de commande ci-dessous :

```
$ ruby -e 'p open("secret").read(16).unpack("H*")'
["b84db9ec23524e4e557703fb55dfc083"]
```

On peut aisément constater que l'empreinte MD5 des données du fichier `secret` actuel n'est pas correcte.

```
$ dd if=secret bs=16 skip=1 2>/dev/null| md5sum -
449253c03e5ab7396173e36999f552f0  -
```

Il va donc falloir trouver un moyen de reconstituer les données du fichier pour obtenir l'empreinte MD5 correcte.

5.2 Reconstitution du fichier `secret`

L'examen du fichier `secret` à l'aide commande `hexdump` permet de s'apercevoir très rapidement que les données sont invalides à partir d'un certain décalage. En effet, tous les octets après la position `0x3000` (`= 12288`) sont nuls, ce qui est plutôt suspect pour un fichier supposément chiffré.

```
$ hexdump -C secret
[...]
00002fe0  af 26 b1 cb d3 84 19 00  df 75 6f 5f 61 89 c5 fc  |.&.......uo_a...|
00002ff0  c3 28 81 e9 39 3d 83 a4  1c 3d 37 27 e7 4d 9d 9d  |.(..9=...=7'.M..|
00003000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00100010
```

L'examen cette fois-ci du fichier `secret.dec` permet de confirmer ce problème à la même position. Les données semblent correctement déchiffrées jusqu'à la position `0x2ff0`, le reste des données présente ensuite des caractérisques similaires (forte entropie) à des données aléatoires ou chiffrées.

```
$ hexdump -C secret.dec
[...]
000025f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00002fe0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 c5  |................|
00002ff0  ca 31 61 ad 2c 71 bd 46  23 51 0c 51 a0 10 38 cd  |.1a.,q.F#Q.Q..8.|
00003000  29 44 ee bc 64 d3 22 89  56 ed e1 27 85 a8 d6 3f  |)D..d.".V..'...?|
00003010  58 20 38 ad 60 c1 09 ba  a7 eb 25 6f 65 78 b1 22  |X 8.`.....%oex."|
[...]
```

Le décalage de 16 octets entre les deux positions s'explique par la présence de l'empreinte MD5 au début du fichier `secret`. La taille des blocs du système de fichiers de la partition étant de 4096 octets [10], il est alors possible de conclure que seuls les trois premiers ($12288/4096 = 3$) blocs du fichier `secret` sont valides.

L'utilisation de la commande `debugfs` permet d'obtenir de l'information sur le fichier `secret`.

```
$ debugfs challenge.part
debugfs:  cd /home/sstic
debugfs:  stat secret

Inode: 14   Type: regular    Mode:  0755   Flags: 0x0
Generation: 163417969    Version: 0x00000000
User:     0   Group:     0   Size: 1048592
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 24
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x4f6c3483 -- Fri Mar 23 09:29:55 2012
atime: 0x4f6e53a7 -- Sun Mar 25 00:07:19 2012
mtime: 0x4f6c3483 -- Fri Mar 23 09:29:55 2012
Size of extra inode fields: 0
BLOCKS:
(0-2):26625-26627
TOTAL: 3
```

Au niveau `ext2`, la taille du fichier est de 1048592 octets mais seuls trois blocs (26625 à 26627) sont référencés par l'inode. Il y a donc une incohérence à ce niveau car le fichier doit être stocké dans au moins 257 blocs (arrondi supérieur de 1048592/4096). Il semble alors nécessaire de retrouver les 254 blocs manquants disséminés dans l'image de la partition.

La partition comprenant 261048 blocs [11], il est illusoire de vouloir tester toutes les combinaisons de 257 blocs parmi 261048 pour obtenir l'empreinte MD5 recherchée. Un critère de distinction doit être utilisé pour ne garder que les blocs intéressants.

10. `dumpe2fs challenge.part | grep "Block size"`
11. `dumpe2fs challenge.part | grep "Block count"`

Une heuristique possible est de considérer uniquement les blocs de données présentant une entropie élevée. En effet, le fichier `secret` étant chiffré, l'entropie de ses données est similaire à celle de données aléatoires. Cette heuristique comporte des limites pour deux raisons :

- le dernier bloc utilisé par le fichier `secret` présente une entropie moyenne voire faible (il constitue les $1048592 - 256 * 4096 = 16$ dernier octets du fichier) ;
- tous les fichiers compressés présents sur l'image sont constitués de blocs d'entropie forte.

La fonction C ci-dessous implémente un calcul d'entropie tel que défini par Shannon [12].

fonction de calcul d'entropie

```
double entropy(uint8_t *buf, int size) {
    double ret = 0.0;
    int byte_counts[256];
    int i;

    memset(byte_counts, 0, sizeof(int) * 256);
    for (i = 0; i < size; i++)
        byte_counts[buf[i]]++;

    for (i = 0; i < 256; i++) {
        double p_i = (double) byte_counts[i] / (double) size;
        if (p_i > 0.0)
            ret -= p_i * (log(p_i) / log(2));
    }

    return ret;
}
```

Il suffit alors d'appliquer cette fonction sur l'ensemble des blocs de la partition pour identifier quels sont les candidats potentiels pour la reconstruction du fichier `secret`, en ne gardant que les blocs dont l'entropie est supérieure à 7,8. Le résultat est le suivant :

```
block 20553, entropy = 7.844841
block 26625, entropy = 7.954545
block 26626, entropy = 7.955469
block 26627, entropy = 7.953132
block 26628, entropy = 7.947644
block 26629, entropy = 7.956155
block 26630, entropy = 7.950711
block 26635, entropy = 7.960115
[...]
block 26880, entropy = 7.957481
block 26881, entropy = 7.957886
block 47120, entropy = 7.846456
block 47121, entropy = 7.846456
block 47123, entropy = 7.930853
block 47125, entropy = 7.930853
block 99041, entropy = 7.939410
block 99042, entropy = 7.947699
[...]
block 191900, entropy = 7.950306
[+] found 767 block candidates
```

767 blocs possèdent une entropie supérieure à 7,8. On constate également qu'une série particulière de blocs semble intéressante : celle commençant à 26625 (premier bloc du fichier `secret`)

12. http://fr.wikipedia.org/wiki/Entropie_de_Shannon

et se terminant à 26881. On peut alors essayer de reconstituer le fichier à l'aide de ces blocs.

```
$ dd if=challenge.part of=secret.tmp bs=4096 skip=26625 count=$((26881-26625))
257+0 records in
257+0 records out
1052672 bytes (1.1 MB) copied, 0.00386449 s, 272 MB/s
$ dd if=secret.tmp of=secret.cand bs=1048592 count=1
1+0 records in
1+0 records out
1048592 bytes (1.0 MB) copied, 0.00551943 s, 190 MB/s
$ dd if=secret.cand bs=16 skip=1 2>/dev/null| md5sum -
8409be76a7fb8d393a20b6b829908514  -
```

L'empreinte MD5 du fichier n'est toujours pas correcte, il est néanmoins possible de tenter un déchiffrement.

```
$ ./decrypt fd4185ff66a94afde5df94e3f63d8937 secret.cand secret.cand.dec
```

Le fichier `secret.cand.dec` déchiffré à l'aide du programme `decrypt` A.4 à partir de la reconstitution des blocs précédents présente un certain nombre de chaînes de caractères intéressantes.

```
$ strings -n 6 secret.cand.dec
}agO}agO}agO
}agO}agO}agO
}agO}agO}agO
lost+found
lobster
AVI LIST~
hdrlavih8
strlstrh8
vidsh264
strlstrh8
INFOISFT
MEncoder SVN-r33094-4.5.3
```

On aperçoit notamment la chaîne « lobster » ainsi que des références à H264 et MEncoder. Comme précédemment, il est possible de monter le fichier déchiffré en tant que partition ext2.

```
$ sudo mount -o loop secret.cand.dec /mnt/loop
$ ls /mnt/loop
lobster  lost+found
$ file /mnt/loop/lobster
/mnt/loop/lobster: RIFF (little-endian) data, AVI, 352 x 264, ~15 fps, \
    video: H.264 X.264 or H.264, audio: MPEG-1 Layer 3 (stereo, 44100 Hz)
```

Le fichier `/mnt/loop/lobster` est sans doute la vidéo contenant l'adresse email permettant de valider le challenge. Malheureusement, le fichier semble corrompu :

```
$ mplayer /mnt/loop/lobster
MPlayer svn r34540 (Ubuntu), built with gcc-4.6 (C) 2000-2012 MPlayer Team

Playing /mnt/loop/lobster.
libavformat version 53.21.0 (external)
Mismatching header version 53.19.0
AVI file format detected.
```

```
[aviheader] Video stream found, -vid 0
[aviheader] Audio stream found, -aid 1
AVI: Missing video stream!? Contact the author, it may be a bug :(
libavformat file format detected.
[avi @ 0x7f89afe52940]Could not find codec parameters (Video: h264, 352x264)
```

En concaténant les blocs à partir de 26625, il a donc été possible d'obtenir davantage de données déchiffrées correctement. Pourtant, l'empreinte MD5 n'est pas bonne et le fichier vidéo n'est pas lisible. Deux hypothèses peuvent expliquer pourquoi le fichier `secret.cand` n'est pas bon :

- les blocs sélectionnés ne sont pas les bons ;
- les blocs sélectionnés n'ont pas été ordonnés correctement.

En effet, l'algorithme utilisé pour le chiffrement / déchiffrement des données est RC4 [13]. Cet algorithme travaille sur des flux de données et possède une propriété intéressante : si un seul bit de données est corrompu dans le message chiffré, alors seul un bit du message déchiffré sera affecté. De plus, lors du processus de déchiffrement, les données doivent être traitées exactement dans le même ordre que lors du processus de chiffrement.

Ces deux propriétés permettent de mettre en place l'heuristique suivante pour retrouver l'ordre dans lequel doivent être réarrangés les blocs de forte entropie :

- pour chacune des 257 positions, chaque bloc de forte entropie est testé :
  - le bloc candidat est copié à la bonne position puis est déchiffré,
  - si l'entropie des données déchiffrées (pour le bloc testé) est significativement inférieure à l'entropie du bloc chiffré, alors le bloc candidat est sans doute correctement positionné,
  - le bloc qui est retenu pour la position en cours est celui qui a permis de maximiser la différence d'entropie entre chiffré et déchiffré ;
- à la fin de cette boucle, les blocs ont été correctement ordonnés.

Cette heuristique a été implémentée dans le fichier `findblocks.c` A.4. Le résultat de son exécution est présenté ci-dessous :

```
$ ./findblocks challenge.part
position 000 => block 026625, entropy diff = 7.714120
position 001 => block 026626, entropy diff = 6.943608
position 002 => block 026627, entropy diff = 7.611386
position 003 => block 026628, entropy diff = 7.944363
position 004 => block 026629, entropy diff = 7.952873
position 005 => block 026630, entropy diff = 7.947429
position 006 => block 026631, entropy diff = 7.803239
position 007 => block 026632, entropy diff = 7.942012
position 008 => block 026633, entropy diff = 7.940712
position 009 => block 026634, entropy diff = 4.955540
position 010 => block 026635, entropy diff = 2.071878
position 011 => block 026636, entropy diff = 0.214087
position 012 => block 026638, entropy diff = 1.012545
position 013 => block 026639, entropy diff = 0.162675
position 014 => block 026640, entropy diff = 0.170851
[...]
position 025 => block 026651, entropy diff = 0.120784
position 026 => block 026652, entropy diff = 0.237390
position 027 => block 127331, entropy diff = 0.055969
```

13. http://en.wikipedia.org/wiki/RC4

```
position 028 => block 115392, entropy diff = 0.048782
```

Il est possible de constater que l'heuristique permet d'obtenir le résultat escompté pour les premiers blocs. Cependant, le bloc 127331 est le meilleur candidat pour la position 27 mais la différence d'entropie (0,055969) n'est pas significative. Après investigation, il s'avère que le bloc en position 27 est un bloc de données `H.264`, qui contient donc des données compressées. Dans ce cas, la différence d'entropie entre le bloc chiffré et le bloc `H.264` correspondant est trop faible pour constituer un critère de décision. Il n'est donc pas possible avec cette heuristique de réordonner correctement les blocs de données `H.264`.

De plus, pour la position 12, le bloc 26638 est sélectionné alors que le bloc précédent (position 11) est le bloc 26636 : le bloc 26637 n'est pas utilisé. En regardant le contenu de ce bloc avec un éditeur hexadécimal, il est aisé de comprendre pourquoi.

```
$ dd if=challenge.part bs=4096 skip=26637 count=1 | hexdump -C | head -n 5
00000000  0e 68 00 00 0f 68 00 00  10 68 00 00 11 68 00 00  |.h...h...h...h..|
00000010  12 68 00 00 13 68 00 00  14 68 00 00 15 68 00 00  |.h...h...h...h..|
00000020  16 68 00 00 17 68 00 00  18 68 00 00 19 68 00 00  |.h...h...h...h..|
00000030  1a 68 00 00 1b 68 00 00  1c 68 00 00 1d 68 00 00  |.h...h...h...h..|
00000040  1e 68 00 00 1f 68 00 00  20 68 00 00 21 68 00 00  |.h...h.. h..!h..|
```

L'entropie de ce bloc est visiblement très faible. Intuitivement, on a alors envie de répéter l'opération de reconstituer le fichier `secret` à partir des blocs 26625 à 26882 mais cette fois en excluant le bloc 26637.

```
$ cat <(dd if=challenge.part bs=4096 skip=26625  count=12) \
      <(dd if=challenge.part bs=4096 skip=26638 count=245) > secret.tmp
$ dd if=secret.tmp of=secret.cand bs=1048592 count=1
$ dd if=secret.cand bs=16 skip=1 2>/dev/null| md5sum -
b84db9ec23524e4e557703fb55dfc083  -
```

On retrouve enfin l'empreinte MD5 tant attendue, le fichier est donc cette fois correctement reconstitué.

5.3 Accès à la vidéo

Une fois le fichier correctement reconstitué, il est alors possible de le déchiffrer, de monter la partition ext2 et de visualiser la vidéo `lobster`.

```
$ ./decrypt fd4185ff66a94afde5df94e3f63d8937 secret.cand secret.cand.dec
$ sudo mount -o loop secret.cand.dec /mnt/loop
$ mplayer /mnt/loop/lobster
```

6 Conclusion

Le challenge SSTIC 2012 s'est révélé particulièrement intéressant et fait appel à des compétences techniques variées : rétro-conception, cryptanalyse, analyse d'architecture matérielle, etc.

FIGURE 17 – lobster dog !

J'ai notamment apprécié le fait de devoir travailler sur une architecture peu répandue (CY16), qui a donc nécessitée le développement d'un outillage adapté.

A Code source

A.1 DES white-box

A.1.1 Analyse du byte-code Python

Enregistrer `disas-check.pyc.txt` :

Enregistrer `reverse.py` :

A.2 Analyse de la ROM

A.2.1 Désassemblage de la ROM

Enregistrer `cy16-rom.asm` :

A.2.2 Analyse dynamique

Enregistrer `usb-hook.c` :

Enregistrer `trace-20120425.txt` :

Les fichiers suivants doivent être organisés de façon analogue à ce qui est présenté ci-dessous :

```
$ find . -name "*.rb"
./bin/net.rb
./bin/functions.rb
./bin/cli.rb
./bin/gui.rb
./lib/cy16/alu.rb
./lib/cy16/function.rb
./lib/cy16/loader.rb
./lib/cy16/metaemul.rb
./lib/cy16/debugger.rb
./lib/cy16/gui.rb
./lib/cy16/fakeclient.rb
./lib/testgenerator.rb
./lib/hexdump.rb
```

Enregistrer `bin/net.rb` :

Enregistrer `bin/functions.rb` :

Enregistrer `bin/cli.rb` :

Enregistrer `bin/gui.rb` :

Enregistrer `lib/cy16/alu.rb` :

Enregistrer `lib/cy16/function.rb` :

Enregistrer `lib/cy16/loader.rb` :

Enregistrer `lib/cy16/metaemul.rb` :

Enregistrer `lib/cy16/debugger.rb` :

Enregistrer `lib/cy16/gui.rb` :

Enregistrer `lib/cy16/fakeclient.rb` :

Enregistrer `lib/testgenerator.rb` :

Enregistrer `lib/hexdump.rb` :

Ce code source a été testé avec la version 1.9.3 de Ruby.

A.2.3 Réimplémentation en C de la ROM

Enregistrer `utils.c` :

Enregistrer `utils.h` :

Enregistrer `client.c` :

Enregistrer `client.h` :

Enregistrer `server.c` :

Enregistrer `server.h` :

Enregistrer `netserver.c` :

Enregistrer `test_key.c` :

A.3 Analyse de de la machine virtuelle

A.3.1 Désassembleur / compilateur

Enregistrer `vmdisas.rb` :

A.3.2 Layer 1

Enregistrer `layer1.asm` :

Enregistrer `bflayer1.c` :

A.3.3 Layer 2

Enregistrer `layer2.decoded.asm` :

Enregistrer `bflayer2.c` :

A.3.4 Layer 3

Enregistrer `layer3.decoded.asm` :

Enregistrer `bflayer3.c` :

A.4 Déchiffrement du fichier `secret`

Enregistrer `decrypt.c` :

Enregistrer `findblocks.c` :

Références

[1] Ned Batchelder, *The structure of .pyc files* http://nedbatchelder.com/blog/200804/the_structure_of_pyc_files.html

[2] Python documentation, *marshal - Internal Python object serialization* http://docs.python.org/library/marshal.html

[3] S. Chow, P. Eisen, H. Johnson, P.C. van Oorschot, *A White-Box DES Implementation for DRM Applications*, http://crypto.stanford.edu/DRM2002/whitebox.pdf

[4] Louis Goubin, Jean-Michel Masereel, Michaël Quisquate, *Cryptanalysis of white box DES implementations*, http://eprint.iacr.org/2007/035

[5] Louis Goubin, Jean-Michel Masereel, Michaël Quisquate, *Cryptanalysis of white box DES Implementations (présentation)*, http://www.labri.fr/perso/ly/cryscoe/talk/Goubin.pdf

[6] Wikipedia, *DES supplementary material*, http://en.wikipedia.org/wiki/DES_supplementary_material

[7] Cypress, *CY16 USB Host/Slave Controller/16-Bit RISC Processor Programmers Guide*, http://www.cypress.com/?docID=14345

[8] Cypress, *BIOS User's manual*, http://www.cypress.com/?docID=14346