Solution du Challenge SSTIC 2012

www.oppida.fr

6 avenue du Vieil Etang • Bât.B • 78180 Montigny-le-Bretonneux

Tél. : +33 (0)1 30 14 19 00 • Fax : + 33 (0)1 30 14 19 09 • Mail : contact@oppida.fr

SARL AU CAPITAL DE 270 000 € • RCS : VERSAILLES B419 296 090 • SIRET : 419 296 090 00030 • CODE APE : 6202A

SOMMAIRE

1. Extraction des fichiers

Le challenge SSTIC 2012 se présente sous la forme d'une image disque de 1Go une fois décompressée. La première étape du challenge consiste donc à récupérer les fichiers à partir de cette image disque.

Pour cela nous avons utilisé les utilitaires `file`, `dd`, `fsck` et `mount`.

Tout d'abord file nous informe de la position et de la taille de l'unique partition contenue dans l'image disque :

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

Nous extrayons alors la partition à l'aide de `dd` :

```
eloi@debian:~$ dd if=./dump.img of=./partition1 bs=512 count=2088387 skip=63
2088387+0 enregistrements lus
2088387+0 enregistrements écrits
1069254144 octets (1,1 GB) copiés, 63,4428 s, 16,9 MB/s
```

Nous obtenons alors une partition ext2 que nous montons à l'aide de `mount` :

```
eloi@debian:~$ file ./partition1
./partition1: Linux rev 1.0 ext2 filesystem data, UUID=5712fd31-bf27-4376-bbf4-
aabab500ba7b (large files)
eloi@debian:~$ mkdir ./sstic
eloi@debian:~$ sudo mount -t ext2 -o loop  ./partition1 ./sstic/
```

Nous pouvons alors commencer à explorer la partition, les fichiers intéressants se trouvent dans le répertoire `/home/sstic/` :

```
eloi@debian:~$ ls -al ./sstic/home/sstic/
total 1364
drwxr-xr-x 2 test test    4096 23 mars  09:51 .
drwxr-xr-x 3 root root    4096 26 janv. 13:42 ..
-rw-r--r-- 1 test test     220 26 janv. 13:42 .bash_logout
-rw-r--r-- 1 test test    3116 26 janv. 13:42 .bashrc
-rw-r--r-- 1 root root     871 23 mars  09:51 irc.log
-rw-r--r-- 1 test test     675 26 janv. 13:42 .profile
-rwxr-xr-x 1 root root 1048592 23 mars  09:29 secret
-rwxr-xr-x 1 root root     128 23 mars  09:30 ssticrypt
```

Le fichier `irc.log` nous donne alors des indications sur le challenge :

```
eloi@debian:~$ cat ./sstic/home/sstic/irc.log
#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_inter
esting_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> ...
```

Nous déduisons de ce log que nous devons déchiffrer le fichier `secret` à l'aide de l'exécutable `ssticrypt` et que l'image disque est probablement corrompue, ce qui est confirmé par la taille de `ssticrypt`, bien trop faible pour un exécutable :

```
eloi@debian:~$ file ./sstic/home/sstic/ssticrypt
./sstic/home/sstic/ssticrypt: ELF 32-bit MSB executable, MIPS, MIPS-I version 1
(SYSV), statically linked, stripped
eloi@debian:~$ ls -al ./sstic/home/sstic/ssticrypt
-rwxr-xr-x 1 root root 128 23 mars  09:30 ./sstic/home/sstic/ssticrypt
```

Nous tentons donc de réparer le système de fichier à l'aide de `fsck` (en répondant bêtement oui à toutes les questions ☺) :

```
eloi@debian:~$ sudo fsck.ext2 ./partition1
e2fsck 1.41.12 (17-May-2010)
./partition1 n'a pas été démonté proprement, vérification forcée.
Passe 1 : vérification des i-noeuds, des blocs et des tailles
I-noeud 14, i_blocs est 2064, devrait être 24. Corriger<o>? oui

I-noeud 19, i_size est 128, devrait être 311296. Corriger<o>? oui

I-noeud 40820, i_size est 236, devrait être 40960. Corriger<o>? oui

Passe 2 : vérification de la structure des répertoires
Passe 3 : vérification de la connectivité des répertoires
Passe 3A : optimisation des répertoires
Passe 4 : vérification des compteurs de référence
Passe 5 : vérification de l'information du sommaire de groupe
différences de bitmap de blocs:  -(26628--26882)
Corriger<o>? oui

Le décompte des blocs libres est erroné pour le groupe n°0 (31843, décompté=32098).
Corriger<o>? oui

Le décompte des blocs libres est erroné (232969, décompté=233224).
Corriger<o>? oui


./partition1: ***** LE SYSTÈME DE FICHIERS A ÉTÉ MODIFIÉ *****
./partition1 : 5469/65280 fichiers (1.5% non contigus), 27824/261048 blocs
```

Nous obtenons alors un fichier `ssticrypt` d'une taille raisonnable que nous pouvons commencer à étudier :

```
eloi@debian:~$ ls -al ./sstic/home/sstic/ssticrypt
-rwxr-xr-x 1 root root 311296 23 mars  09:30 ./sstic/home/sstic/ssticrypt
```

2. Etude du binaire principal

Comme `file` nous l'a révélé dans le chapitre précédent, `ssticrypt` est un exécutable MIPS big endian, nous utilisons donc IDA Pro 6.2 pour l'étudier.

L'analyse montre rapidement que `ssticrypt` permet de chiffrer ou déchiffrer un fichier à l'aide d'une clé de 128 bits qui lui est passé en paramètre sous la forme d'une chaine de 32 caractères hexadécimaux. Les fichiers sont chiffrés ou déchiffré à l'aide de l'algorithme RC4, importé depuis OpenSSL 0.9.8 et d'une simple boucle permettant le *mix* des octets du fichier (chaque octet chiffré dépend des précédents). Dans le cas où le fichier doit être déchiffré, son MD5 (stocké dans les 16 premiers octets du fichier) est vérifié. Enfin, avant toute opération de chiffrement ou de déchiffrement, la clé passée en paramètre est découpée en deux parties de 16 caractères qui sont vérifiées indépendamment à l'aide de deux routines différentes situées dans la fonction `check_key`.

A noter que la clé ne peut être `513239706269456751323970626945d`, dans le cas contraire un message d'erreur est affiché (`Error: You're a duck`) et le programme se termine. Cette limitation a surement pour but d'empêcher les canards d'utiliser ce programme, la haine ancestrale qui existe entre les fou à pieds bleus (`blue_footed_booby`) et les canards étant légendaire.

```
eloi@debian:~$ python -c "print
'513239706269456751323970626945d3d'.decode('hex').decode('base64')"
Coin! Coin!
```

3. Vérification de la première partie de la clé

La vérification de la première partie de la clé est effectuée à l'aide d'un script python 2.5 compilé (`check.pyc`), extrait dans le répertoire courant lors de l'exécution et exécuté à l'aide de la fonction `system`. Si le code de retour du script est nul alors la première partie de la clé est valide.

3.1. Décompilation du script

Le script python étant compilé, la première étape de l'analyse consiste à le décompiler afin d'obtenir du code python lisible à partir du bytecode[1]. Les décompileurs pour python 2.5 disponibles ne parvenant pas à décompiler correctement le code, nous avons utilisé un décompileur pour python 2.6 codé dans le cadre d'une précédente mission, basé sur uncompyle 2 (permettant la décompilation de bytecode python 2.7), que nous avons porté pour python 2.5.

Les modifications apportées à uncompyle 2 pour lui permettre de décompiler du python 2.5 et 2.6 sont de deux types :

- Conversion du bytecode 2.5 / 2.6 en bytecode 2.7 :
  - Remplacement des instructions `JUMP_IF_{FALSE, TRUE}` par `POP_JUMP_IF_{FALSE, TRUE}`.
  - Suppression des `POP_TOP` associés.
  - Correction des opcodes `LIST_APPEND` qui prennent un argument dans python 2.7 et non dans python 2.5 / 2.6.
  - Remplacement des anciennes suite d'opcodes par la nouvelle instruction `SETUP_WITH` introduite avec python 2.7[2].
  - Correction des deltas utilisés par les adresses relatives.
  - Correction de la structure `lnotab` (servant à faire le lien entre les opcodes et les numéros de lignes[3]).
- Ajout des structures propres à python 2.5 et 2.6 :
  - Compréhension de liste (optimisé dans python 2.7).
  - Gestion des *tranches* pour l'itération de liste (par exemple `for i in l[1:2:3]`, optimisé à partir de python 2.6).

Le résultat obtenu n'est pas parfait pour python 2.5 (quelques erreurs de structures dans le cas de conditions imbriquées subsistent) mais le résultat est aisément corrigeable et nous obtenons rapidement un code fonctionnel équivalent au bytecode.

[1] http://docs.python.org/release/2.5.2/lib/bytecodes.html
[2] http://bugs.python.org/issue6101
[3] http://hg.python.org/cpython/file/default/Objects/lnotab_notes.txt

3.2. Récupération de la clé DES

A la lecture du script décompilé, on se rend rapidement compte que nous sommes en face d'une WhiteBox[1] DES (nom de classe, sboxes, tableaux de grande tailles). La clé passée en paramètre est utilisée pour chiffrer un bloc choisi aléatoirement, si le chiffré est égale au chiffré obtenu avec la WhiteBox alors la clé est considérée comme valide.

Il est à noter que dans le domaine des WhiteBox il est rarement (jamais ?) nécessaire de casser entièrement la WhiteBox, c'est à dire de trouver la clé correspondante. Il est en effet beaucoup plus simple de récupérer un algorithme équivalent (dans le cas où la WhiteBox sert à chiffrer des challenge ou à signer des données) ou de l'inverser (dans le cas où l'on souhaite déchiffrer des données chiffrées à l'aide d'une WhiteBox) sans avoir connaissance de la clé.

3.2.1. Description de la WhiteBox

La WhiteBox est relativement simple et a une structure très proche d'une implémentation standard de DES :

WhiteBox	DES
– Substitution initiale tM1 : $B_{64} \rightarrow B_{96}$.	– Permutation initiale IP : $B_{64} \rightarrow B_{64}$.
– 16 tours.	– 16 tours.
– Substitution finale tM3 : $B_{96} \rightarrow B_{64}$.	– Permutation finale FP : $B_{64} \rightarrow B_{64}$.

Tour de WhiteBox	Tour de DES
– Découpage du bloc : $B_{96} \rightarrow {B^1}_8, {B^2}_8, ..., {B^{12}}_8$.	– Découpage du bloc : $B_{64} \rightarrow L_{32}, R_{32}$.
– ${B^i}_8 = \text{S-BOX}^{KTi}{}_8({B^i}_8)$.	– Calcul de la clé de tour $K_i = F(K, R_{32}, i)$
– $B_{96} = {B^1}_8 \,\|\| \, {B^2}_8 \,\|\| ... \,\|\| \, {B^{12}}_8$.	– $B_{64} = R_{32} \,\|\| \, L_{32} \oplus K_i$
– $B_{96} = \text{S-BOX}_{FX}(B_{96})$	

Les différentes S-Boxs utilisées par la WhiteBox sont sérialisées à l'aide du module `pickle` de python, probablement pour des raisons de performance lors de la compilation et de l'exécution du script (la taille des S-Boxs étant de plus de 1 Ko).

3.2.2. Calcul des clés de tour

La première chose à faire pour casser une WhiteBox DES est de récupérer la valeur intermédiaires du bloc chiffré afin d'en déduire les clés de tour qui permettrons à leur tour de retrouver la clé principale.

Etant donné la structure de la WhiteBox, nous faisons l'hypothèse instinctive que pour obtenir la valeur intermédiaire du bloc *classique* ${B^i}_{64}$ à partir du bloc de tour de la WhiteBox ${B^i}_{92}$, il suffit de lui appliquer l'inverse de la permutation initiale de la WhiteBox (tM1) puis la permutation initiale classique de DES (IP).

[1] http://www.phrack.org/issues.html?issue=68&id=8

La permutation initiale de la WhiteBox n'étant pas injective, (le bloc de sortie est plus grand que le bloc d'entrée) nous ne retenons que les bits étant utilisés dans la permutation finale de la WhiteBox.

Cette inversion est faite à l'aide du code python suivant :

```
>>> t = [None]*64
>>> for i, j in enumerate(tM1) :
...     if i in tM3 :
...             t[j] = i
...
>>> t
[16, 31, 32, 63, 48, 75, 0, 91, 71, 30, 79, 62, 87, 74, 95, 90, 70, 23, 78, 55, 86,
73, 94, 89, 5, 22, 21, 54, 37, 72, 53, 88, 8, 15, 24, 47, 40, 67, 56, 83, 69, 14, 77,
46, 85, 66, 93, 82, 68, 7, 76, 39, 84, 65, 92, 81, 61, 6, 13, 38, 29, 64, 45, 80]
```

Pour vérifier notre hypothèse nous modifions le code de la WhiteBox afin d'afficher la valeur intermédiaire du bloc et nous vérifions que nous avons bien $L^{i+1}_{32} = R^{i}_{32}$.

Code modifié de la WhiteBox permettant de récupérer les *blocs de tour*
(les lignes ajoutées sont identifiées par un '!')

```
def _cipher(self, M, d):
    if not M.size == 64:
        raise AssertionError
!   tM1_inv = [16, 31, 32, 63, 48, 75, 0, 91, 71, 30, 79, 62, 87, 74, 95, 90, 70, 23,
78, 55, 86, 73, 94, 89, 5, 22, 21, 54, 37, 72, 53, 88, 8, 15, 24, 47, 40, 67, 56, 83,
69, 14, 77, 46, 85, 66, 93, 82, 68, 7, 76, 39, 84, 65, 92, 81, 61, 6, 13, 38, 29, 64,
45, 80]
    if d == 1:
        blk = M[self.tM1]
        for r in range(16):
            t = 0
!           blk_n = IP(blk[tM1_inv])
!           L = blk_n[:32].ival
!           R = blk_n[32:].ival
!           print "Tour %d :"%r
!           print "\tL = %08X"%L
!           print "\tR = %08X"%R
            for n in range(12):
                nt = t + 8
                blk[t:nt] = self.KT[r][n][blk[t:nt].ival]
                t = nt

            blk = self.FX(blk)
!       blk_n = IP(blk[tM1_inv])
!       L = blk_n[:32].ival
!       R = blk_n[32:].ival
!       print "Dernier tour :"
!       print "\tL = %08X"%L
!       print "\tR = %08X"%R
        return blk[self.tM3]
    if d == -1:
        raise NotImplementedError
```

Sortie du code modifié

```
Tour 0 :
        L = 5C6EDC2C
        R = CBA7E952
Tour 1 :
        L = CBA7E952
        R = AF77450C
Tour 2 :
        L = AF77450C
```

```
Tour 8 :
        L = B1087047
        R = 43255E1E
Tour 9 :
        L = 43255E1E
        R = 06FE7CBF
Tour 10 :
        L = 06FE7CBF
```

```
        R = 5C85A1B1
Tour 3 :
        L = 5C85A1B1
        R = DC204C45
Tour 4 :
        L = DC204C45
        R = 4A8DC7C9
Tour 5 :
        L = 4A8DC7C9
        R = 50E36396
Tour 6 :
        L = 50E36396
        R = 758EFF65
Tour 7 :
        L = 758EFF65
        R = B1087047
```

```
        R = BCAD73F5
Tour 11 :
        L = BCAD73F5
        R = 3B650C21
Tour 12 :
        L = 3B650C21
        R = 593A2C06
Tour 13 :
        L = 593A2C06
        R = 217419AB
Tour 14 :
        L = 217419AB
        R = D703BBF3
Tour 15 :
        L = D703BBF3
        R = 1B702DF6
Dernier tour :
        L = 1B702DF6
        R = 2D353057
```

La sortie du code modifié confirme notre hypothèse, nous avons bien réussi à calculer les blocs de tours. Pour calculer la clés du $i^{ème}$ tour il suffit maintenant de calculer le ou exclusif bit à bit du demi bloc $R^{i+1}_{32} \oplus L^{i}_{32} = K_i$:

Code modifié de la WhiteBox permettant de calculer les clé de tour
(les lignes ajoutées sont identifiées par un '!')

```
def _cipher(self, M, d):
    if not M.size == 64:
        raise AssertionError
    tM1_inv = [16, 31, 32, 63, 48, 75, 0, 91, 71, 30, 79, 62, 87, 74, 95, 90,
70, 23, 78, 55, 86, 73, 94, 89, 5, 22, 21, 54, 37, 72, 53, 88, 8, 15, 24, 47,
40, 67, 56, 83, 69, 14, 77, 46, 85, 66, 93, 82, 68, 7, 76, 39, 84, 65, 92,
81, 61, 6, 13, 38, 29, 64, 45, 80]
    if d == 1:
        blk = M[self.tM1]
        for r in range(16):
            t = 0
!           L = IP(blk[tM1_inv])[:32].ival
            for n in range(12):
                nt = t + 8
                blk[t:nt] = self.KT[r][n][blk[t:nt].ival]
                t = nt

            blk = self.FX(blk)
!           R = IP(blk[tM1_inv])[32:].ival
!           print "K%d : %08X"%(r, R^L)

        return blk[self.tM3]
    if d == -1:
        raise NotImplementedError
```

3.2.3. Calcul de la clé DES

Maintenant que nous sommes en possession des clés de tour et des valeurs intermédiaires du bloc lors de son chiffrement, nous pouvons calculer la clé DES.

Dans DES, les clés de tour sont calculées à partir d'une partie de la clé principale (différente à chaque tour, qu'on appellera sous-clé) et du demi bloc droit (R_{32}) à chiffrer. Pour cela DES utilise plusieurs S-Boxes différentes, dont certaines non injectives (la séries des S-Boxes S_1 à

S_8), c'est à dire qu'à une clé de tour donné et une valeur de R32 donnés, on peut associer plusieurs sous-clés DES différentes.

Il est néanmoins possible de déterminer de manière sûre quelques bits de la sous clé pour chaque valeur de R_{32}. Il suffit alors de chiffrer un blocs choisi aléatoirement, d'obtenir quelques bit de chaque sous clé et donc de la clé principale et de recommencer tant que tous les bits de la clé n'ont pas été découvert.

En pratique, pour retrouver les bits de la clé, il suffit d'inverser les différentes S-Boxes utilisées pour calculer les clé de tour et de conserver deux masques, l'un identifiant les bits nuls de la clé et l'autre les bits à 1.

La routine principal effectuant ce calcul est la suivante :

Code modifié de la WhiteBox permettant de calculer la clé DES
(les lignes ajoutées sont identifiées par un '!')

```
class WB :
    [...]
    def _cipher(self, M, d):
        if not M.size == 64:
            raise AssertionError
        if d == 1:
            blk = M[self.tM1]
!           tM1_inv = [16, 31, 32, 63, 48, 75, 0, 91, 71, 30, 79, 62, 87, 74, 95, 90,
70, 23, 78, 55, 86, 73, 94, 89, 5, 22, 21, 54, 37, 72, 53, 88, 8, 15, 24, 47, 40, 67,
56, 83, 69, 14, 77, 46, 85, 66, 93, 82, 68, 7, 76, 39, 84, 65, 92, 81, 61, 6, 13, 38,
29, 64, 45, 80]
            for r in range(16):
                t = 0
!               L = IP(blk[tM1_inv])[:32].ival
!               R = IP(blk[tM1_inv])[32:].ival
                for n in range(12):
                    nt = t + 8
                    blk[t:nt] = self.KT[r][n][blk[t:nt].ival]
                    t = nt

                blk = self.FX(blk)
!               K = L ^ IP(blk[tM1_inv])[32:].ival

!               (a, b) = Finv(Bits(K, 32), Bits(R, 32), r)
!               self.kinv |= a
!               self.nkinv |= b

!           print "K ", self.kinv
!           print "nK", self.nkinv
!           print
!           self.OK = (self.nkinv|self.kinv).ival == 0xffffffffffffff
!           assert (self.nkinv&self.kinv).ival == 0
            return blk[self.tM3]

if __name__ == '__main__':
!       wb = WB()
!       wb._cipher(Bits(random.getrandbits(64), 64), 1)
!
!       while not wb.OK :
!           wb._cipher(Bits(random.getrandbits(64), 64), 1)
!
!       k = PC1inv(wb.kinv)
!
!       # Calcul des bits de clé non chiffrant
!       for i, j in enumerate(xrange(7, 64, 8)) :
!           k[j] = (175 >> i) & 1
!
!       print "clé DES : %016x"%int(str(k), 2)
```

L'ensemble du code, les S-Boxes inversées ainsi que le code permettant d'extraire les bits significatif des sous-clés, est disponible dans le fichier `breakWB.py`.

Le résultat affiché (le nombre de clairs aléatoires nécessaire à la récupération de tous les bits de la clé varie d'une exécution à l'autre) :

`K` est le masque contenant les bits à 1 de la clé et `nK` les bits à 0

```
K  0010110110011011001110000000010110001001010101101001100 1
nK 0101001000100000000001000011101001110110001000010000010 0

K  1010110111011011101110000000010110001001110111101001100 1
nK 0101001000100100010001000011101001110110001000010110010 0

K  1010110111011011101110011000010110001001110111101001100 1
nK 0101001000100100010001100111101001110110001000010110010 0

K  1010110111011011101110011000010110001001110111101001100 1
nK 0101001000100100010001100111101001110110001000010110011 0

clé DES : fd4185ff66a94afd
```

Un test avec le fichier compilé original confirme que la clé est valide.

4. Vérification de la seconde partie de la clé

A la lecture du code MIPS correspondant à la vérification de la seconde partie de la clé, on se rend rapidement compte qu'elle fait appel à un périphérique externe USB via les APIs unix `usb_*`.

A l'aide des valeurs comparées aux champs `idProduct` (009d) et `idVendor` (04c1) dans la fonction `vicpwn_init`, on trouve rapidement que le périphérique est une webcam HomeConnect Webcam vicam du constructeur U.S. Robotics (3Com)[1].

Avant la vérification à proprement parler, le firmware de la webcam est initialisé par l'envoi de deux buffer, commençant tous deux par les octets `0xB6`, `0xC3`. Après avoir tenté de désassembler ces deux buffers avec les différents processeurs supportés par IDA et après avoir passé des heures à chercher le type de processeur utilisé par la webcam, nous avons fini par lancer un appel à l'aide sur IRC et twitter.

Ayant commencé le challenge une semaine après sa sortie, nous avons appris de personnes plus avancées que nous que le processeur utilisé était un CY16 de Cypress. Le site de Cypress[2] met à disposition toutes les informations nécessaires à la compréhension de l'architecture (jeu d'instruction, information de codage, BIOS, etc.) ainsi qu'un ensemble d'outils de développement (outils que nous n'avions pas remarqué lors de la résolution du challenge...).

Le processeur CY16 est un processeur 16bit little endian très simple, relativement proche des processeurs ARM par sa simplicité, son mode d'adressage avec incrémentation automatique, ses nombreux registres (14 plus les registres spécialisés PC et SP) et ses instructions conditionnelles.

Son jeu d'instruction est restreint (28 instructions différentes) et ne comporte pas d'instructions complexes (comme la multiplication ou la division), à noter que les instructions de branches peuvent être conditionnelles (`call`, `jmp`, `jmp long`, `ret`).

Une des particularité à noter de CY16 est que l'ensemble des registres (y compris le PC et les flags) sont en fait des emplacements mémoire dont l'adresse est donnée par un registre particulier, le REGBANK, situé lui à une adresse constante. Une autre particularité est que les flags ne sont mis à jour que si les opérations sont effectuées sur 16bits.

Pour plus d'informations sur le CY16, nous vous invitons à lire le (court) manuel mis à disposition par Cypress[3].

En plus du processeur, la webcam comporte un BIOS permettant de charger le firmware de la webcam à la volée en envoyant des données commençant par la signature ... `0xC3`, `0xB6` et contenant des pseudo-instructions détaillées dans le manuel du BIOS[4]. C'est aussi le BIOS qui permet les interactions entre le code CY16 et l'extérieur grâce à des interruptions, elles aussi détaillées dans le manuel[4].

La vérification à proprement parler de la deuxième partie du serial est faite en trois étapes, la clé étant découpée en 3 sous parties, converties en nombres à l'aide de la formule suivante :

[1] http://cateee.net/lkddb/web-lkddb/USB_QUICKCAM_MESSENGER.html
[2] http://www.cypress.com/?rID=14316
[3] http://www.cypress.com/?docID=14345
[4] http://www.cypress.com/?docID=14346

$$K_i = \text{int}(K_{str}[i*2:(i+2)*4], 16) \ \&\ 0xFFFFFFFF$$

Chaque partie est alors vérifiée à l'aide de trois *layers* différents interagissant avec le firmware chargé dans la webcam et déchiffré à l'aide de la sortie du *layer* précédent (à l'aide d'un algorithme de type vigenere puis à l'aide de RC4).

4.1. Outils

Etant donné le jeu d'instructions restreint de CY16 et l'absence d'outils d'analyses performants pour ce processeur, nous avons rapidement codé un désassembleur ainsi qu'un émulateur pour ce processeur en python (disponible dans le fichier emuCY16.py).

Le désassembleur ne se contente pas de faire un désassemblage linaire du code comme le ferait objdump mais extrait le CFG du programme et le représente sous la forme d'un graph coloré, comme le ferait IDA Pro, basé sur un code de Baboon[1].

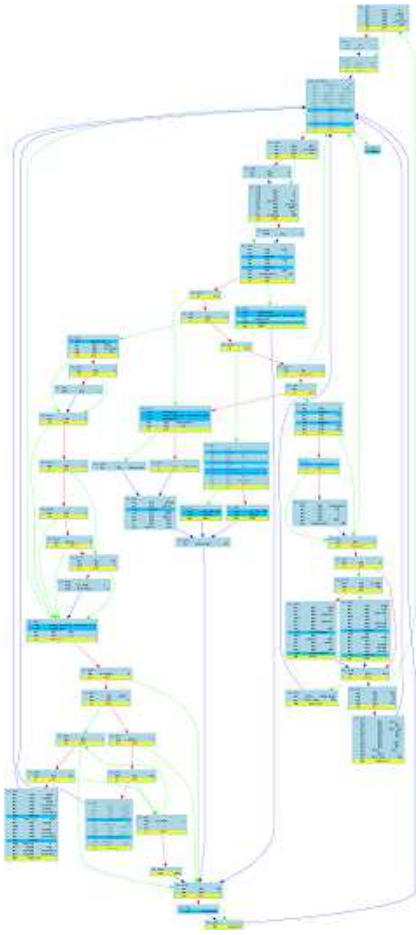
De plus il supporte la définition de labels globaux (pour les cases mémoires) ainsi que locaux (alias des registres dans un intervalle d'adresse donné).

Enfin la représentation sous forme de graph étant réalisé avec graphviz et le format SVG, il est possible de faire des recherche de texte directement depuis un navigateur.

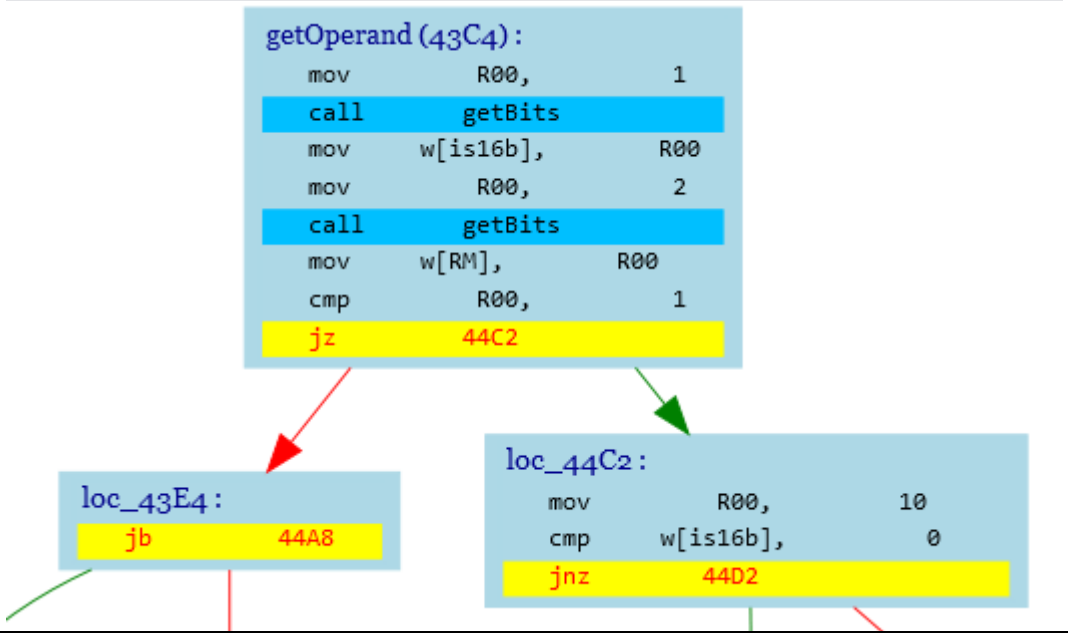
A noter que le firmware utilise des instructions non documentées de CY16, l'analyse du code et la structure de l'instruction (proche de celle des instructions `stc` et `clc`) permet rapidement de déduire que ces instructions sont de simples alias à `stc` et `clc`. Leur existence provient surement d'un décodage *paresseux* de CY16 que ne vérifie que les bits permettant d'identifier l'instruction courante, sans vérifier l'ensemble de l'instruction.

[1] http://baboon.rce.free.fr/index.php?post/2010/10/14/Pourquoi-le-libre-c-est-nul-sauf-le-python

CFG de la fonction principale du firmware



Détail d'une fonction



L'émulateur ne se contente pas d'émuler les instructions mais émule aussi les différentes interruptions utilisées par le firmware, les transfert USB et enfin le chargement du firmware (les fameuses données commençant par la signature C3B6) ce qui lui permet d'émuler entièrement la webcam ainsi que les APIs de gestion USB du noyau linux.

L'émulateur comporte une interface de débogage en ligne de commande permettant de poser des points d'arrêts à l'exécution, l'écriture ou la lecture ; d'exécuter le code pas à pas, en passant dans ou à travers les appels à des routines ; de lire la mémoire du firmware, les registres ; d'enregistrer la suite des instructions exécutées et enfin de piloter l'exécution symbolique du code.

Vue classique de l'interface de débogage

```
Breakpoint at address : 43C4
        R00 : 0000      R01 : 8000
        R02 : 4058      R03 : 0014
        R04 : FFF4      R05 : A000
        R06 : 0000      R07 : 0000
        R08 : 406E      R09 : 4076
        R10 : 4126      R11 : 0000
        R12 : 0000      R13 : 0004
        R14 : 0000      R15 : 03F8
        pc  : 43C4      flags : ZF=1 SF=0 CF=0 OF=0 IF=0

R00 : ((((((b[w[406E]] >> 0001) & 7FFF) >> 0001) & 7FFF) >> 0001) & 0001)
R01 : 8000
R02 : 4050
R03 : 0014
R04 : (w[404C] & FFFE)
R05 : (w[412A] & FFF0)
R06 : ((((((b[w[406E]] >> 0001) & 7FFF) >> 0001) & 7FFF) >> 0001) & 0001)
R07 : 0000
R08 : 4062
R09 : 405A
R10 : 411A
R11 : 0000
R12 : ((w[4144] >> 0004) & 000F)
R13 : (((((((b[w[406E]] >> 0001) & 7FFF) >> 0001) & 7FFF) >> 0001) & 7FFF) >> 0001)
& 0007)
R14 : ((((((((((((((b[w[406E]] >> 0001) & 7FFF) >> 0001) & 7FFF) >> 0001) & 7FFF) >>
0001) & 7FFF) >> 0001) & 7FFF) >> 0
001) & 7FFF) >> 0001) & 0001)
R15 : w[4070]


              getOperand : 07C0 0001  mov  R00, 1

>>> di 43c4 3
              getOperand : 07C0 0001  mov  R00, 1
  ->          43C8 : AF9F 4268        call getBits
              43CC : 0027 412E        mov  w[is16b], R00

>>> dw 4050 5
        0  1 2  3 4  5 6  7 8  9 A  B C  D E  F

4050 :  0000 0000 0000 FFF2 A000
```

L'exécution symbolique peut être activée lors du débogage et est effectuée en parallèle de l'exécution effective du code. Les symboles ne sont associés qu'aux registres, quand une case mémoire est écrite, le symbole de la valeur est affiché associé à l'adresse mémoire mais n'est pas stocké, ce qui permet d'avoir une vision de plus haut niveau de l'exécution du code (seuls les accès mémoires étant enregistrés) :

Exemple de log de l'exécution symbolique du code.
Les suites de ((X >> 1) & 0x7FFF) viennent du fait que les décalages CY16 sont signés et constants

```
42B4 : w[PC] = (w[PC] + (((w[PC.low] + 0010) >> 0003) & 1FFF))
42BC : w[PC.low] = ((w[PC.low] + 0010) & 0007)
44A2 : w[curAddr] = ((((((((((((b[w[406E]] >> 0001) & 7FFF) >> 0001) & 7FFF) >>
0001) & 7FFF) >> 0001) & 7FFF) >> 0001)
 & 7FFF) >> 0001) & 0003) | ((((b[w[406E]] & 003F) << 0008) | b[w[406E]]) << 0002))
41F4 : w[405E] = (w[405E] + 0001)
41F4 : w[4060] = (w[405C] + 0001)
41F4 : w[4062] = (w[405A] + 0001)
41F4 : w[4064] = (w[4058] + 0001)
41F4 : w[4066] = (w[4056] + 0001)
422A : w[4062] = 0000
41AC : w[406E] = ((w[curAddr] - (w[curAddr] & FFF0)) + 4020)
442A : w[OperandV] = w[w[406E]]
4382 : w[V_R03] = w[OperandV]
```

Un moteur de simplification des équations (basique) a été codé afin de réduire la taille des expressions.

Au total, l'ensemble de ces outils représente à peu près 2000 lignes de code python.

4.2. Etude du code du firmware

Une fois ces différents outils développés, l'analyse du firmware devient beaucoup plus simple et nous comprenons que ce firmware n'est autre qu'une machine virtuelle (!!!) les différents *layers* vérifiant les différentes parties de la clé sont en fait des bytecodes interprétés par cette machine virtuelle.

La machine virtuelle comporte 15 registres de 16bits, 2 registres de flags et un pointeur d'exécution de 19bits ; elle est capable d'adresser 64Ko de ram.

La taille de la mémoire adressable par la VM étant la même que celle adressable par CY16, l'ensemble de la mémoire de la VM ne peut être présent dans le firmware. Pour contourner ce problème, la mémoire de la VM est stockée dans la mémoire de l'exécutable principale MIPS (dans la variable globale `ram`), le firmware ne contenant en cache que 5 buffers de 16 octets alignés sur 16 octets. Lorsque la VM tente d'accéder à une adresse mémoire qui n'est pas en cache, le firmware renvoie un code particulier par USB à l'exécutable MIPS qui se charge alors de synchroniser le cache par une autre requête USB.

Toutes les instructions sont conditionnelles et peuvent utiliser l'un ou l'autre jeu de flags, si la, une des particularité de ces instructions étant qu'elles sont alignés sur un bit (d'où la taille du pointeur d'exécution de 19 bits : 16 bits pour identifier l'octet, 3 bits pour le bit).

Structure générale des instructions					
0	1 - 9	10 - 13	14		
flagsUsed	condition	opcode	updateFlags	[operand1]	[operand2]

L'octet exprimant la condition a la même structure que ceux utilisés par CY16[1], l'octet étant directement utilisé pour modifier un saut conditionnel dans le code de la VM.

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

Le champ flagsUsed identifie le jeu de flags à utiliser et le champ updateFlags sert à définir si les flags vont être mis à jour après l'exécution de l'instruction ou non.

Les opérandes disponibles sont du même type que celles de CY16, avec le même traitement spécifique pour le registre de pile R15 lorsqu'il est utilisé dans un accès mémoire indirect (bien que R15 ne soit jamais utilisé dans les différents *layers*), seule une opérande a un comportement différent. Cette opérande (identifiée par `HashMemAcc` dans `VMDiss.py`) est un accès mémoire indexé (adresse + registre), la valeur lue ou écrite étant chiffrée respectivement après lecture ou avant écriture à l'aide d'une clé calculée à partir d'une constante et de la valeur du registre (et donc non prévisible lors du désassemblage).

4.3. Bruteforce de la clé

Le code des *layers* interprété par le firmware est complexe et comporte de nombreuses instructions qu'il est difficile d'analyser, du fait de l'existence des instructions conditionnelles, des deux jeux de flags, de la simplicité des instructions (qui implique une plus grande quantité de code pour un résultat équivalent). La solution la plus simple est donc de bruteforcer les différentes sous-parties de la seconde partie de la clé, l'espace maximum des sous-parties étant de 32bits.

L'émulateur CY16 permettrait en théorie ce bruteforce, néanmoins la vérification de la première sous partie de la clé prend plusieurs minutes. Nous avons donc décidé de convertir les instructions de la VM en code C à l'aide d'un code en python d'approximativement 500 lignes.

Le code C produit est particulièrement illisible mais permet de retrouver les différentes sous parties de la clé en quelques minutes.

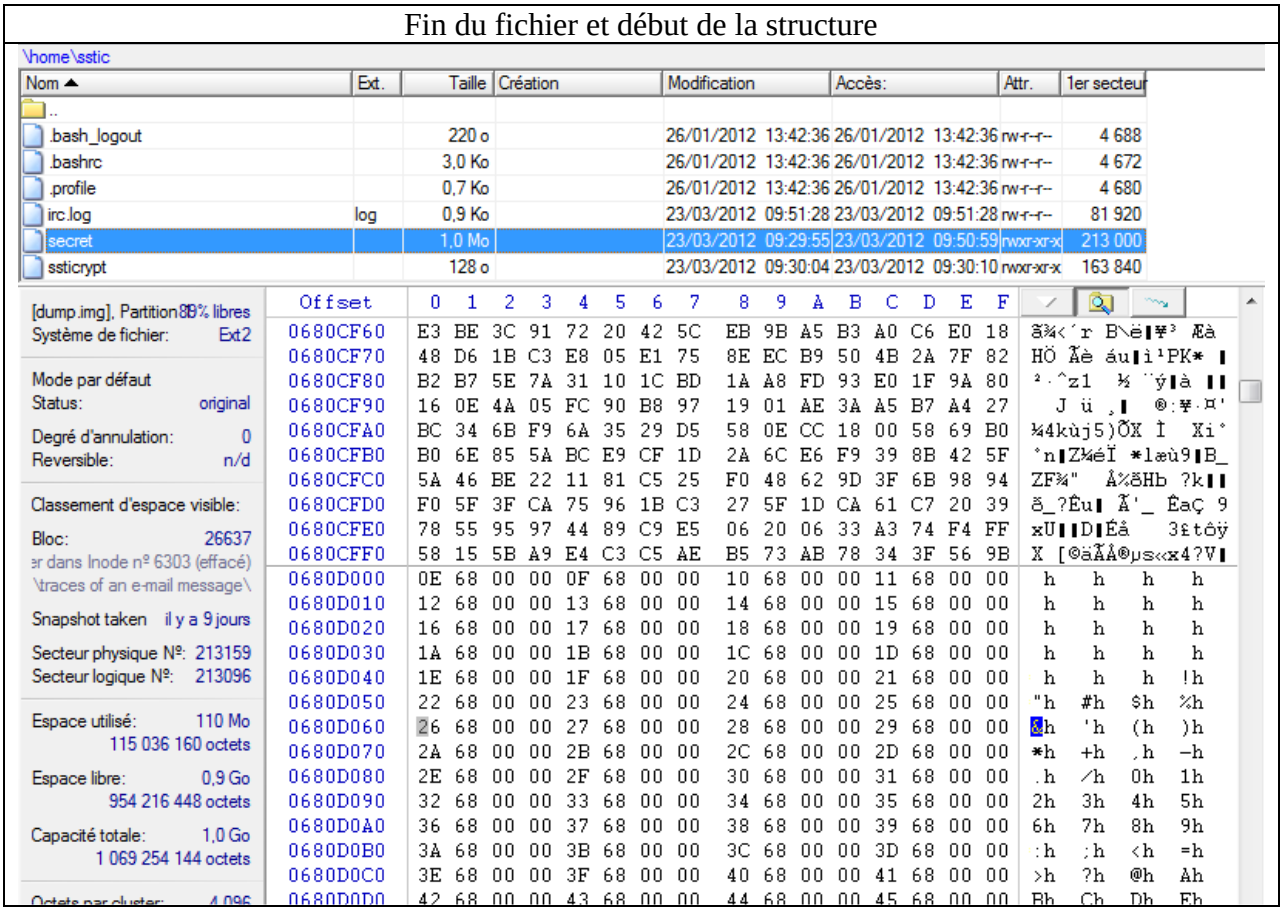
Nous sommes maintenant en possession de l'intégralité de la clé : fd4185ff66a94afde5df94e3f63d8937

5. Récupération de l'adresse mail

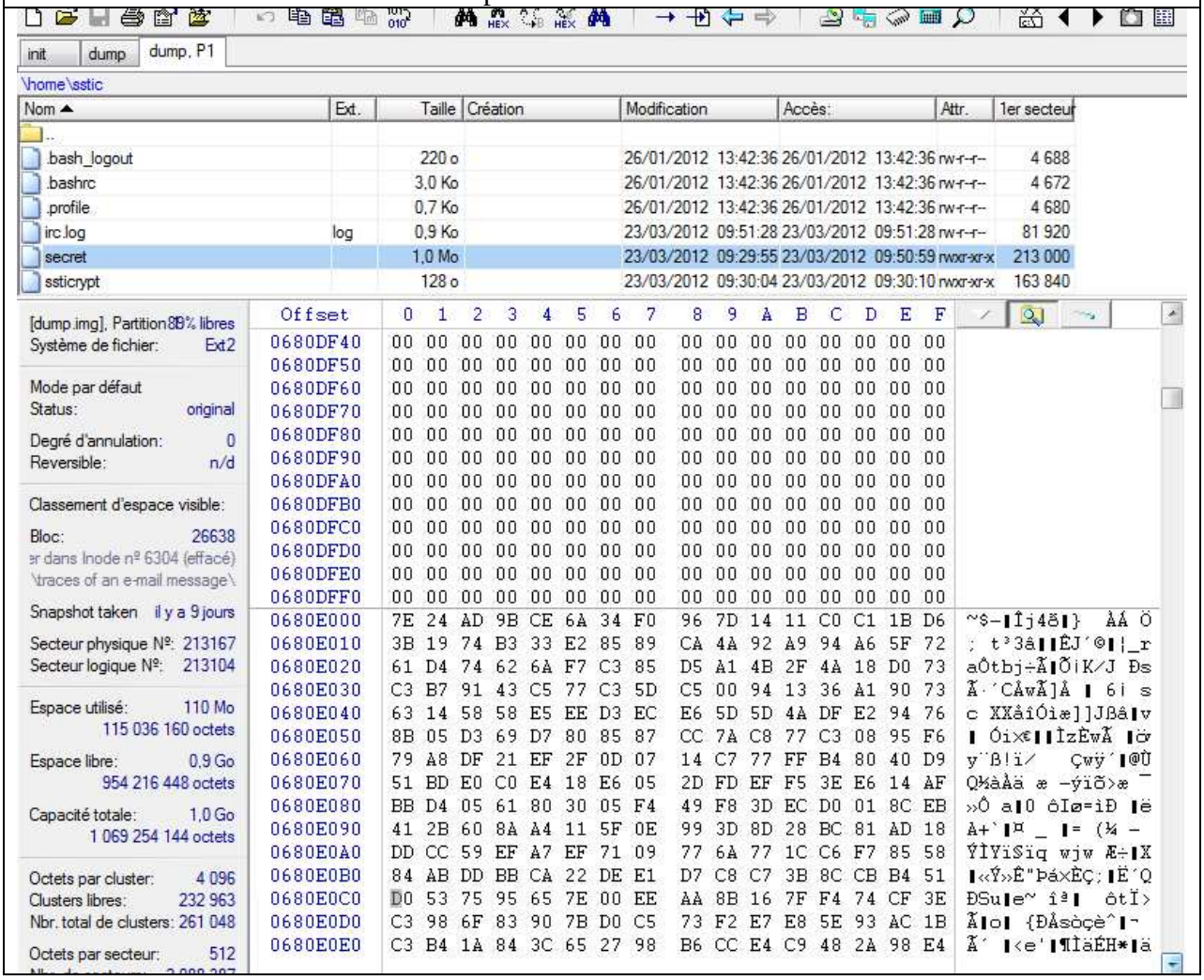
Un dernier obstacle freine la récupération de l'adresse mail, le fichier secret est en effet corrompu, même après la réparation du système de fichier à l'aide de fsck. La fin du fichier est constitué d'octets nuls, au lieu d'octets aléatoires attendus. Ce qui est confirmé par le MD5 du chiffré situé dans les 16 premier octet du fichier.

Pour récupérer la totalité du fichier, nous avons utilisé WinHex qui permet d'explorer l'arborescence de l'image disque tout en affichant les données hexadécimales de l'image correspondantes. Il a alors été possible de reconnaitre à la suite du début du fichier une structure semblant être liée au système de fichier suivie d'autres données aléatoires qui se sont révélé être la fin manquante du fichier.

Fin du fichier et début de la structure



Reprise du fichier chiffré



Nous avons alors pu déchiffrer le fichier secret.py (à l'aide du script python `decrypt_final.py`) qui s'est révélé être une autre image disque. Celle ci contient un film au format AVI représentant un chien déguisé en homard[1] et dans laquelle est incrusté l'adresse mail à contacter.

[1] http://www.youtube.com/watch?v=A3uF1-I5uzA

Capture de la vidéo finale obtenue

6. Remerciements

L'auteur tient à remercier les personnes suivantes :

- Florent Marceau, Fabien Perigaud et Axel Tillequin pour ce challenge extrêmement enrichissant, bien qu'il lui ait couté de nombreux cheveux et heures de sommeil.
- Oppida pour lui avoir laissé le temps de travailler sur ce challenge.
- Ivanlef0u et indirectement Yoann Guillot pour l'avoir informé que le processeur de la webcam était un CY16.
- Jean Sigwald pour avoir supporté ses accès de rage sur IRC (difficile de partager sa frustration en rapport avec un challenge qui est encore ouvert avec d'autre personnes que celles qui l'ont déjà résolu ☺)
- Son co-bureau, Luc, pour ses conseils précieux et avisés ("T'as essayé de faire *pomme* + *c* ?", "utilise xcode pour désassembler le CY16, ca ira mieux !"), son support sans faille et l'ambiance générale de travail.
- Eric Freyssinet et Nicolas Brulez pour leur relecture.