Solution du challenge SSTIC 2012

Benoît Camredon

Table des matières

Table des figures

1 Introduction

Le challenge SSTIC 2012 consistait à retrouver une adresse e-mail dans l'image d'un disque dur. Un challenge très intéressant car il fait appel à divers domaines de compétence, comme le forensic, la cryptographie et le reverse engeenering. Ci-dessous une liste d'outils *existants* ayant été utilisés :

- Les outils de base disponibles dans une distribution Linux
- *IDA* pour le reverse du code MIPS
- *qemu* pour émuler une architecture MIPS
- *emacs* pour faire du développement d'outils
- *python*
- *gcc*

2 Analyse du fichier d'entrée

Le fichier d'entrée appelé *challenge.txt* n'est en réalité pas un fichier texte mais un fichier compressé avec *gzip*.

```
$ file challenge.txt
challenge.txt: gzip compressed data, was "dump.img",
from Unix, last modified: Fri Mar 23 10:11:37 2012
$ mv challenge.txt challenge.gz
$ gunzip challenge.gz
$ file challenge
challenge: x86 boot sector; partition 1: ID=0x83,
active, starthead 1, startsector 63, 2088387 sectors, code offset 0xb8
```

Une fois le fichier décompressé, la commande *file* nous apprend qu'il s'agit d'une image de disque dur, et qu'elle est à priori bootable. La commande *fdisk* quant à elle nous dit que cette image contient une partition qui commence au 63^{e} secteur et qu'elle serait de type Linux.

```
$ /sbin/fdisk challenge

Command (m for help): p

Disk challenge: 1073 MB, 1073741824 bytes
255 heads, 63 sectors/track, 130 cylinders, total 2097152 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0xcf660900

    Device Boot      Start         End      Blocks   Id  System
challenge1   *          63     2088449     1044193+  83  Linux
```

Comme cette partition commence au 63^{e} secteur et que chaque secteur a une taille de 512 octets, il est possible de monter cette partition à l'offset 32256 [1].

```
$ mkdir disk
$ sudo mount -o loop,offset=32256 challenge disk
```

Une fois l'image montée, on remarque qu'il s'agit d'un système Linux tournant sous debian. Le système de fichier est de type *ext2* et un utilisateur *sstic* existe sur le système. Plusieurs fichiers intéressants se trouvent dans son arborescence.

1. 63*512

```
$ ls -l disk/home/sstic
total 1,4M
-rw-r--r-- 1 root root  871 mars  23 09:51 irc.log
-rwxr-xr-x 1 root root 1,1M mars  23 09:29 secret
-rwxr-xr-x 1 root root  128 mars  23 09:30 ssticrypt
```

Le fichier *irc.log* relate une discussion confidentielle entre *lobster_dog* et *blue_footed_booby*. *lobster_dog* veut protéger ses fichiers de l'infâme *lobster_cat*. Pour cela il les envoie à *blue_footed_booby* qui va les chiffrer avec son système révolutionnaire. Les données sont protégées et *lobster_dog* peut donc les supprimer tranquillement. Le problème est que le disque dur bon marché de *blue_footed_booby* n'est plus tout jeune et rend l'âme. Les données de *lobster_dog* n'existent donc plus que sous forme chiffrées... sur un disque dur agonisant.

Les données en question doivent être situées dans le fichier *secret* du répertoire */home/sstic*. Et la méthode de chiffrement de ces données doit se situer dans le fichier *ssticrypt*.

Le fichier *ssticrypt* est un fichier *ELF 32-Bit* pour architecture *MIPS*.

```
$ file disk/home/sstic/ssticrypt
ssticrypt: ELF 32-bit MSB executable, MIPS, MIPS-I version 1 (SYSV), statically
linked (uses shared libs), stripped
```

L'image doit donc être l'image du disque dur d'une machine *MIPS*. Cependant la taille du fichier *ssticrypt* est suspecte, seulement 128 octets. Si elle s'avère vraie, le reverse de ce binaire devrait être rapide...

```
$ readelf -a 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!
readelf: Error: Unable to read in 0x100 bytes of program headers
```

A priori le fichier *ssticrypt* a été victime de la loi de la tartine de confiture... Le crash du disque dur a eu un impact direct sur ce fichier...

Après quelques lectures sur les rudiments du fonctionnement de l'*ext2* [1], il est temps de reconstruire le fichier *ssticrypt*. Pour cela l'utilitaire *debugfs* est utilisé.

```
$ dd if=challenge of=partition skip=63 bs=512
2097089+0 enregistrements lus
2097089+0 enregistrements écrits
1073709568 octets (1,1 GB) copiés, 5,27244 s, 204 MB/s
$ sudo debugfs partition -R "stat /home/sstic/ssticrypt"
Inode: 19   Type: regular    Mode:  0755   Flags: 0x0
Generation: 163417970    Version: 0x00000000
User:     0   Group:     0   Size: 128
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 616
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x4f6c348c -- Fri Mar 23 09:30:04 2012
atime: 0x4fa78167 -- Mon May  7 10:01:43 2012
mtime: 0x4f6c348c -- Fri Mar 23 09:30:04 2012
Size of extra inode fields: 0
BLOCKS:
(0-11):20480-20491, (IND):20492, (12-75):20493-20556
TOTAL: 77
$ sudo dumpe2fs partition  | grep -i "block size"
dumpe2fs 1.42.2 (9-Apr-2012)
Block size:               4096
```

On apprend que le fichier *ssticrypt* est en réalité composé de 77 blocs, et qu'ils sont contigües ce qui nous arrange vraiment, de plus chaque bloc a une taille de 4096 octets.

En *ext2* les 12 premiers blocs sont des blocs directs, donc des blocs de données. Ensuite l'*inode* contient si besoin un bloc indirect (c'est à dire un bloc qui va pointer sur 256 autres blocs directs), un bloc doublement indirect(qui pointe sur 256 blocs indirects), et enfin un bloc triplement indirect (qui pointe sur 256 blocs doublement indirects). En résumé, certains blocs sont des blocs de données, d'autres sont des blocs contenant des pointeurs soit sur des données soit sur d'autres pointeurs. C'est important pour la restauration de notre fichier afin ne pas considérer des pointeurs comme des données.

Le fichier *ssticrypt* est composé des blocs 20480 à 20556. Les 12 premiers blocs sont des blocs de données, le 13^{e} (bloc 20492) est un bloc indirect et ensuite les blocs 20493 à 20556 sont à nouveau des blocs de données (référencés par le bloc indirect 20492). Il n'y a pas dans ce cas là, de bloc doublement indirect ou triplement indirect. La reconstruction se résume donc à prendre les blocs 20480 à 20491 et les blocs 20493 à 20556.

Remarque : Le bloc 20556 n'est probablement pas à prendre en totalité, cependant la taille du fichier ayant été altérée il est encore trop tôt pour le dire. Quoiqu'il en soit, cela ne sera pas un frein pour la résolution du challenge.

```
$ dd if=partition of=ssticrypt_part1 skip=20480 bs=4096 count=12
12+0 enregistrements lus
12+0 enregistrements écrits
49152 octets (49 kB) copiés, 9,4727e-05 s, 519 MB/s
$ dd if=partition of=ssticrypt_part2 skip=20493 bs=4096 count=64
64+0 enregistrements lus
64+0 enregistrements écrits
262144 octets (262 kB) copiés, 0,00036684 s, 715 MB/s
$ cat ssticrypt_part1 ssticrypt_part2 > ssticrypt
```

Remarque : Une solution plus simple aurait été de corriger directement la taille du fichier dans l'inode avec *debugfs*. Cependant cette méthode permet de mieux comprendre le fonctionnement de l'*ext2*.

```
$ readel -a ssticrypt
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
[...]

  00459ecc -32532(gp) 00402b00 00402b00 FUNC    UND calloc
  00459ed0 -32528(gp) 00000000 00000000 NOTYPE  UND _Jv_RegisterClasses
  00459ed4 -32524(gp) 00000000 00000000 FUNC    UND __gmon_start__
  00459ed8 -32520(gp) 00402af0 00402af0 FUNC    UND signal
  00459edc -32516(gp) 00402ae0 00402ae0 FUNC    UND memcmp
  00459ee0 -32512(gp) 00402990 00402990 FUNC     11 __libc_csu_fini
  00459ee4 -32508(gp) 00402ad0 00402ad0 FUNC    UND open
  00459ee8 -32504(gp) 00402ac0 00402ac0 FUNC    UND sprintf
  00459eec -32500(gp) 00402ab0 00402ab0 FUNC    UND usb_release_interface
```

Le fichier semble cette fois-ci plus complet. Afin de s'en assurer, nous allons essayer de l'exécuter. Comme c'est un binaire pour architecture *MIPS* nous allons utiliser *qemu*. Le kernel *vmlinux* a été récupéré dans */boot* de l'image.

```
$ qemu-system-mips -kernel vmlinux -hda challenge -append "root=/dev/hda1 rw"
-nographic -monitor stdio -serial pty
char device redirected to /dev/pts/1
QEMU 1.0,1 monitor - type 'help' for more information
(qemu)
```

Puis dans un autre terminal :

```
$ screen /dev/pts/1
[...]
[17179572.456000] Kernel panic - not syncing: No init found.
Try passing init= option to kernel.
[...]
```

Aïe, comme par hasard... un *kernel panic* sur le *init*. Ce contre-temps peut être évité en passant *init=/bin/bash* au démarrage, cependant le système sera assez limité. Après analyse du fichier */sbin/init*, on s'aperçoit que comme *ssticrypt* il a été victime du disque dur bon marché (comme par hasard !). La même méthode est donc utilisée pour le restaurer.

Remarque : Avant le redémarrage de l'image, le fichier */etc/shadow* est modifié afin ne pas à avoir rentrer de mot de passe sur le système.

```
$ ./ssticrypt
--> SSTICRYPT <--
usage: ./ssticrypt [-d|-e] <key> <secure container>
        -d: uncrypt
        -e: crypt
```

Le fichier semble fonctionner. Il ne reste plus qu' à comprendre ce qu'il fait.

Remarque : L'image *MIPS* a été modifiée pour installer tous les utilitaires digne de ce nom, comme *gdb*, *objdump*...

2.1 ssticrypt à la loupe

Le fichier *ssticrypt* est un fichier *elf* qui n'est pas strippé. Le désassemblage de ce fichier par *IDA* ne pose aucun problème. Voilà le pseudo code de la fonction principale (avec beaucoup de raccourcis) :

```
Main():
mode = "dechiffrement"
Si len(arg) != 4:
   exit;
Si len(arg[3]) != 32:
       exit;

Si(arg[1]) == "-e":
   mode = "chiffrement"

f = open(arg[3]).read()
i = 0

Si mode == "dechiffrement":
   md5 = f[:16]
       Si md5sum(f[16:]) != md5:
             Warning !
i = 16

buf = f[i:]
key_part1 = arg[2][:16]
key_part2 = arg[2][16:]
Verif_coincoin()

Si mode == "dechiffrement":
   check_key(key_part1,1)
   check_key(key_part2,2)
   Transform_XOR(buf)

buf = RC4(key,buf)

Si mode == "chiffrement":
   Transform_XOR-1(buf)
   buf = md5sum(buf)+buf

EcritBuf(buf,mode)
```

Le programme *ssticrypt* permet donc de chiffrer ou de déchiffrer un fichier avec l'algorithme *RC4* et une clé passée en paramètre de 32 octets (modulo une transformation à base de *XOR*). En cas de chiffrement, un *md5* est inséré au début du fichier et porte sur tout le reste du fichier. En cas de déchiffrement deux vérifications sont faites sur la clé fournie, une première sur les 16 premiers octets et une seconde sur les 16 derniers. Deux informations sont ici importantes : La première est que s'il nous est possible de chiffrer un fichier avec n'importe quelle clé, il nous sera impossible de le déchiffrer si nous n'avons pas la clé attendue. La deuxième est que les 16 premiers octets du fichiers secret sont le *md5* du fichier secret (moins les 16 premiers octets). Nous pouvons donc immédiatement faire cette vérification.

```
$ dd if=secret bs=1 count=16 | hexdump -C
16+0 records in
16+0 records out
16 bytes (16 B) copied, 0.00335965 s, 4.8 kB/s
00000000  b8 4d b9 ec 23 52 4e 4e  55 77 03 fb 55 df c0 83  |.M..#RNNUw..U...|
00000010
$ dd if=secret skip=16 | md5sum
2032+1 records in
2032+1 records out
1040400 bytes (1.0 MB) copied, 0.0515589 s, 20.2 MB/s
94a509d51d73e9bc690eefb133dc4d18  -
```

Et bien non, le *md5* ne marche pas... (comme par hasard encore une fois...).

2.2 secret à la loupe

La même méthode que pour *ssticrypt* ne peut pas ici être utilisée, car la taille donnée par *debugfs* est supérieure au nombre de block.

```
$ sudo debugfs partition -R "stat /home/sstic/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: 2064
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x4f6c3483 -- Fri Mar 23 09:29:55 2012
atime: 0x4fa78164 -- Mon May  7 10:01:40 2012
mtime: 0x4f6c3483 -- Fri Mar 23 09:29:55 2012
Size of extra inode fields: 0
BLOCKS:
(0-2):26625-26627
TOTAL: 3
```

Faisons la supposition que la taille est correcte et que les blocs constituant secret sont contigües. Il faut 257 blocs de données (*ndlr : 1048592/4096*) pour contenir le fichier *secret*. Comme le 13^{e} bloc est un bloc indirect, il faut prendre 258 blocs en partant du bloc 26625 et sauter le bloc 26637. Seuls les 16 (*ndlr : 1048592%4096*) premiers octets du dernier bloc devront être pris.

```
$ dd if=partition bs=4096 skip=26625 count=12 of=secret_part1
$ dd if=partition bs=4096 skip=$((26625+13)) count=$((256-12)) of=secret_part2
$ dd if=partition bs=4096 skip=$((26625+257)) count=1 of=secret_last_bloc
$ head -c 16 secret_last_bloc > secret_part3
$ cat secret_part1 secret_part2 secret_part3 > secret
$ tail -c +17 secret | md5sum
b84db9ec23524e4e557703fb55dfc083  -
```

Notre supposition était donc bonne. Le fichier *secret* est maintenant correct. Plus qu'à trouver la clé pour le déchiffrer.

La vérification de la clé pour le déchiffrement se fait via la fonction *check_key*.

3 Dans les méandres d'une White Box DES en python

Les 16 premiers octets de la clé sont vérifiés grâce à un programme *python* embarqué dans *ssticrypt*.

3.1 Reverse de bytecode Python

Le programme *python* est extrait par *check_key* dans le répertoire courant, exécuté avec en paramètre les 16 premiers octets de la clé, puis supprimé. L'extraction peut soit se faire en évitant la suppression après l'exécution, soit en le récupérant directement dans *ssticrypt* (se référer au *readelf -a* pour les offsets).

```
$ dd if=ssticrypt of=check.pyc bs=1 skip=$((0x413030-0x413020+0x3020))
count=$((0x4457d))
279933+0 enregistrements lus
279933+0 enregistrements écrits
279933 octets (280 kB) copiés, 0,325456 s, 860 kB/s
$ file check.pyc
check.pyc: python 2.5 byte-compiled
```

Bien entendu c'est du bytecode *python* et le code source n'est pas fourni... Après avoir exploré une partie du net et avoir testé une bonne grosse quantité d'outils permettant d'avoir un code source à partir d'un bytecode *python* (*unpyc*, *uncompyle*, *unpyc3*, *uncompyle2*, *UnPyc*, *decompyle*, *byteplay*...), force est de constater qu'aucun n'a fonctionné réellement correctement. Seule la librairie *marshal* [2] inclue de base dans python a été utilisée :

```
[...]
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)

[...]
```

Le code produit par cette librairie est certes loin de s'approcher du code source :

```
[...]
            276 CALL_FUNCTION            1
            279 STORE_NAME               27 (WT)

50101         282 LOAD_NAME                28 (len)
            285 LOAD_NAME                6 (sys)
            288 BUILD_MAP                29
            291 CALL_FUNCTION            1
            294 LOAD_CONST               23 (1)
            297 LOAD_ATTR                2 (log)
            300 JUMP_IF_FALSE_OR_POP     19
            303 POP_TOP

50102         304 LOAD_CONST          24 ('Usage: python check.pyc <key>')
            307 PRINT_ITEM
            308 PRINT_NEWLINE

50103         309 LOAD_CONST          25 ('   - key: a 64 bits hexlify-ed string')
            312 PRINT_ITEM
            313 PRINT_NEWLINE

50104         314 LOAD_CONST          26 ('Example: python check.pyc 0123456789abcdef')
            317 PRINT_ITEM
            318 PRINT_NEWLINE
            319 JUMP_FORWARD            159 (to 481)
            322 POP_TOP
[...]
```

Mais avec une bonne (grosse) dose de motivation et une bonne documentation [3] il est possible d'obtenir un code source fidèle.

L'analyse de ce code source montre la définition d'une classe *Bit*, l'implémentation de l'algorithme symétrique *DES* et la définition d'une classe appelée *WhiteDES*. L'objectif est de trouver la clé qui quel que soit le message en clair, produira le même chiffré avec l'algorithme *DES* qu'avec la classe *WhiteDES*. Cette dernière étant configurée avec des tables contenues dans des objets *pickles* de taille suffisamment grande pour faire ramer la plupart des éditeurs de texte...

Après quelques recherches sur internet, on trouve rapidement un papier très intéressant décrivant le fonctionnement d'une *White-Box DES* [7].

3.2 White-Box DES

La supposition est faite ici que le lecteur connait le fonctionnement de l'algorithme *DES*.

3.2.1 Principe

L'objectif d'une *White-Box DES* est d'implémenter l'algorithme *DES* avec une clé fixe. Cette implémentation doit ou devrait faire en sorte que l'extraction de cette clé soit impossible. Le principe est de modifier l'ensemble des *SBOX DES* originales par des *TBOX* qui produiront le même résultat que les *SBOX* associées à une clé fixe. Pour rendre le procédé plus robuste à des attaques statistiques, les *TBOX* sont rendues bijectives en modifiant la quantité d'informations qu'elles traitent (entrée) et la quantité d'information qu'elles produisent (sortie). Pour des besoins d'implémentation et de sécurité, 4 *TBOX* supplémentaires sont ajoutées ne correspondant pas directement à des *SBOX*. Enfin des transformations d'initialisation (*M1*), à la fin de chaque tour (*M2*) et à la fin du process entier (*M3*) sont mise en place pour effectuer des permutations, des

expansions/réductions de données, des mélanges de *TBOX* et aussi pour des besoins d'implémentation.

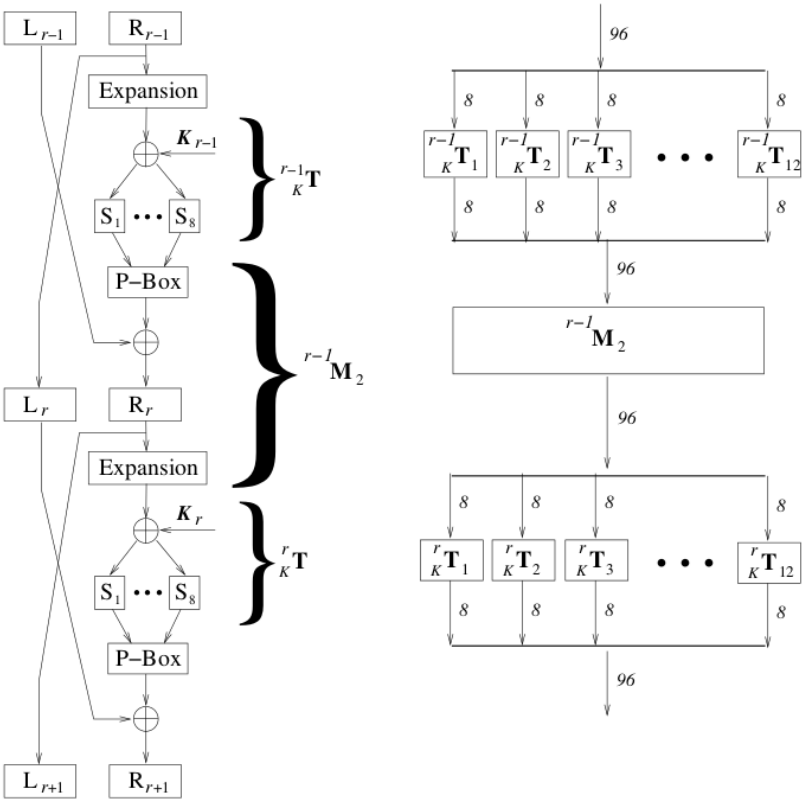


FIGURE 1 – Linéarisation des *SBOX DES*

Dans l'implémentation de cette *WhiteBox*, les seules données dépendantes de la clé sont les *TBOX* correspondantes aux *SBOX*. Même si les transformations (*M1*, *M2*, *M3*) sont importantes pour le bon fonctionnement de l'algorithme, elles n'ont pas de lien direct avec la clé recherchée et peuvent donc être ignorées dans l'attaque qui sera menée. Un élément toutefois important produit par ces transformations est le mélange des *TBOX*. La *TBOX i* ne correspond pas forcément à la *SBOX i*.

3.2.2 Construction d'une *TBOX*

Pour attaquer une *WhiteBox* il est important de comprendre comment sont constituées les *TBOX*. Chaque *SBOX S* est remplacée par une *SBOX Sk* qui produirait le même résultat que *S* si *S* était utilisée avec la clé *K*. Ajouté à cela, chaque *SBOX* qui prend normalement 6 bits en entrée et produit 4 bits en sortie, est modifiée pour prendre 2 bits supplémentaires en entrée et produire

8 bits en sortie. Pour une *TBOX* les 8 bits d'entrée se décomposent donc de la façon suivante :

- les 6 bits de poids fort sont les bits d'entrée de la SBOX correspondante
- les 2 bits de poids faible sont 2 bits supplémentaires [2]

De la même façon, les 8 bits de sorties se décomposent de la façon suivante :

- les 4 bits de poids fort sont les bits de sortie de la *SBOX* correspondante
- les 2 bits suivants sont le bits de poids fort et le bit de poids faible des 6 bits d'entrée de la *SBOX* correspondante
- les 2 bits de poids faible sont les 2 bits supplémentaires ajoutée en entrée de la *TBOX*

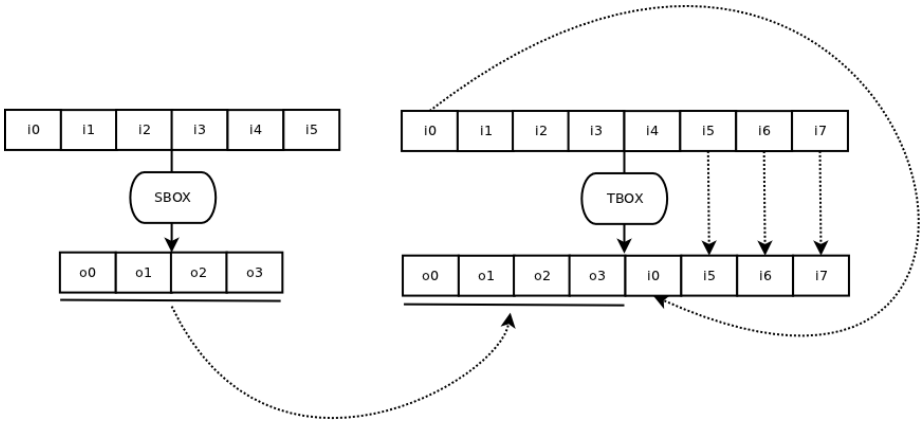


FIGURE 2 – Construction d'une *TBOX*

3.2.3 Attaque sur la WhiteBox

Comme dit précédemment, chaque *SBOX* prend en entrée 6 Bits. Ces 6 bits, sont *xorés* avec 6 bits de la clé puis produisent 4 bits. Cette sortie sera égale aux 4 premiers bits de la sortie de la *TBOX* correspondante. Il est donc possible de tester l'ensemble des clés en entrée d'une *SBOX*, et de vérifier quelle clé produit les mêmes résultats qu'une *TBOX*. Si le résultat correspond, alors la *TBOX* peut correspondre à la *SBOX* testée et la clé est un candidat possible. Il suffit de réitérer la manipulation avec des messages différents jusqu'à qu'il n'y ait plus qu'une seule clé candidate.

2. La source de ces bits n'est pas importante pour notre besoin

```
[...]
def break_sbox_key(wt,sbox):
    """ Casse la sous clé de la sbox correspondant à la WhiteBox wt """
    # Pour toute les clés possible
    for k in xrange(64):
        tbox = range(12)
        bad_key = False
        for m in xrange(64):
            M = Bits(m,6)
            res_sbox = SBOX(M,k,sbox)
            for i in xrange(4):
                M2 = M//Bits(i,2)
                good_tbox = []
                for ntbox in tbox:
                    res_tbox = wt.KT[0][ntbox][M2.ival]
                    if Bits(res_tbox,8)[0:4] == res_sbox:
                        good_tbox.append(ntbox)
                tbox = good_tbox
                if len(tbox) == 0:
                    bad_key = True
                    break
            if bad_key: break
        if not bad_key: return Bits(k,6)
    return None
[...]
```

Cette manipulation permet de récupérer 6 bits de la clé *DES* par *SBOX*, donc au total 48 bits. Les 8 bits restants seront trouvés par bruteforce.

Le résultat final produit la clé *fd4185ff66a94afd* qui représente la première moitié de la clé *RC4* recherchée.

4 Analyse du MIPS

La deuxième partie de la clé est vérifiée par un mécanisme tout autre, l'interrogation d'un périphérique USB, comme le montre les appels à *usb_get_bus* et *usb_ctrl_msg*. Les librairies utilisées par *ssticrypt* confirment que le programme utilise la libusb [4].

```
$ ldd ssticrypt
        libssl.so.0.9.8 => /usr/lib/libssl.so.0.9.8 (0x2aada000)
        libusb-0.1.so.4 => /lib/libusb-0.1.so.4 (0x2ab37000)
        libc.so.6 => /lib/libc.so.6 (0x2ab4f000)
        libcrypto.so.0.9.8 => /usr/lib/libcrypto.so.0.9.8 (0x2acc4000)
        libdl.so.2 => /lib/libdl.so.2 (0x2ae44000)
        libz.so.1 => /usr/lib/libz.so.1 (0x2ae58000)
        /lib/ld.so.1 (0x2aaa8000)
```

La fonction *check_key* pour la deuxième partie de la clé, commence par initialiser une zone mémoire de 4096 octets (qui sera appelée *RAM* dans la suite de ce document) puis recherche un périphérique USB avec les identificats *0x41c 0x9d*. Une fois ce périphérique trouvé, 2 buffers embarqués dans *ssticrypt* lui sont envoyés : *init.rom* et *stage2.rom*. Enfin la fonction *vicpwn_handle* est appelée avec en paramètre la deuxième partie de la clé. Cette fonction sera responsable du refus ou non de notre clé.

Remarque : Tous les détails d'implémentation comme les allocations, libération mémoire, inversion d'octets... non directement nécessaire à la compréhension du problème seront ignorés (même si bien entendu ils ne l'ont pas été pour la résolution).

4.1 Fonction vicpwn_handle

La fonction *vicpwn_handle* a une boucle principale pouvant faire au maximum 3 tours. A chaque tour, des données embarquées dans *ssticrypt* et appelées *layer* sont chargées dans la RAM (*load_layer*). Une partie de la clé est elle aussi chargée dans la *RAM* (*set_my_key*). Ensuite intervient une boucle secondaire, qui va communiquer avec le périphérique USB. A chaque tour de cette boucle, *vicpwn_handle* va envoyer une partie de la *RAM* au périphérique et se mettre en attente d'une réponse de 20 octets. Une fois celle-ci reçue, les 16 premiers octets seront considérés comme des données à écrire dans la RAM à l'adresse contenu dans les 16 et 17^{e} octet. Les octets 18 et 19 seront eux considérés comme l'adresse des 20 octets suivants en *RAM* dont le périphérique a besoin.

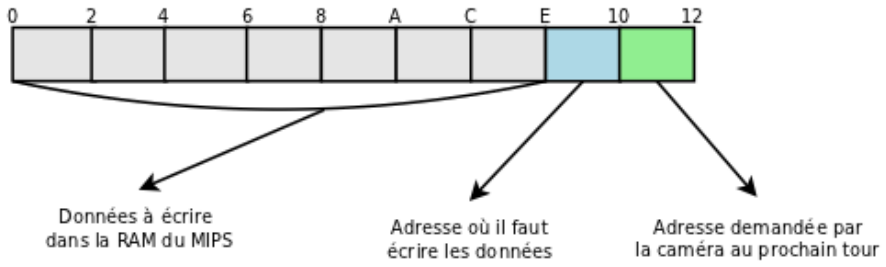


Figure 3 – Données reçues par l'USB

Cette boucle secondaire peut être considérée comme un moyen pour le périphérique USB d'interagir directement avec la zone mémoire *RAM* du programme *ssticrypt*, comme si cette dernière était mappée dans son propre adressage mémoire.

Dès que l'adresse *0x8000* de la *RAM* est modifiée, la boucle d'interaction avec le périphérique USB s'interrompt et une vérification est faite sur l'état de la *RAM* laissé par le périphérique USB (*vicpwn_check*). Cette vérification est différente à chaque tour de boucle et donc à chaque *layer*. Si l'état de la *RAM* est valide vis à vis de *vicpwn_check*, alors on continue dans *vicpwn_handle* en chargeant le *layer* suivant (modifié lors de l'appel de *vicpwn_check*), sinon on s'arrête pour cause de clé invalide.

Le pseudo-code (simplifié) de la fonction *vicpwn_handle* peut se résumer à cela :

```
def vicpwn_handle():
layer = all_layers[0]
addr_ram = 0

for ilayer in xrange(3):
 load_layer(layer, ilayer, 0)
 set_my_key(key, ilayer, 0xa000)

 while ram[0x8000:0x8002] == "\x00\x00":
     send_to_usb_device(ram[addr_ram:addr_ram+20],20)
       buf = rcv_from_usb_device(20)
       addr_write = buf[-4:-2]
       addr_ram = buf[-2:]
       ram[addr_write:addr_write+16] = buf[:16]
layer = vicpwn_check(i,0xa000,key)
ram[0x8000:0x8002] = "\x00\x00"
```

Remarque : Le premier *layer* est chargé tel quel dans la *RAM*. Les suivants sont retournés par *vicpwn_check* après transformation de leur contenu. On peut donc supposer que les *layer2* et *layer3* ne sont pas en clair dans le programme *ssticrypt*, contrairement au *layer1*.

4.2 Fonction load_layer

La fonction *load_layer* est appelée à chaque tour de boucle par *vicpwn_handle*. Sa seule action est de charger à l'adresse 0 de la *RAM* le *layer* issu de *vicpwn_check* (ou le premier *layer* si nous sommes au premier tour).

4.3 Fonction set_my_key

La fonction *set_my_key* est elle aussi appelée à chaque tour de boucle par *vicpwn_handle*. Elle charge une partie de la clé à l'adresse *0xa000* de la RAM. Au second tour, les données *blob* contenues dans les data du programme *ssticrypt* seront elles aussi chargées dans la RAM à l'adresse *0xa010*. Au troisième tour, ce seront les données *blah* qui seront chargées en *0xa010*.

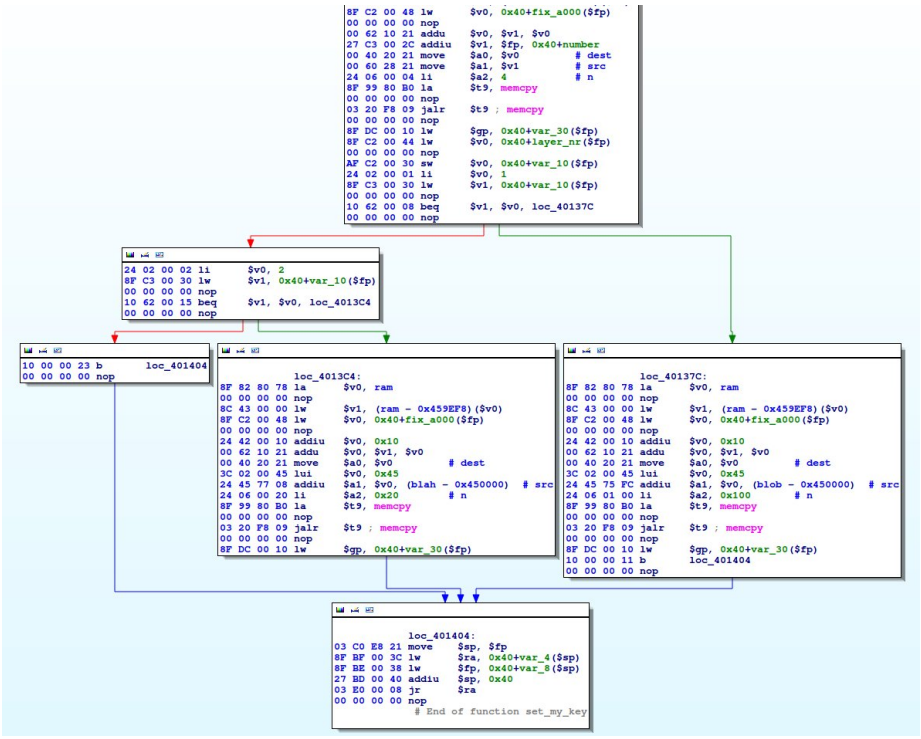


FIGURE 4 – set_my_key sous *IDA*

Au premier tour les caractères de 0 à 8 seront chargés en mémoire, au second tour les caractères de 4 à 12 et enfin au 3° tour les caractères de 8 à 16. Comme chaque caractère représente en réalité un chiffre hexadécimal, il est fort probable que le premier tour nous permette de découvrir 4 octets la clé, le suivant 2 nouveaux octets (4 octets au total mais 2 déjà découvert au tour précédent) et enfin le dernier encore 2 nouveaux octets (même remarque que précédemment).

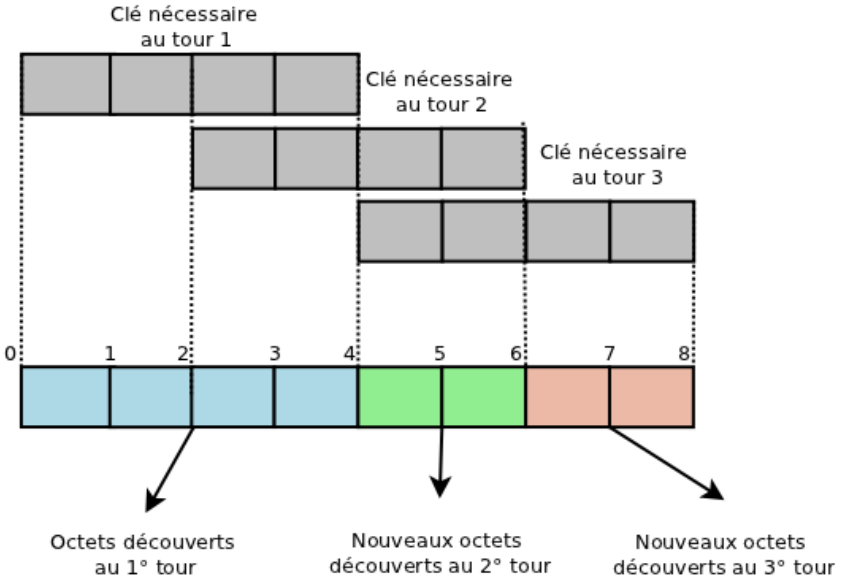


FIGURE 5 – Entrelacement des octets de la clé

Ce fonctionnement laisse donc penser que comme la découverte de la clé se fait de façon progressive, il serait possible, de lancer un bruteforce sur les derniers octets de la clé (cependant par soucis de compréhension cette technique pour challenger pressé ne sera pas utilisée).

4.4 Fonction vicpwn_check

Cette fonction est appelée à la fin de chaque tour de boucle par *vicpwn_handle* pour vérifier l'état de la *RAM* et mettre à jour le layer suivant. La vérification de l'état de la *RAM* est propre à chaque tour de boucle (et donc propre au *layer* chargé), ce qui veut dire que chaque morceau de clé est vérifié de façon différente.

4.4.1 Premier tour

Pour que la clé soit valide au premier tour, il suffit que la *RAM* à l'adresse *0xa000* ne soit pas égale aux 4 premiers octets de la clé (pour rappel les 4 premiers octets de la clé sont chargés lors du *set_my_key* en *0xa000*). L'équipement USB doit donc changer la clé en *0xa000* pour que l'on puisse considérer les 4 premiers octets de la clé comme valide, si ce n'est pas le cas alors le programme *ssticrypt* s'arrètera en disant que la clé est mauvaise.

Les modifications apportées au *layer* suivant sont faites à base de *XOR*. Les deux premiers octets du *layer2* (contenu dans les data de *ssticrypt*) sont *xorés* avec la nouvelle valeur contenue en *0xa000*, puis les 2 octets suivants sont *xorés* avec ce résultat et ainsi de suite. Le résultat final donnera le *layer* qui sera chargé en mémoire. Le programme suivant permet d'effectuer cette transformation.

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

import struct,sys

if len(sys.argv) != 2:
    print ("Usage: %s hex_key" % sys.argv[0])
    sys.exit(1)

key = int(sys.argv[1],16)
layer1_str = open("layer2.bin","r").read()
layer_size = len(layer1_str)

def swap_word(word):
    w1 = word&0xff
    w2 = ((word&0xff00)>>8)&0xff
    return ((w1<<8)|w2)&0xffff

def swap_key(x):
    w1 = (x&0xffff)&0xffff
    w2 = ((x&0xffff0000)>>16)&0xffff
    w1 = swap_word(w1)
    w2 = swap_word(w2)
    return ((w2<<16)|w1)&0xffffffff

def unpack(s):
    return struct.unpack(">I",s)[0]

def pack(i):
    return struct.pack(">I",i)

key = swap_key(key)
count = 1
ptr = pack(key ^ unpack(layer1_str[:4]))
while True:
    if count >= (layer_size-2)/4:
        f=open("layer2_unencode.bin","wb")
        f.write(ptr)
        f.close()
        sys.exit(0)

    a0 = unpack(layer1_str[count*4:(count+1)*4])
    v0 = ((1 - count)*4)
    v0 = -v0
    v0 = unpack(ptr[v0:v0+4])
    ptr = ptr + pack(v0 ^ a0)
    count += 1
```

La transformation du *layer2* étant dépendante de la valeur en *RAM* à l'adresse *0xa000* (modifiée par le dispositif USB), il n'est pas possible de décoder le *layer2* à cet instant.

4.4.2 Deuxième tour

Pour que la clé soit valide au deuxième tour, il suffit que la *RAM* à l'adresse *0xa108* soit différente de *0xffff*. A cette adresse normalement il y a les 2 derniers octets de *blob* cité plus haut. Ces deux octets sont différents de *0xffff*, on peut donc conclure que le contenu de *blob* est modifié par le dispositif USB. Les modifications apportées au *layer* suivant sont basées sur le contenu final du *blob*. Les octets du *layer3* sont *xorés* avec les valeurs contenues l'intérieur. Contrairement au tour précédent, une opération dépendante des valeurs du *blob* permet de trouver quelle valeur du *blob* il faut utiliser pour le *xor*.

La fonction C suivante permet de déchiffrer le contenu du *layer* grâce au *blob*.

```
void unencode_layer3(uint8_t *layer, uint8_t *blob, uint8_t *ptr) {
   uint16_t count = 0;
   uint8_t byte = 0;
   uint8_t store1 = 0;

   while(count<SIZE_LAYER) {
      uint8_t store2;
      uint8_t addr;
      uint8_t v1,v0,a0,a2;

      byte += 1;

      // SWAP
  // Comme store1 est sur 8 bits et que blob fait 256 octets,
      // l'overflow ne pose pas de problème
      store1 += VALUE_AT(blob+byte);
      store2 = VALUE_AT(blob+byte);
      v1 = VALUE_AT(blob+store1);
      VALUE_AT(blob+byte) = v1;
      VALUE_AT(blob+store1) = store2;

      a2 = VALUE_AT(layer+count);
      a0 = VALUE_AT(blob+byte);
      v0 = VALUE_AT(blob+store1);

      v1 = (v0+a0) & 0xff;
      v0 = VALUE_AT(blob+v1);

      v0 ^= a2;
      VALUE_AT(ptr+count) = v0;
      count += 1;
   }
}
```

4.4.3 Troisième tour

Le troisième et dernier tour effectue une comparaison de chaînes de caractères pour savoir si la clé est correcte. Il faut que la *RAM* à l'adresse *0xa010* contienne une chaîne de caractères qui soit égale à *V29vdCAhISBTbWVsbHMgZ29vZCA6KQ==* (soit *Woot!! Smells good :)* en base64).

Si à chaque tour les vérifications sont correctes, la clé finale est alors la bonne. Dans les autres cas, le programme *ssticrypt* s'arrête. A chaque tour pour savoir si les morceaux de clé sont correctes, il faut que le contenu de la *RAM* ait une certaine valeur. Or celle-ci n'est modifiée que par le dispositif USB. Donc pour réellement comprendre comment est modifiée la RAM, il faut comprendre quelles sont les actions menées par le dispositif USB.

Les seules informations que nous avons sur ce périphérique USB sont les identifiants *0x41c* et *0x9d*, et les drivers embarqués dans le système Linux correspondant à l'image. Après quelques recherches, le périphérique en question est controlé par le driver *vicam*, un driver de webcam [5]. Etant donné qu'il y a peu de chance qu'une webcam embarque de base un code faisant des vérifications sur une clé pour valider un challenge, le reverse des firmwares *init.rom* et *stage2.rom* envoyé au démarrage à la caméra semble malheureusement la seule solution.

5 CPU CY16

Après de nombreuses recherches sur le net, la caméra en question semble posséder un CPU *cy16*, dont bien sûr nous ne connaissons rien. La seule docu-

mentation trouvée est quand même relativement complète sur le site cypress [6]. Nous n'avons toutefois aucune certitude sur ce CPU. D'autres recherches nous apprennent que des outils de developpement existent pour ce CPU, mais qu'ils ne sont disponibles qu'avec l'achat d'une carte *cypress*... tentant mais onéreux juste pour un challenge, surtout quand on a pas la certitude du processeur.

La documentation trouvée étant relativement complète, un désassembleur a été developpé pour comprendre les firmwares *init.rom* et *stage2.rom*. De par mon manque d'expérience en reverse, un émulateur a aussi été écrit, pour confirmer l'analyse statique de ces firmwares.

5.1 Désassembleur *CY16*

L'écriture d'un désassembleur n'est en soit pas si difficile que ça (si on omet tous les bugs que l'on peut avoir, la mauvaise compréhension de la documentation, ou encore les informations manquantes). Il faut cependant prévoir tous les cas possibles et la moindre erreur provoque un décalage qui fait que toute la suite du code sera mal désassemblée. Les premiers résultats du désassembleur étaient tout simplement catastrophiques... et le code produit ne voulait strictement rien dire... remettant en cause régulièrement la véracité du CPU. Pour essayer de limiter les erreurs, des warnings ont été affichés à chaque fois qu'un *JMP* ou un *CALL* se faisait sur une adresse non valide.

```
Warning: 0x0 point to [0x8a9] which is not an instruction
Warning: 0x8c point to [0xe8] which is not an instruction
Warning: 0xfc point to [0x4da] which is not an instruction
Warning: 0x18c point to [0x1a6] which is not an instruction
Warning: 0x198 point to [0x1a6] which is not an instruction
[..]
Warning: 0x782 point to [0x4da] which is not an instruction
Warning: 0x7a2 point to [0x4da] which is not an instruction
Warning: 0x7b0 point to [0x4da] which is not an instruction
Warning: 0x8ae point to [0x1bd] which is not an instruction
Warning: 0xa70 point to [0x1] which is not an instruction
Warning: 0xa70 point to [0xa74] which is not an instruction
      b6 c3 a9 08       0000                    JNC      [$r14+0x8a9]   (to [0x8a9])
            02 64       0004                    AND      $r2 , [$r8]
e7 07 76 00 aa 00       0006                    MOV      [0xaa] , 0x76
e7 07 56 f7 b4 00       000c                    MOV      [0xb4] , 0xf756
      c8 07 1a 01       0012                    MOV      $r8 , 0x11a
      e0 07 00 40       0016                    MOV      [$r8] , 0x4000
                                                ADDI     $r8 , 2
      e0 07 00 00       001a                    MOV      [$r8] , 0x0
                                                ADDI     $r8 , 2
      e0 07 00 00       001e                    MOV      [$r8] , 0x0
                                                ADDI     $r8 , 2
            97 cf       0022                    RET
            00 00       0024                    MOV      $r0 , $r0
[...]
            00 00       0054                    MOV      $r0 , $r0
            00 00       0056                    MOV      $r0 , $r0
            00 00       0058                    MOV      $r0 , $r0
            f1 ff       005a                    UKN      ?????????
            f2 ff       005c                    UKN      ?????????
            f3 ff       005e                    UKN      ?????????
            f4 ff       0060                    UKN      ?????????
            f5 ff       0062                    UKN      ?????????
            ff ff       0064                    UKN      ?????????
            ff ff       0066                    UKN      ?????????
[...]
```

En téléchargeant plusieurs firmwares sur internet et en les comparant, il a été possible de détecter une sorte de header (*b6c3* + 4 octets inconnus), que bien entendu il ne faut pas considérer comme des *opcodes*. Finalement et à force

de suppositions (sur lesquelles ont longtemps planées un doute), il a été possible de séparer des zones comme étant des données et non plus des *opcodes*, pour au final obtenir un désassemblage intéressant et cohérent.

```
$ ./sstic_pwn.py -c \emph{cy16} -o 6 -x stage2.rom
e7 07 76 00 aa 00 0000                MOV [0xaa] , 0x76
e7 07 56 f7 b4 00 0006                MOV [0xb4] , 0xf756
      c8 07 1a 01 000c                MOV $r8 , 0x11a
      e0 07 00 40 0010                MOV [$r8] , 0x4000
                                      ADDI $r8 , 2
      e0 07 00 00 0014                MOV [$r8] , 0x0
                                      ADDI $r8 , 2
      e0 07 00 00 0018                MOV [$r8] , 0x0
                                      ADDI $r8 , 2
            97 cf 001c                RET
-----------------------------------------------
00 00 00 00 00 00       001e
00 00 00 00 00 00       0024
00 00 00 00 00 00       002a
00 00 00 00 00 00       0030
00 00 00 00 00 00       0036
00 00 00 00 00 00       003c
00 00 00 00 00 00       0042
00 00 00 00 00 00       0048
00 00 00 00 00 00       004e
f1 ff f2 ff f3 ff       0054
f4 ff f5 ff ff ff       005a
ff ff ff ff ff ff       0060
ff ff 00 00 54 00       0066
00 00 00 00 00 00       006c
00 00 00 00 00 0e       0072
01 00                   0078
-----------------------------------------------
      c0 57 51 00 007a                CMP $r0 , 0x51
            05 c0 007e                JZ +0x5                       (to [0x8a])
      c0 57 56 00 0080                CMP $r0 , 0x56
            14 c0 0084                JZ +0x14                      (to [0xae])
      9f cf e8 00 0086                JMP 0xe8                      (to [0xe8])
      c9 07 68 00 008a                MOV $r9 , 0x68
            41 08 008e                MOV $r1 , [$r9]
                                      ADDI $r9 , 2
            4a 08 0090                MOV $r10 , [$r9]
                                      ADDI $r9 , 2
            48 d8 0092                ADDI $r8 , 2
            22 08 0094                MOV [$r10] , [$r8]
                                      ADDI $r8 , 2
                                      ADDI $r10 , 2
            21 08 0096                MOV [$r9] , [$r8]
                                      ADDI $r8 , 2
                                      ADDI $r9 , 2
            48 02 0098                MOV $r8 , $r9
            61 94 009a                XOR [$r9] , [$r9]
                                      ADDI $r9 , 2
            61 00 009c                MOV [$r9] , $r1
                                      ADDI $r9 , 2
      e1 07 10 00 009e                MOV [$r9] , 0x10
                                      ADDI $r9 , 2
      e1 07 f6 00 00a2                MOV [$r9] , 0xf6
                                      ADDI $r9 , 2
      c1 07 00 80 00a6                MOV $r1 , 0x8000
            51 af 00aa                INT [0x51]
            97 cf 00ac                RET
[..]
```

L'option *-x* demande le désassemblage, l'option *-o* spécifie l'offset de début (pour sauter le *header*) et l'option *-c* dit que le CPU est un *cy16* (deux seulement existent, *cy16* et *vm* qui sera vu par la suite). Les adresses de données sont stockées en dures[3] (c'est mal !). A cet instant tous les *JMP* et *CALL* se font

3. Elles se trouvent de *0x1e* à *0x7a*, de *0xfe* à *0x146*, et enfin de *0x8a8* à *0xa6e*

a des adresses valides, mais deux *opcodes* restent inconnus et non documentés et ont les valeurs *0xdfc6* et *0xdfc7*. Le dernier opcode documenté a la valeur *0xdfc5*, ce qui nous laisse supposer que nous avons la bonne famille de CPU mais pas exactement le bon CPU (ou que la documentation n'est pas à jour).

Pour simplifier le reverse, un graphique a été généré permettant de comprendre l'enchainement des blocs (grâce à *graphviz*).

```
$ ./sstic_pwn -o 6 -g stage2.png stage2.rom
```

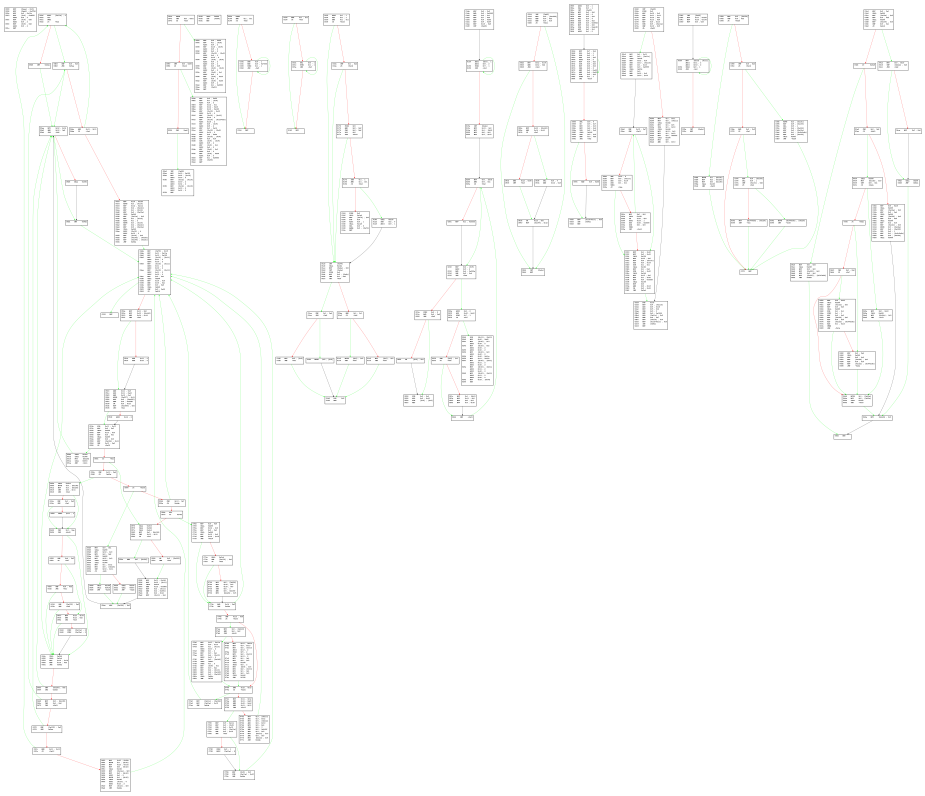


FIGURE 6 – Machine virtuelle implémentée par le firmware de la webcam

5.2 Emulateur *CY16*

L'émulateur *CY16* développé *from scratch* gère tous les *opcodes* (*MOV*, *CALL*, *JMP*...), les flags et quelques interruptions. Il possède de plus certaines fonctionnalités nécessaires à son utilisation :

- gestion de l'historique des commandes
- *breakpoints*, *breakpoints* conditionnels, *breakpoint* en lecteur sur zone mémoire, *breakpoint* en écriture sur zone mémoire...
- affichage de zones mémoire de la caméra ou de la *RAM* du *MIPS*, registres ou flags
- possibilité d'exécuter des commandes quand on arrive à certaines adresses (de façon conditionnelle)

– pas à pas, saut dans un call ...
– possibilité d'automatisation via l'exécution d'un script

Voilà un exemple de script utilisé :

```
key = "\xbb\xaa\xdd\xcc"
extra = open("blob.bin","rb").read()
self.cpu.ram_mips = open("layer2_unencode.bin","rb").read()
self.cpu.ram_mips = self.cpu.ram_mips + "\x00"*(0x10000-len(self.cpu.ram_mips))
self.cpu.ram_mips = self.cpu.ram_mips[0:0xa000] + key +
self.cpu.ram_mips[0xa000+len(key):]
self.cpu.ram_mips = self.cpu.ram_mips[0:0xa010] + extra +
self.cpu.ram_mips[0xa010+len(extra):]

self.cpu.registers[15].val = 0x950
self.cpu.ram[0x11a] = "\x00\x40"
self.cpu.registers[8].val = 0x120
start 0x8a
#h 0xaa print "DATA at %s" % hex(self.cpu.unpack(
self.cpu.ram[self.cpu.registers[8].val+2]))
#h 0x4f4 print "First bit = %u" % self.cpu.registers[0].val
#h 0x502 print "2,3,4 bit = %u" % self.cpu.registers[0].val
#h 0x544 print "OPCODE = %u" % self.cpu.registers[0].val
#h 0x42e print "OPERAND = %r" % self.cpu.ram[0x126:0x128]
#h 0x26c print "NB_BITS:%r" % self.cpu.registers[0].val,
#h 0x2c6 print "VALUE:%s" % hex(self.cpu.registers[0].val)
#h 0x4d6 print "IMMEDIATE VALUE"
#h 0x4a8 print "DIRECT ADDRESSING"
#h 0x3ee print "MOVO"
#h 0x49a print "AUTRE: 1"
#h 0x454 print "AUTRE: 2"
#h 0x226 print "SIZE: %s" % hex(self.cpu.registers[10].val)


######## WARNINGS #######
#h 47e print "WARNING Unknown OPCODE 2 !!!''
#h 7b6 print "WARNING 1 ****************************"
#h 7e0 print "WARNING 2 ****************************"
#h 6ac print "WARNING UKN1 dans MOV *****************"
#h 620 print "WARNING ** REGISTRE 15 (0x620)"
#h 644 print "WARNING ** REGISTRE 14, on décrément le compteur (0x644)"
#h 6c0 print "WARNING ** On va jouer avec PC (0x6c0)"
#h 7e0 print "WARNING ** On va jumper mechant (0x7e0)"
#h 7b6 print "WARNING ** Sauvegarde PC (0x7b6)"


######## CMDS ###########
h 0x5be self.cpu.cmd="NOT"
h 0x58e self.cpu.cmd="AND"
h 0x588 self.cpu.cmd="OR"
h 0x60a self.cpu.cmd="MOV"
h 0x600 self.cpu.cmd="SHL"
h 0x5f6 self.cpu.cmd="SHR"
h 0x566 self.cpu.cmd="JMP"
h 0x4ec self.last_addr=hex(self.cpu.unpack(self.cpu.ram[0x11c:0x11e])*8
+self.cpu.unpack(self.cpu.ram[0x11e:0x120]))
h 0x4de if self.cpu.cmd is not None: print "%s => %s " %
(self.last_addr,self.cpu.cmd)

r
```

5.3 stage2.rom

L'analyse approfondie du *stage2.rom* via le désassembleur et l'émulateur nous permet d'aboutir à la conclusion que ce firmware implémente une machine virtuelle possédant son propre jeu d'instruction. L'état de cette machines virtuelle (registres, flags...) est en réalité situé dans ce que nous avons appelé les données du *stage2.rom* et est développé dans le chapitre XXX. La fonction

principale située à l'adresse *0x4da* permet en fonction des données en entrée d'exécuter la bonne instruction. Un pseudo-code ultra simplifié de cette fonction serait le suivant :

```
Si opcode == 0:
   AND_OPCODE()
Sinon Si opcode == 1:
   OR_OPCODE()
Sinon Si opcode == 2:
   NOT_OPCODE()
Sinon Si opcode == 3:
   SHL_or_SHR_OPCODE()
Sinon Si opcode == 4:
   MOV_OPCODE()
Sinon Si opcode > 4:
   JMP_or_CALL_OPCODE()
```

5.3.1 Communication USB

Les communications USB avec la machine *MIPS* sont réalisées par la fonction à l'adresse *0x7a*. Que ce soit pour l'envoi ou la réception de données, la structure de données suivantes semble être utilisée :

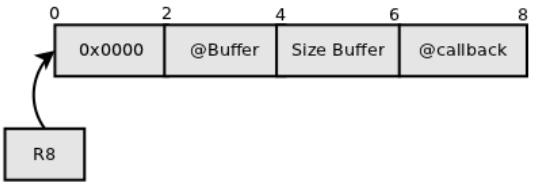


FIGURE 7 – Structure USB pour l'envoi et la réception

S'il s'agit d'une réception USB alors l'interruption *0x51*[4] sera appelée puis la *callback* située en *0xf6*. Elle appelle la fonction principale qui permet de parser et exécuter les *opcodes* puis appelle l'interruption *0x59*[5] qui termine la communication USB. S'il s'agit d'un envoi USB alors l'interruption *0x50*[6] est appelée puis la *callback* située en *0xe8*.

Pour émuler l'interruption *0x51* (réception), le registre d'instruction est modifié pour prendre l'adresse du *handle* de la structure précédemment décrite. De plus une partie de la mémoire *RAM* du *MIPS* est copiée dans la mémoire de la caméra. La taille de cette zone mémoire est précisée dans la structure mais

4. *USB_RECEIVE_INT*
5. *USB_FINISH_INT*
6. *USB_SEND_INT*

est toujours la même (*0x10*). L'adresse dans la *RAM* du *MIPS* dépend du précédent envoi USB (voir figure 3). L'adresse destination dans la mémoire de la caméra évolue avec le temps et sera développée plus tard.

Pour émuler l'interruption *0x50* (envoi), le registre d'instruction est modifié pour prendre l'adresse du *handle* de la structure. De plus une partie de la mémoire de la caméra est copiée dans la mémoire *RAM* du *MIPS*. L'adresse de destination dans la *RAM* du *MIPS* dépend de l'avant dernier mot de la zone mémoire.

Pour émuler l'interruption *0x59* (terminaison), un tour sur deux l'adresse *0xae* est mise en sommet de pile et l'autre tour c'est au tour de l'adresse *0x8a*. Ce fonctionnement permettra d'alterner entre l'envoi et la réception USB. Ce fonctionnement n'est bien entendu pas correct, mais dans ce cas précis, il est conforme à ce que fait le programme *MIPS*.

5.3.2 Mapping de la mémoire

Certaines instructions ont besoin d'un accès direct à la *RAM* du *MIPS* que ce soit en écriture ou simplement en lecture. Il est donc nécessaire de faire un mapping de ces zones dans la mémoire de la caméra, comme cela a déjà été vu dans le détail de la fonction *vicpwn_handle* (voir 4.1). La fonction située en *0x1da* a un lien direct avec ce besoin.

L'objectif de cette fonction est double, si la mémoire *MIPS* nécessaire à une instruction est disponible (mappée dans la mémoire de la caméra), alors son adresse dans la mémoire de la caméra sera retournée.

```
$ ./sstic_pwn.py -d -o 6 stage2.rom
...
>>> b 1da
>>> r
BREAKPOINT 1 !
01da MOV $r5 , $r0
>>> ir
$r0 => 0xa000
...
>>> b 228
>>> r
BREAKPOINT 2 !
0228 ADD $r0 , $r10
>> ir
$r0 => 0x0
...
>> x 2 0
bb aa
```

Dans cet exemple, l'adresse *RAM MIPS* en *0xa000* sera mappée en *0x0* de la mémoire de la caméra.

Le deuxième objectif de cette fonction est de demander un mapping de la mémoire si cette dernière ne l'est pas. Pour cela, la fonction utilise la zone mémoire allant de *0x54* à *0x5c* pour connaitre quelles sont les zones actuellement mappées. Il y a donc 5 zones mémoire qui peuvent être mappées au même moment dans la caméra. Chaque zone mémoire ayant une taille de 16 octets. La zone mémoire correspondant à l'adresse MIPS située en 0x54, sera située en 0x0, la zone mémoire correspondant à l'adresse MIPS située en 0x56 sera située en 0x10, ainsi de suite. Dans notre exemple précédent l'adresse *0xf* de la caméra correspondra donc à l'adresse *0xa00f* de la *RAM* du *MIPS*, l'adresse *0x10* correspondra à une zone mémoire complètement différente (ici *0x10* dans

la *RAM* du *MIPS*[7]).

```
$ ./sstic_pwn.py -d -o 6 stage2.rom
...
>> x 10 54
00 a0 10 00 20 00 f4 ff f5 ff
```

Si la zone mémoire n'est pas mappée, alors la fonction (en jouant avec les adresses de retour des fonctions), va faire une demande via l'USB au *MIPS* pour mapper cette mémoire et la même instruction sera réexécutée et pourra être menée à son terme (si plus aucune zone mémoire ne vient à manquer).

Le mapping d'une zone mémoire, entraîne le demapping d'une autre zone mémoire, étant donné que la quantité de zone mémoire mappée est limitée à 5. Pour cela, via un système de compteur situé dans les zones mémoire allant de *0x54+10* à *0x5c+10*, la zone mémoire la plus ancienne non utilisée (celle qui a le compteur le plus élevé) est demappée puis commitée dans le *MIPS*. A chaque fois qu'une zone mémoire est utilisée, son compteur est mis à 0, et à chaque fois que la fonction est appelée, tous les compteurs sont augmentés de 1. Ce système permet de s'assurer que les zones mémoires utilisées en dernier seront demappées les premières (car les moins suceptibles d'être utilisée).

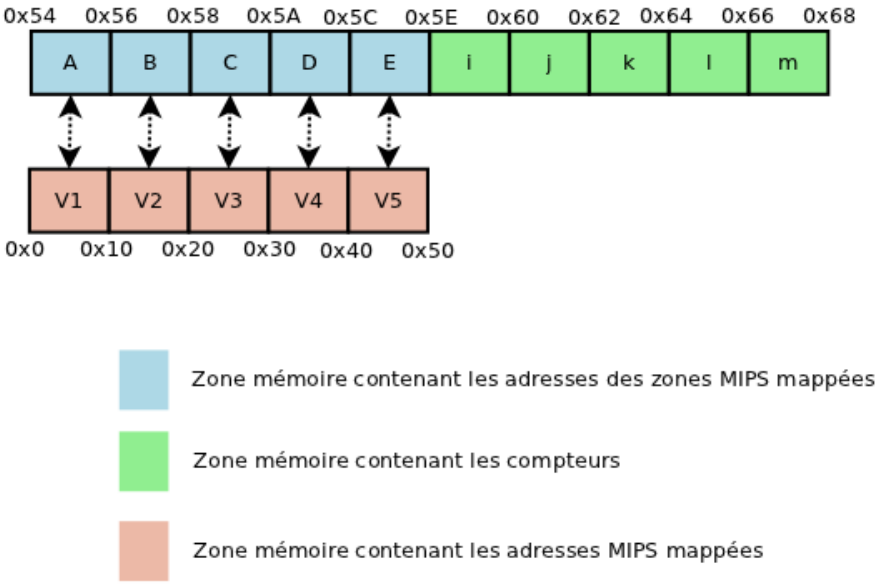


FIGURE 8 – Mapping de la mémoire MIPS dans l'adressage de la Webcam

5.3.3 Lecture des *opcodes* de la VM

La lecture des *opcodes* de la machine virtuelle se fait via la fonction en *0x268*. Cette fonction consomme un certain nombre [8] de bits à chaque fois qu'elle est appelée (contenu dans *r0*), dépendant de ce qu'elle a lu précédemment.

7. L'exemple n'est pas le meilleur qui soit...
8. Contenu dans le registre r0

Contrairement au CPU *CY16*, les *opcodes* du CPU implémenté par la machine virtuelle ne sont pas alignés. Pour savoir où elle en est, et qu'est ce qu'elle doit consommer, cette fonction stocke 2 valeurs. La première se situe en *0x11c* et représente l'indice du prochain octet à lire, et la seconde se situe en *0x11e* et représente l'indice du prochain bit à lire dans le prochain octet.

Cette fonction lit la *RAM* du *MIPS* via la fonction permettant le mapping d'adresse. Ce qui suggère que le code de la machine virtuelle se situe directement dans la *RAM* du *MIPS*. Les valeurs de *0x11c* et *0x11e* étant initialisées en dur à 0 dans le *stage2.rom*, le code de la machine virtuelle commence à l'adresse 0 de *RAM* et est donc contenu dans les *layers*.

Remarque : Dans le cas ou l'*opcode* a besoin d'une adresse non mappée, la fonction responsable du mapping restore les valeurs *0x11c* et *0x11e* à leur valeur d'origine pour rééxecuter la même instruction.

5.3.4 Gestion des opérandes

La gestion des opérandes des instructions est faite par deux fonctions. La première fonction se situe en *0x3c4*. Cette fonction va consommer des bits (ref lectures *opcodes* de la machine virtuelle) et en fonction de leurs valeurs, alors l'opérande pourra être soit la valeur contenue dans un registre, soit une valeur stockée en dur dans le layer, soit une adresse, soit une adresse contenue dans un registre, soit un offset relatif à une adresse contenue dans un registre. La suite de bits consommés pour la création de l'opérande a cette forme :

```
size = ReadBits(1)
type = ReadBits(2)
if type == 0:
  reg = ReadBits(4)
  res = *reg
elif type == 1:
  if size == 0:
    res = ReadBits(8)
  else:
    res = ReadBits(16)
elif type == 2:
  ref = ReadBits(2)
  if ref == 0:
    reg = ReadBits(4)
    temp = *reg
  elif ref == 1:
    temp = ReadBits(16)
  elif ref == 2:
    temp = ReadBits(16)
    reg = ReadBits(4)
    temp += *reg
  res = Get(temp)
else:
  # MOV chiffré décrit plus tard
  pass
```

Cette fonction construit une structure qui caractérisera l'opérande et qui aura la forme suivante :

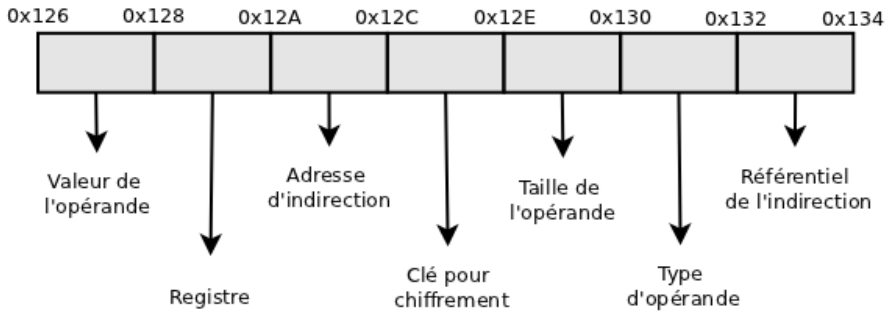


FIGURE 9 – Structure mémoire d'une opérande

Dans tous les cas la valeur en *0x126* sera la valeur finale (par exemple dans le cas d'une adresse ça sera la valeur située à cette adresse).

La deuxième fonction se situe en *0x346*. Elle a pour seul rôle de copier le contenu de la structure précédemment décrite en *0x134*. Concrètement cette fonction fait en sorte que l'opérande précédemment parsée, devienne la deuxième opérande.

5.3.5 Sauvegarde des résultats

La sauvegarde des résultats de chaque instruction est faite par la fonction *0x35e*. Elle peut se faire soit dans un registre, soit en mémoire (dépendant du type d'opérande). Ce qui est important, c'est que cette sauvegarde est conditionnelle, et dépend des bits consommés. Ce fonctionnement permet de conclure que toutes les commandes de la machine virtuelle sont conditionnelles.

5.3.6 Gestion des flags

Les gestion des flags est faite par la fonction *0x814*. Cette fonction sauvegarde les flags du CPU de la caméra soit en *0x144*, soit en *0x145*. Il y aura donc un flag *Z1*, un flag *Z2*, un flag *C1*, un flag *C2*... Chaque instruction pourra donc soit modifier les *flags 1*, soit modifier les *flags 2*, soit ne pas toucher aux flags. De plus l'exécution de chaque instruction dépendra des *flags 1* ou des *flags 2*.

Pour savoir si une condition est valide ou non, les *flags 1* ou les *flags 2* de la dernière instruction sont restaurés avant l'exécution de l'instruction courante. Le saut situé à l'adresse *0x537* est patché pour refléter la condition sauvegardée. S'il est pris alors le bit 1 du registre *r14* sera positionné. Au final l'instruction courante ne positionnera son résultat qu'en fonction de ce bit.

```
$ ./ssticpwn -o 6 -d stage2.rom
Python 2.7.2+ (default, Oct  4 2011, 20:06:09)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> b 524
>>> f script.txt
008a MOV $r9 , 0x68
BREAKPOINT 0 !
0524 MOVB [0x537] , $r11
>>> lb
BREAKPOINTS:
0 => 0x524:
>>> dis 536 2
0536 JNC +0x1
0538 ADDI $r14 , 2
>>> ir
[...]
$r11 => 0x94e
[...]
>>> s
0528 MOV $r0 , [0xc000]
>>> dis 536 2
0536 JMP +0x1
0538 ADDI $r14 , 2
```

Dans cet exemple, le *JNC* en *0x536* est devenu *JMP*.

5.3.7 MOV

Le pseudo-code de la fonction *MOV* située en 0x1a6 est le suivant :

```
Si R1 == 4:
   R0 = LoadWord(R0)
Si R1 == 0:
   StoreByte(R0,R2)
Si R1 == 1:
   StoreWord(R0,R2)
Sinon
   R0 = LoadByte(R0)
```

Cette fonction est appelée par la fonction *0x166* dont le pseudo-code est :

```
Si R1 == 0:
   StoreByte(R0,R2)
Si R1 == 2:
   R0 = LoadByte(R0)
Si R1 == 1:
   StoreWord(R0,SWAP(R2))
Si R1 == 3:
   StoreWord([R0],R2)
Si R1 == 4:
   R0 = LoadWord([R0])
```

5.3.8 MOV Chiffré

Une fonction située en *0x84c* peut être appelée soit dans la préparation des opérandes (*0x3c4*), soit dans la sauvegarde des résultats (*0x35e*). Cette fonction permet de lire ou d'écrire en mémoire tout en chiffrant la valeur. Par exemple, si une opérande fait appel à cette fonction lors d'une écriture, alors la valeur de l'opérande sera chiffrée avant d'être écrite en mémoire. Le chiffrement de la valeur se fait par l'intermédiaire d'une clé, constituée d'une adresse mémoire, d'une valeur de registre et d'un entier.

Cette fonction a une particularité, dans le cas ou elle est appelée avec les mêmes paramètres, une valeur chiffrée qui passera dans cette fonction sera déchiffrée.

En d'autres termes, si f représente cette fonction et k est la clé. Si $f(k,x) = y$, alors $f(k,y) = x$.

5.3.9 Fonction principale

La fonction principale située en *0x4da* permet en fonction du code lu venant du *layer* de construire les bonnes opérandes, d'exécuter les bonnes instructions et de sauvegarder ou non les résultats. Le reverse de cette fonction permet de construire un désassembleur pour la machine virtuelle implémentée par le firmware *stage2.rom*.

Le premier bit lu par cette fonction permet de savoir quels sont les flags qui seront utilisés par l'instruction, est ce que ce seront les *flags 1* ou les *flags 2* (voir 5.3.6). Les 8 bits suivants représentent les flags qui conditionnent l'exécution ou non de l'instruction [9]. En fonction du premier bit lu, seul les 4 premiers bits seront gardés, ou seul les 4 derniers. Cette condition sera comparée aux flags sauvegardés et le mécanisme d'altération du code présenté dans la gestion des flags sera exécuté. Les 3 bits suivants permettent de préciser quelle instruction doit être exécutée (*MOV*, *AND*, *OR*...). Enfin le bit suivant permet de spécifier si oui ou non la commande doit modifier les flags.

Le traitement suivant dépend de l'instruction à exécuter, et suivra généralement le modèle du chargement d'une ou deux opérandes et de la sauvegarde ou non des résultats.

5.3.10 Photographie de la mémoire

Pour fonctionner, la machine virtuelle va stocker son état dans la mémoire de la caméra. Après rétroconception du code, il est possible d'avoir une photographie précise de cette mémoire.

9. plus exactement la sauvegarde ou non des résultats de cette instruction

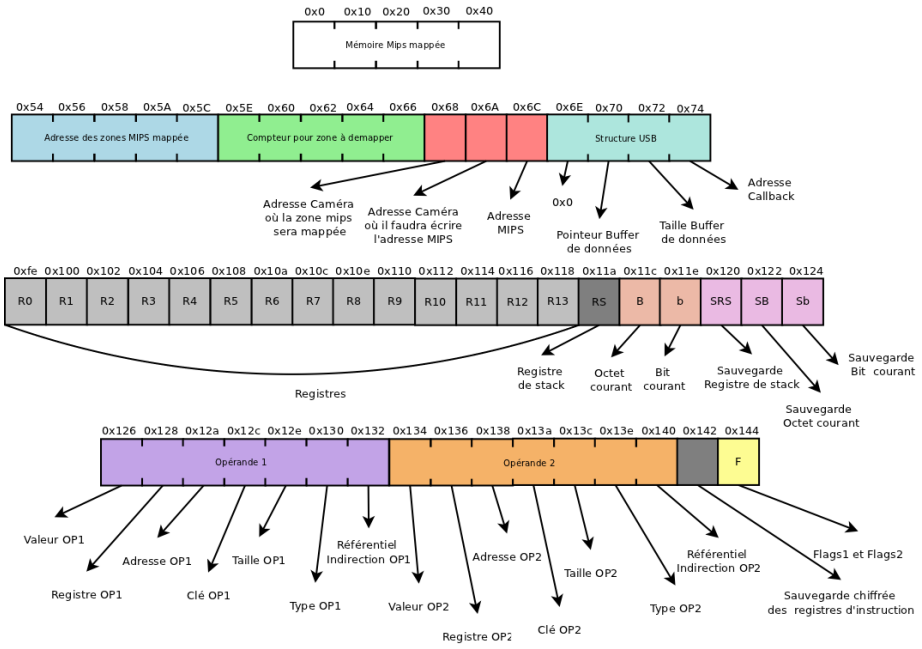


FIGURE 10 – Représentation de l'état de la machine virtuelle en mémoire

6 CPU Machine virtuelle

Grâce au reverse du firmware *stage2.rom*, il est possible d'écrire un désassembleur pour le code interprété par la machine virtuelle. Un émulateur n'a pas été écrit, mais l'émulateur du *cy16* a été réutilisé. Pour cela de nouvelles fonctions lui ont été ajoutées comme :

- *bvm* permettant de mettre des *breakpoints* à des adresses de la machine virtuelle (conditionnel, en lecture, écriture...)
- *irvm* permettant d'afficher les registres de la machine virtuelle
- *sfvm* permettant d'afficher les flags de la machine virtuelle

6.1 Layer 1

Comme pour *stage2.rom*, un listing du code assembleur du layer1 est disponible dans cette solution, ainsi qu'un graphique pour simplifier le reverse.

```
$ ./sstic_pwn -c vm -x layer1.bin > layer1.asm
$ ./sstic_pwn -c vm -g layer1.png layer1.bin
```

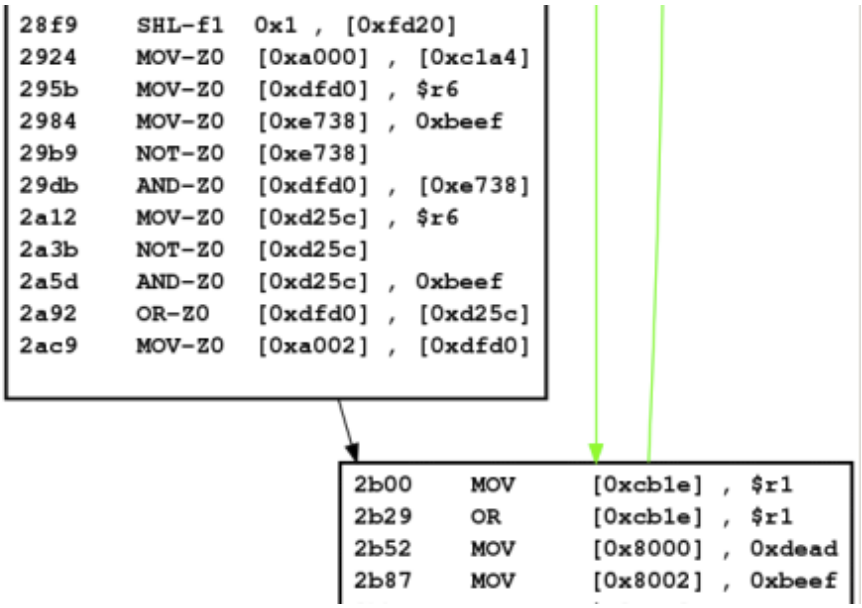


FIGURE 11 – Fin du *layer1*

Le code du *layer 1* semble correct car les dernières instructions modifient l'adresse *0x8000*, condition de sortie comme vu dans le reverse de *vicpwn_handle*. Cependant c'est à priori un code obfusqué, dans le sens ou le code produit pourrait être grandement simplifié. La machine virtuelle ne possède pas l'opération *XOR*, pourtant le *layer1* en utilise à outrance. Celui-ci se retrouve codé avec un mélange d'opération *OR*, *AND* et *NOT*[10].

Même si les blocs du *layer1* sont au final très semblables, ne connaissant pas le niveau de sadisme des auteurs du challenge, ils ont tous étaient reversés et simplifiés à la main, pour éviter une mauvaise surprise.

Au delà de la simplification du code avec des *XOR*, on remarque très rapidement qu'une partie vraiment très faible est dépendante de la clé (chargée dans les registres *r12* et *r3* lors des deux premières instructions). La majeur partie du code est fixe et peut donc être simplifiée sans problème. Au final, l'algorithme qui paraissait très long, se limite à son équivalent en C suivant :

10. Par exemple $p \oplus q = (p \wedge \neg q) \vee (\neg p \wedge q)$

```
[...]
int is_valid_key(uint32_t key) {
   uint8_t k0 = key&0xff;
   uint8_t k1 = (key>>8)&0xff;
   uint8_t k2 = (key>>16)&0xff;
   uint8_t k3 = (key>>24)&0xff;
   uint16_t k = (k3^k1^k0)<<8 | (k0^k1^k2^k3);

   if(k != 0xae4d) {
  return 0;
   } else {
  uint16_t x = (((k1^0x95)<<8) | (k1^k0^0x77))-0x539;

  if(((x+0x94ec)&0xffff) == 0) {
 printf("Key=%x A000=%x A002=%x",key,0x8cfa,x^0xbeef);
  }
   }
}
[...]
```

Une recherche exhaustive des clés ne pose aucun problème, et nous permet de connaître les nouvelles valeurs qui seront stockées en *RAM MIPS* à l'adresse *0xa000* (différentes de la clé de départ, conditions pour que la clé entrée soit correcte (voir 4.4.1).

```
$ ./layer1.c
Key=94e3e5df A000=8cfa A002=d5fb
```

La clé *0x94e3e5df* est une partie de la clé que nous recherchons (modulo l'inversion d'octets). Le contenu de la zone mémoire en *0xa000* nous permet de déchiffrer le *layer2* avec le programme fourni précédemment.

```
$ ./unencode_layer1.py fa8cfbd5
```

6.2 Layer 2

On commence par désassembler le *layer2* avec notre outil.

```
$ ./sstic_pwn -c vm -x layer2_unencode.bin > layer2.asm
$ ./sstic_pwn -c vm -g layer2.png layer2_unencode.bin
```

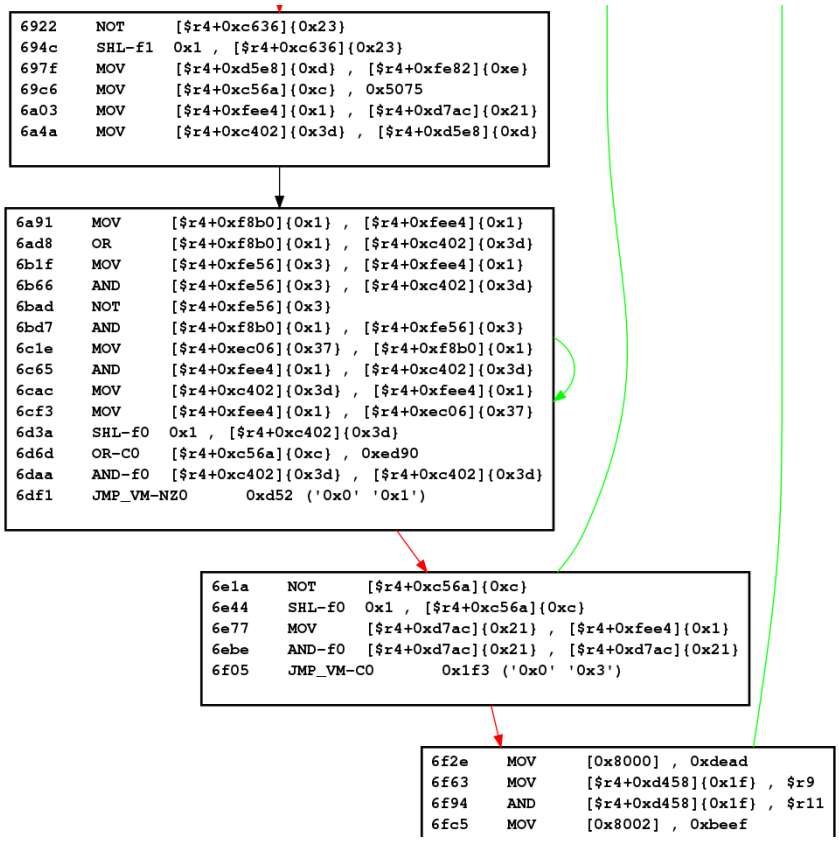


FIGURE 12 – Fin du *layer2*

Comme pour le *layer1*, le code semble correct à cause de la modification de la valeur contenue à l'adresse *0x8000* et du pattern *0xdead 0xbeaf* très connu. Contrairement au *layer1*, le *layer2* fait une utilisation à outrance des *opcodes* de chiffrement. Cependant comme expliqué lors du reverse de ces *opcodes*, quand ils lisent une donnée ils la chiffrent automatiquement, et si cette donnée avait été préalablement chiffrée avec la même clé, elle est automatiquement déchiffrée.

```
06b3                    MOV       [$r4+0xc56a]{0xc} , 0x24
06f0                    SHL       0x2 , [$r4+0xc56a]{0xc}
```

Dans cet exemple, la valeur *0x24* est stockée chiffrée à l'adresse *$r4+0xc56a* (en utilisant comme clé *4,0xc56a,0xc*). Cependant lors du *SHL*, la valeur est déchiffrée, le *SHL* s'effectue et le résultat est rechiffré avant d'être stocké en mémoire. Il est donc tout à fait possible de raisonner comme si le chiffrement n'avait pas lieu (tant que la même clé est utilisée).

Par contre si une zone mémoire non chiffrée est lue, celle-ci est automatiquement chiffrée avant que le calcul dessus ait lieu. Dans le *layer2* à l'exception de 3 instructions qui se trouvent aux adresses *0xf9b*, *0x1174* et *0x566A*, toutes les actions de chiffrement peuvent être ignorées. Cependant à ces adresses la mémoire qui est lue se situe dans la *RAM* dont le contenu n'est pas chiffré. Le

résultat de la lecture chiffrera donc l'information. Après analyse, ces instructions lisent à l'intérieur du *blob* qui est concaténé à la suite de la clé et qui servira pour déchiffrer le dernier *layer*.

Voilà un exemple de la première valeur qui va être lue dans blob et qui sera ensuite chiffrée :

```
$ ./ssticpwn -o 6 -d stage2.rom
Python 2.7.2+ (default, Oct  4 2011, 20:06:09)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> bvmr 0xa010 0xa110
>>> f script.txt
[...]
BREAKPOINT 0 !
01d0 MOV $r0 , [$r8]
>>> ir
[...]
$r8 => 0x10
[...]
>>> x 10 54
f0 01 10 a0 50 ed e0 01 90 ee
>>> s
>>> ir
$r0 => 0x4861
[...]
>>> s
01d2 JMP +0x1
>>> s
01d6 POP $r8
>>> s
01d8 RET
>>> s
0894 XOR $r0 , $r6
>>> ir
$r0 => 0x4861
[...]
$r6 => 0x3361
[...]
>>> s
0896 AND $r3 , 0x4
>>> ir
$r0 => 0x7b00
[...]
```

Un premier *breakpoint* a été positionné et sera déclenché dès qu'un zone mémoire MIPS entre les adresses 0xa010 et 0xa110 [11] sera lue. La valeur *0x4861* est lue à l'adresse *0xa010* et correspond aux deux premiers octets du *blob*. Cette valeur est immédiatement chiffrée via un *XOR* en *0x894*.

Le même principe d'obfuscation que pour le *layer1* est utilisé. Le code suivant en C permet non seulement d'obtenir la clé par recherche exhaustive, mais aussi de modifier le *blob*. Le déchiffrement du *layer3* a déjà été évoqué précédemment.

11. *0xa010* et *0xa110* sont ici des adresses MIPS et non de la caméra

```
[...]
uint16_t is_valid(uint16_t key) {
   int i;
   uint16_t k = key;
   uint16_t k1 = 0x94e3;
   uint8_t blob[BLOB_SIZE];

   FILE *fd = fopen(BLOB,"r");
   fread(blob,1,BLOB_SIZE,fd);
   fclose(fd);

   for(i=0;i<64;i++) {
  uint16_t v = o(0x9fc0,0x50+4*i+2,7,*((uint16_t*)blob+2*i+1));
  uint16_t v1 = o(0x9fc0,0x50+4*i,7,*((uint16_t*)blob+2*i));

  ((uint16_t*)blob)[2*i] = v1 ^ k1;
  ((uint16_t*)blob)[2*i+1] = v ^ k;
  k -= i;
  k1 += i;
   }
   if(((uint16_t*)blob)[127] == 0xbe92) {
  write_layer(blob);
  printf("blob written\n");
  return 1;
   }
   return 0;
}
[...]
```

Et le résultat :

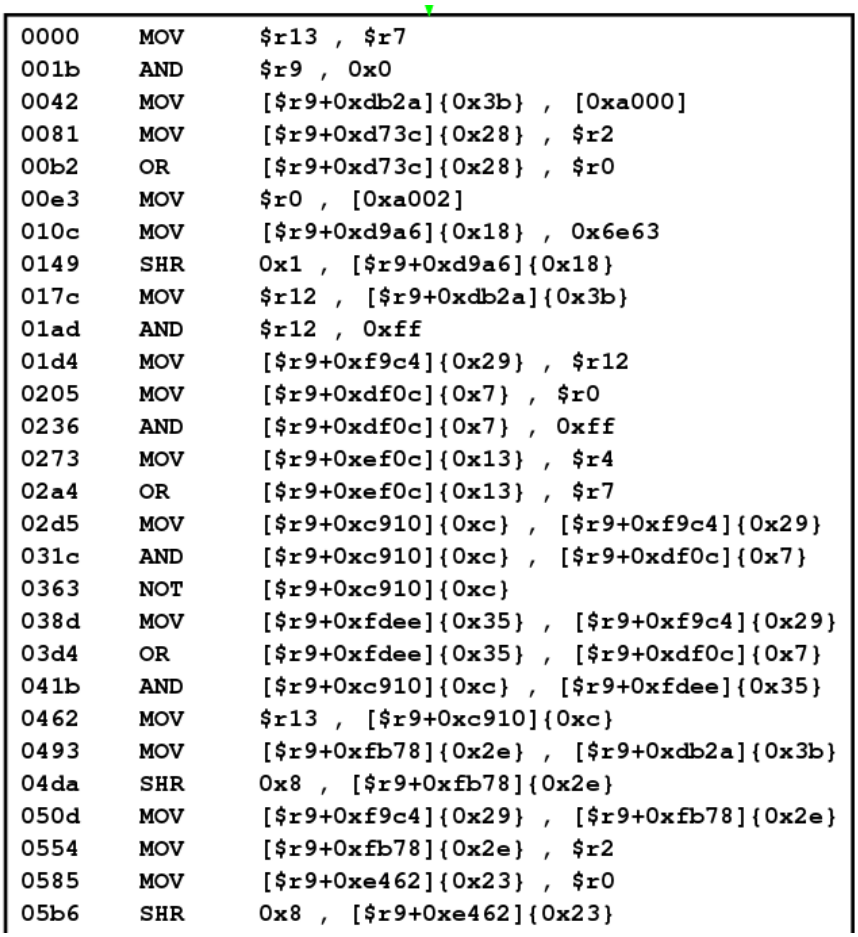
```
$ ./layer2
blob written
KEY = f63d
```

Seul les octets manquants sont écrits, les deux octets (*0xe5fd*) en commun avec le *layer1* ont été ignorés.

6.3 Layer 3

On commence par désassembler le *layer3* avec notre outil.

```
$ ./sstic_pwn -c vm -x layer3_unencode.bin > layer3.asm
$ ./sstic_pwn -c vm -g layer3.png layer3_unencode.bin
```



```
0000    MOV     $r13 , $r7
001b    AND     $r9 , 0x0
0042    MOV     [$r9+0xdb2a]{0x3b} , [0xa000]
0081    MOV     [$r9+0xd73c]{0x28} , $r2
00b2    OR      [$r9+0xd73c]{0x28} , $r0
00e3    MOV     $r0 , [0xa002]
010c    MOV     [$r9+0xd9a6]{0x18} , 0x6e63
0149    SHR     0x1 , [$r9+0xd9a6]{0x18}
017c    MOV     $r12 , [$r9+0xdb2a]{0x3b}
01ad    AND     $r12 , 0xff
01d4    MOV     [$r9+0xf9c4]{0x29} , $r12
0205    MOV     [$r9+0xdf0c]{0x7} , $r0
0236    AND     [$r9+0xdf0c]{0x7} , 0xff
0273    MOV     [$r9+0xef0c]{0x13} , $r4
02a4    OR      [$r9+0xef0c]{0x13} , $r7
02d5    MOV     [$r9+0xc910]{0xc} , [$r9+0xf9c4]{0x29}
031c    AND     [$r9+0xc910]{0xc} , [$r9+0xdf0c]{0x7}
0363    NOT     [$r9+0xc910]{0xc}
038d    MOV     [$r9+0xfdee]{0x35} , [$r9+0xf9c4]{0x29}
03d4    OR      [$r9+0xfdee]{0x35} , [$r9+0xdf0c]{0x7}
041b    AND     [$r9+0xc910]{0xc} , [$r9+0xfdee]{0x35}
0462    MOV     $r13 , [$r9+0xc910]{0xc}
0493    MOV     [$r9+0xfb78]{0x2e} , [$r9+0xdb2a]{0x3b}
04da    SHR     0x8 , [$r9+0xfb78]{0x2e}
050d    MOV     [$r9+0xf9c4]{0x29} , [$r9+0xfb78]{0x2e}
0554    MOV     [$r9+0xfb78]{0x2e} , $r2
0585    MOV     [$r9+0xe462]{0x23} , $r0
05b6    SHR     0x8 , [$r9+0xe462]{0x23}
```

FIGURE 13 – Début du *layer3*

Le *layer3* ne présente aucune difficulté supplémentaire comparée au *layer2*. Les octets de *blah* sont lus et chiffrés à la volée puis *xorés* avec une transformation faite sur la clé pour être réécrits dans *blah* à la même position. La valeur finale de *blah* doit être égale à *V29vdCAhISBTbWVsbHMgZ29vZCA6KQ==*. Le code C suivant permet d'effectuer une recherche exhaustive sur les 2 derniers octets manquants de la clé [12].

12. Les deux premiers octets sont trouvés grâce au *layer* précédent et sont égals à *0xf63d*

```
[...]
int is_valid_key(uint16_t key) {
   uint8_t i;
   uint8_t x = (KEY_PART1&0xff)^(key&0xff);
   uint8_t y = (KEY_PART1>>8)^(key>>8);
   uint16_t k = (y<<8) + x + y;

   uint16_t blah[SIZE_BLAH];
   memcpy(blah,blah_orig,SIZE_BLAH);

   for(i=0;i<32;i+=2) {
  // o = fonction de chiffrement
  uint16_t v = o(0xa00c,i+4,0x33,*((uint16_t*)blah+(i/2)));
  ((uint16_t*)blah)[i/2] = v ^ k;
   }
   if(!strncmp((const char *)blah,SOLUTION,32))
  return 1;
   return 0;
}
[...]
```

Et le résultat :

```
$ ./layer3
KEY = 8937
```

La rétroconception des 3 layers permet donc d'obtenir la deuxième partie de la clé RC4 qui est *e5df94e3f63d8937*.

7 Dernière ligne droite

Le cassage de la *White-Box DES*, puis le reverse du firmware de la webcam et de ses *layers* nous permettent d'obtenir la clé *RC4* tant attendue : *fd4185ff66a94afde5df94e3ff63d8937*.

Comme bien entendu nous ne disposons pas de caméra, le programme *ssticrypt* ne pourra initialiser le périphérique et la vérification de la deuxième partie de la clé va échouer. Soit il nous suffit de patcher *ssticrypt* pour qu'il ne faisse plus la vérification de la 2e partie de la clé, soit nous réécrivons juste la partie responsable du déchiffrement de *secret*. La deuxième solution est choisie. Les deux points à ne pas oublier sont qu'il ne faut pas prendre le *md5* contenu en début de fichier, et apporter les modifications nécessares à base de *XOR* au secret avant de le déchiffrer.

```
[...]
// secret est transformé avant d'être déchiffré
void unencode(char *data, int size) {
   int i;
   for(i=1;i<size;i++) {
      data[i-1] = data[i-1] ^ data[i];
   }
}

[...]
void decrypt_rc4(char *key, char *indata, int size) {
   RC4_KEY rc4_k;
   char outdata[size];

   RC4_set_key(&rc4_k,LEN,key);
   RC4(&rc4_k,size,indata,outdata);

   int i;
   for(i=0;i<size;i++) {
      printf("%c",outdata[i]);
   }
}
[...]
```

Et enfin la récompense [13] :

```
$ ./rc4_decrypt fd4185ff66a94afde5df94e3f63d8937 secret > result
$ file result
result: Linux rev 1.0 ext2 filesystem data, UUID=ace27cef-09d4-4d79-ad64-42597535b42e
$ sudo mount -o loop result /mnt/loop
$ ls -l /mnt/loop
total 918K
-rw-r--r-- 1 root root 901K 2012-03-19 17:41 lobster
drwx------ 2 root root  12K 2012-03-19 17:40 lost+found
$ file /mnt/loop/lobster
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)
$ mplayer /mnt/loop/lobster
```

13. Après quelques prières pour que ce fichier *result* ne soit pas encore une énième étape ;)

FIGURE 14 – Lobster Dog !!!!!

Remarque : En réalité le programme ci-dessus (fourni avec ce rapport) permet aussi de bruteforcer les 2 derniers octets de la clé. J'avoue n'avoir reversé le *layer3* qu'après avoir soumis la solution. Les deux derniers octets de la clé ont été trouvé par bruteforce.

8 Conclusion

Pour ma première participation au challenge SSTIC, j'avoue ne pas être déçu. Je n'avais qu'une peur en commençant ce challenge, c'est que ça soit du *reverse*. J'avais visé juste !

Un grand merci aux auteurs, que ça soit pour la partie *forensic*, *cryptographie* ou bien entendu la partie *reverse*. Chaque partie était vraiment très intéressante, difficile mais sans pour autant être insurmontable [14].

14. On se dit toujours ça après coup

Références

[1] http://en.wikipedia.org/wiki/Ext2.

[2] http://docs.python.org/library/marshal.html.

[3] http://unpyc.sourceforge.net/Opcodes.html.

[4] http://libusb.sourceforge.net/doc/index.html.

[5] http://homeconnectusb.sourceforge.net.

[6] http://www.cypress.com/?docID=14345.

[7] H. Johnson S. Chow, P. Eisen and P.C. Oorschot. A white-box des implementation for drm applications. *Proceedings of ACM CCS-9 Workshop DRM*, 2003. http://crypto.stanford.edu/DRM2002/whitebox.pdf.