SSTIC CHALLENGE 2013

Solution et démarche

03/04/2013

Haithem El abed

0) Introduction

Le challenge consiste à analyser une trace réseau. L'objectif est d'y retrouver une adresse e-mail (...@challenge.sstic.org).

L'examen avec la commande linux « file » confirme la nature du fichier

$file dump.bin

dump.bin: tcpdump capture file (little-endian) - version 2.4 (Ethernet, capture length 65535)

On va donc examiner la trace réseau avec l'outil open source « wireshark »

I) Analyse de la trace

1) Analyse des flux

Un examen visuel révèle la présence de 4 flux, une seule direction est conservée pour tous les flux :

- Le premier flux reconstruit est un message qui énonce le premier défi :

Déchiffrer Un fichier envoyé par le protocole FTP contenu dans la trace réseau. La clé AES est envoyée dans deux canaux cachés, 1 bit est dans un canal caché temporel concaténé avec 3 bits de canal caché non temporel. Le checksum de l'archive déchiffrée est donné ainsi que le vecteur d'initialisation (IV).

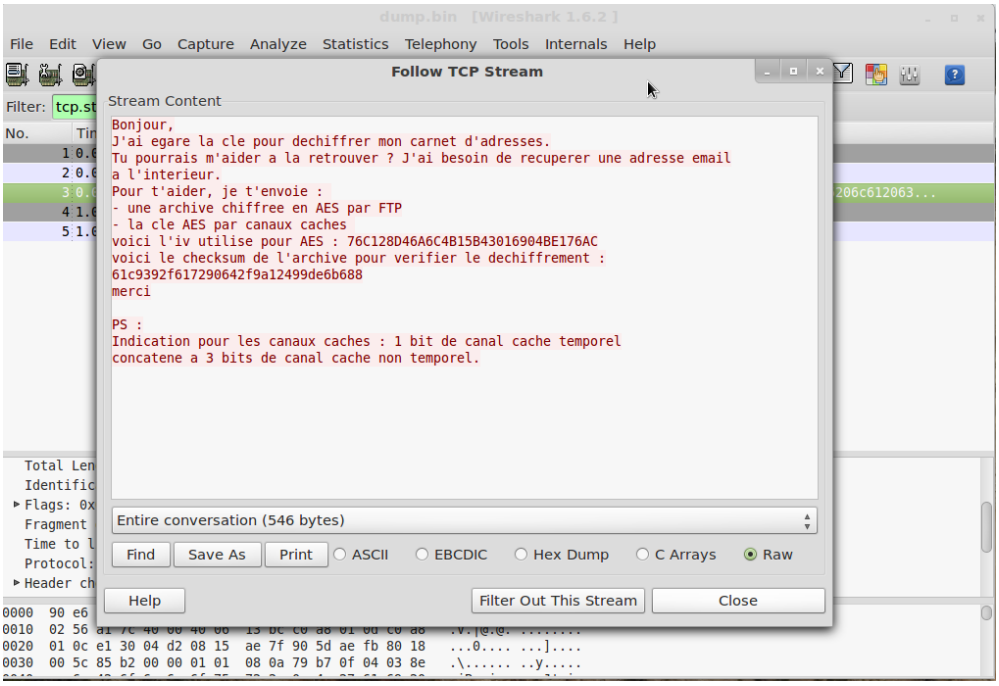


Figure 1

- Le deuxième flux est une série de 65 ping (ICMP echo request), qu'on va étudier plus en détail par la suite.

- Le troisième flux reconstruit contient les commandes FTP envoyées

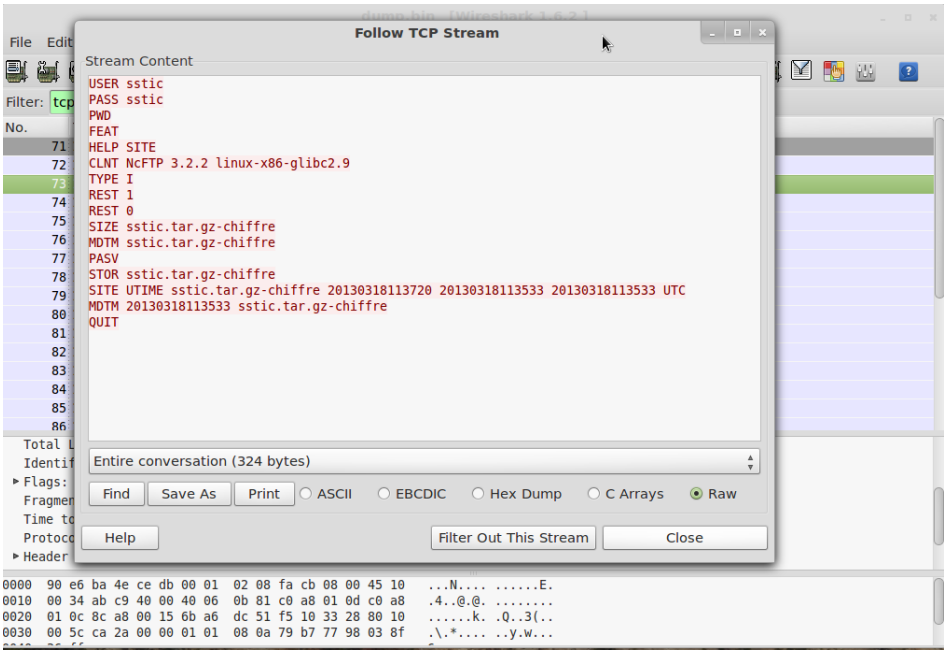


Figure 2

C'est un échange FTP classique avec les commandes de login, pwd, etc… On note toutefois deux informations : le nom du fichier « sstic.tar.gz-chiffré » et une indication sur le système d'exploitation et la version de la librairie C : linux-x86-glibc2.9 (qui servira dans la dernière étape)

- Le quatrième flux permet de récupérer le fichier chiffré tel qu'il a été transmis, en le sauvegardant à partir de wireshark avec l'option « raw ».

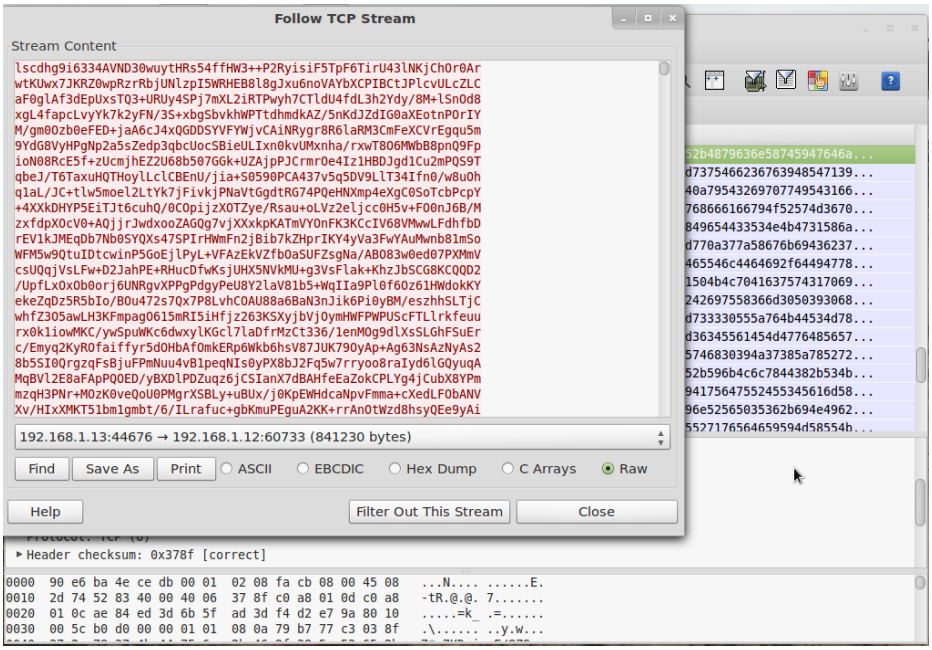


Figure 3

Ce fichier ne contient que des valeurs ascii, et ne peut être le produit d'un chiffrement AES, on fait alors l'hypothèse que c'est du codage de base 64. On le décode pour obtenir le fichier chiffré.

2) Recherche des canaux cachés et la clé AES

La première hypothèse qu'on va faire est que les canaux cachés se trouvent dans la série de 65 ping qui n'ont à priori aucune autre utilité. Ceci colle à peu prés avec les 4bits en total par paquet ce qui donne à priori une clé de taille 256 bits.

En examinant les différents champs dans les paquets ICMP on relève les candidats possibles. On se sert de la fonction de customisation de Wireshark pour ne visualiser que les champs qui nous intéressent comme illustré dans la Figure 4, On utilise ensuite la fonction d'export pour récupérer les valeurs dans un format exploitable par un programme de tabulation comme Openoffice Calc.

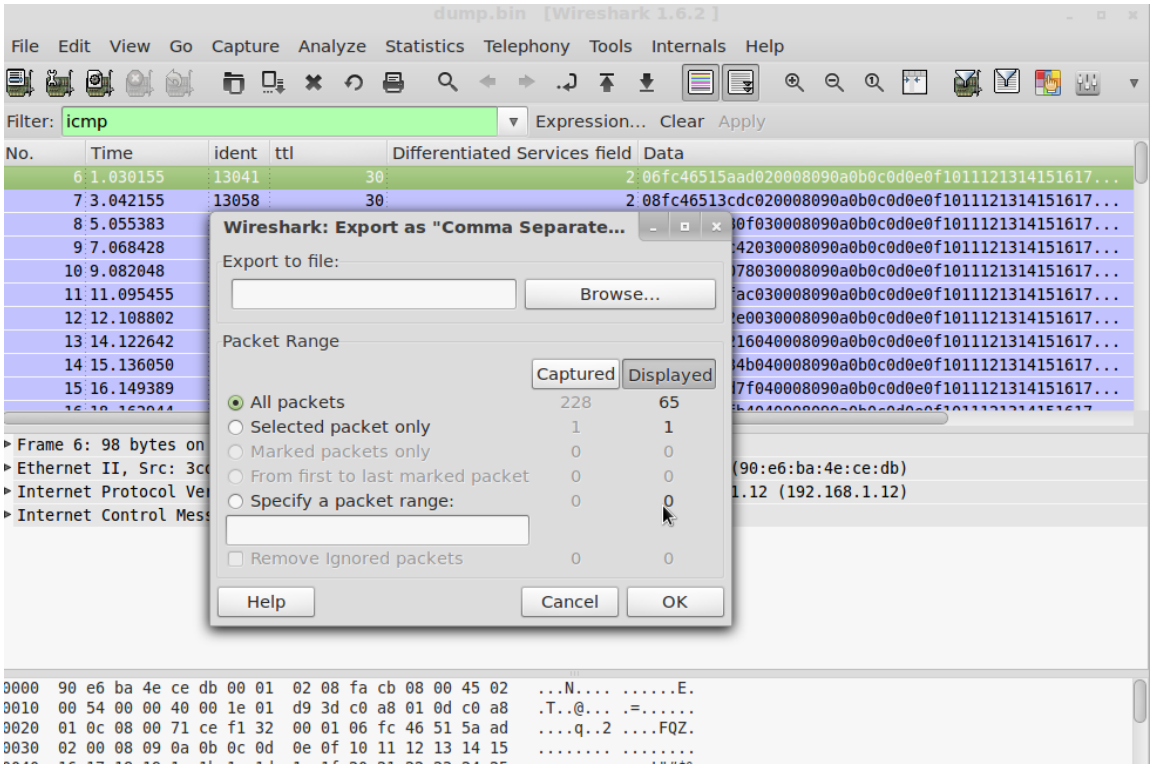


Figure 4

On procède alors à notre analyse en utilisant openOffice Calc, la Figure 5 illustre cette analyse :

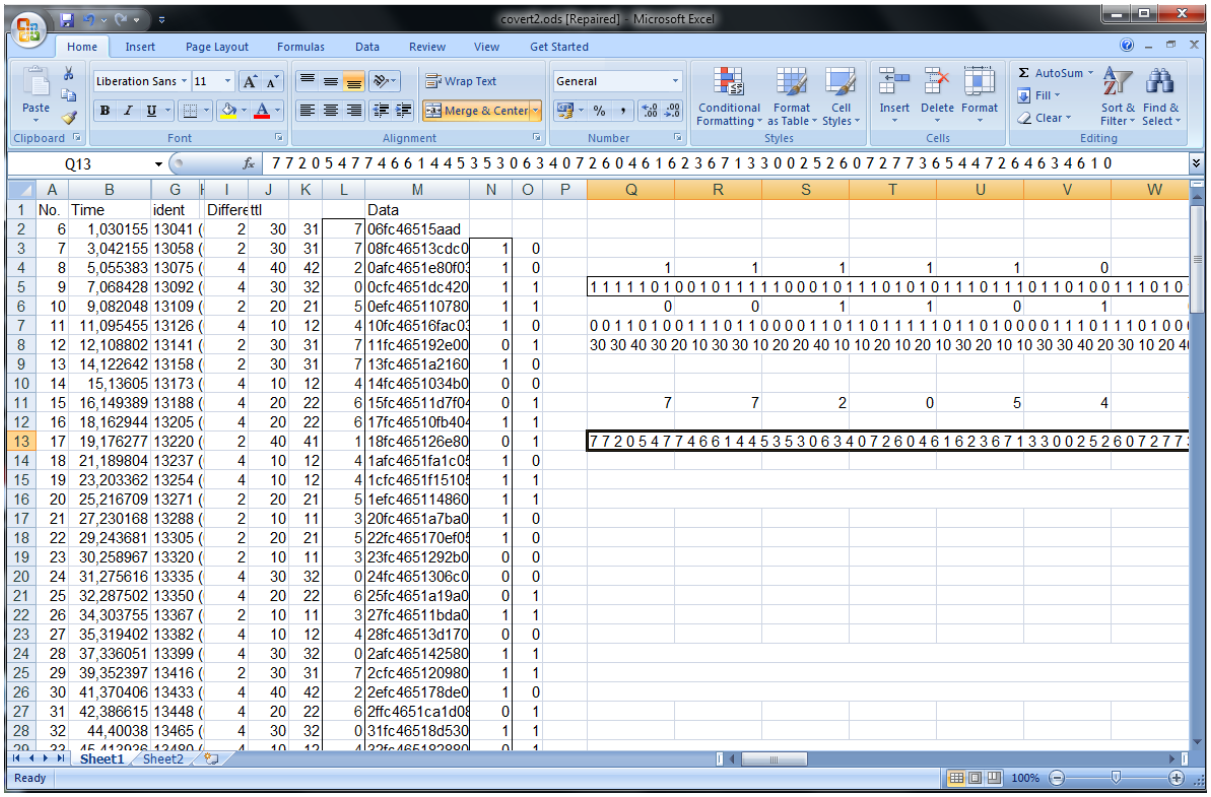


Figure 5

Le temps inter-arrivés des paquets oscillent autour de 1 et 2 secondes, la colonne N capture cette information, d'où l'intérêt du 65ème paquet (pour avoir 64 bits). C'est notre candidat pour le canal temporel à 1 bit.

On note au passage que le champ identité croit avec des sauts qui oscillent entre deux valeurs en capturant cette information on récupère exactement le même canal temporel, ceci est peut être dû à la façon dont les rédacteurs du challenge ont généré ces paquets (en supprimant des paquets intermédiaires pour créer le canal temporel). Le champ data aussi contient presque une copie du canal temporel (à une erreur près).

Pour les canaux non-temporels on remarque, le champ tos (colonne I) qui prend deux valeurs ce qui donne 1 bit d'information et le champ TTL (colonne J) qui prend 4 valeurs ce qui donne 2 bits d'information. On combine les deux canaux dans la colonne K et puis une division modulo 8 pour obtenir des valeurs variant entre 0 et 7.

On transpose alors les deux colonnes L et N et on « merge » les cellules pour pouvoir les exporter facilement dans notre programme python qui va tester les différentes permutations possibles.

Une première tentative qui consiste à comparer le checksum md5 des données déchiffrées à la valeur fourni n'aboutit à aucun « hit ». On se sert alors du fait que le fichier initial en clair était en format gz, ce qui nous permet de tester les deux premiers octets déchiffrés et en même temps accélérer la recherche puisqu'on ne déchiffre que le premier bloc de 128 bit (Taille fixe d'AES).

La Figure 6 montre le program python qui met en œuvre cette dernière approche.

findKey.py

```
import binascii
from Crypto.Cipher import AES
import itertools

chan1=[1,1,1,1,1,0,1,0,0,1,0,1,1,1,1,1,0,0,0,1,0,1,1,1,0,1,0,1,0,1,1,1,0,1,1,1,0,1,
1,0,1,0,0,1,1,1,0,1,0,1,0,1,1,1,1,1,0,1,0,0,1,1,1,0]
inv_chan1=[ ~i & 0x1 for i in chan1]
chan2=[7,7,2,0,5,4,7,7,4,6,6,1,4,4,5,3,5,3,0,6,3,4,0,7,2,6,0,4,6,1,6,2,3,6,7,1,3,3,
0,0,2,5,2,6,0,7,2,7,7,3,6,5,4,4,7,2,6,4,6,3,4,6,1,0]

iv=binascii.a2b_hex("76C128D46A6C4B15B43016904BE176AC")
ciphertext='\x96\xc7\x1d\x86\x0fb\xeb}\xf8\x01SC\xdfL.\xca'

chan1_perm=[[i<<3 for i in inv_chan1], [i<<3 for i in chan1] ]
chan2_perm=itertools.permutations([0,1,2,3,4,5,6,7])

for perm2 in chan2_perm:
     co2=[perm2[i]  for i in chan2]
     for co1 in chan1_perm:
        half=[x+y for x,y in zip(co1,co2)]
        keyBytes=[(half[i]<<4) + half[i+1] for i in range(0,63,2)]
        key="".join([chr(i) for i in keyBytes])
        cipher= AES.new(key,AES.MODE_CBC,iv)
        cleartext=cipher.decrypt(ciphertext)
        if (cleartext[0:2] == '\x1f\x8b'):
                print "Found possible Key"
                possiblekey="".join([binascii.hexlify(chr(i)) for i in keyBytes])
                print "key: " , possiblekey
```

Figure 6

Cette approche fournit 3 possibilités de clé :

$python findKey.py

Found possible Key

key: 00125c08c7f64453dba7b42097a4f671b706b32a1d97209083f54401f4fb476a

Found possible Key

key: dd8cf2d52e69aafb734e3acd0e4a69e83ed93bc4870ecd0d5b6faad86a63ae94

Found possible Key

key: 11352f19f6e07724acd6c751b6d7e063c610c45d3ab651b194e27713e7ec760d

On les teste alors en déchiffrant le fichier complet pour trouver la bonne clé.

dd8cf2d52e69aafb734e3acd0e4a69e83ed93bc4870ecd0d5b6faad86a63ae94

Avant le chiffrement l'archive initiale a subit un bourrage de 6 octets ce qui explique pourquoi la première tentative se basant sur le checksum a échouée. En supprimant le bourrage on vérifie bien le bon checksum.

II) Analyse de l'archive

La décompression de l'archive produit un ensemble de 4 fichier :

archive/

archive/smp.py

archive/data

archive/s.ngr

archive/decrypt.py

0) Analyse du programme python decrypt.py

Le programme python ouvre un équipement externe, et l'utilise pour déchiffrer le fichier data, ceci est fait par bloc de 224 octets, le tableau smp (du fichier smp.py) est envoyé à l'équipement avec chaque itération, et on récupère des donnés de l'équipement. Une clé doit être fournie au programme qui elle aussi est envoyée à l'équipement avec chaque itération.

Une fois que toutes les donnés ont été récupérées de l'équipement, le programme les décode en base 64 et vérifie le checksum.

Il nous faut donc identifier ce mystérieux équipement. *L'objectif est de trouver l'algorithme de déchiffrement ainsi que la clé du déchiffrement pour déchiffrer le fichier data*

1) Analyse du fichier s.ngr

C'est un fichier généré par Xilinx Synthesis Technology (XST), On utilisera le programme ISE (de Xilinx) en version d'évaluation, pour visualiser cette représentation graphique de la conception RTL (Register Transfer Level) du système.

On découvre alors l'équipement « S » avec les entrés sorties de contrôle.

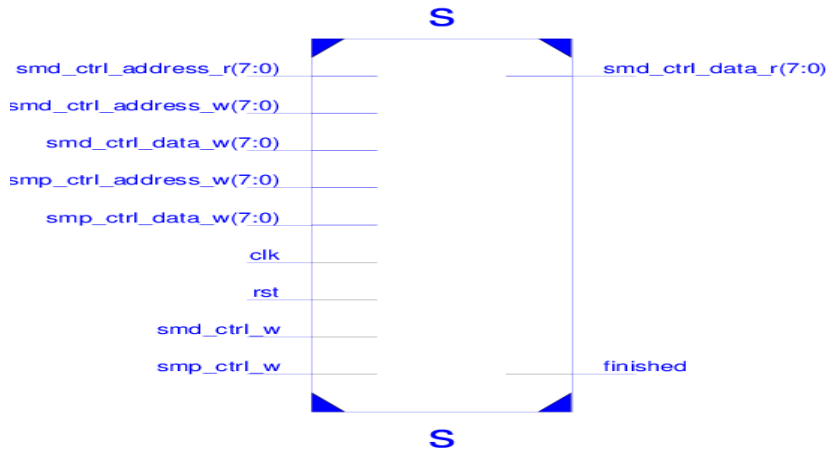


Figure 7

On va donc procéder à une analyse du contenu pour comprendre le fonctionnement de S.

2) Reverse du device S

Le premier zoom dans le device S révèle plusieurs blocs qu'on analysera un par un :

a) Bloc s_m_p

C'est une mémoire de 256 octets (Figure 8), qui recevra le programme à exécuter.

Pour écrire un octet : ctrl_address_w =adresse ; ctrl_data_w= valeur ; ctr_w = active.

Pour lire : address_r= adresse ; data_r = valeur

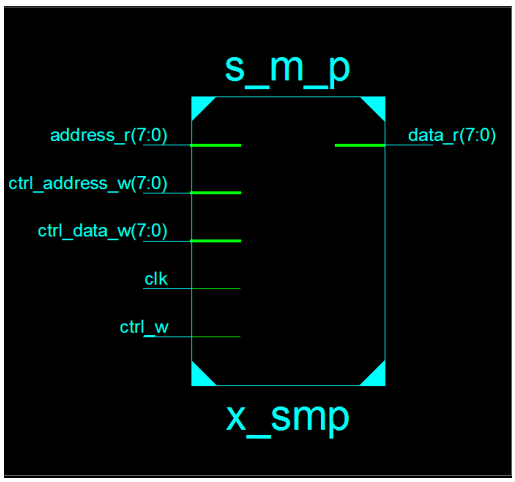


Figure 8

b) Bloc s_m_d

C'est une mémoire de 2x256 octets. Chaque bloc de 256 octets a ses propres lignes de contrôle.

Un bloc servira pour recevoir les donnés smd, l'autre bloc contiendra les résultats de l'exécution.

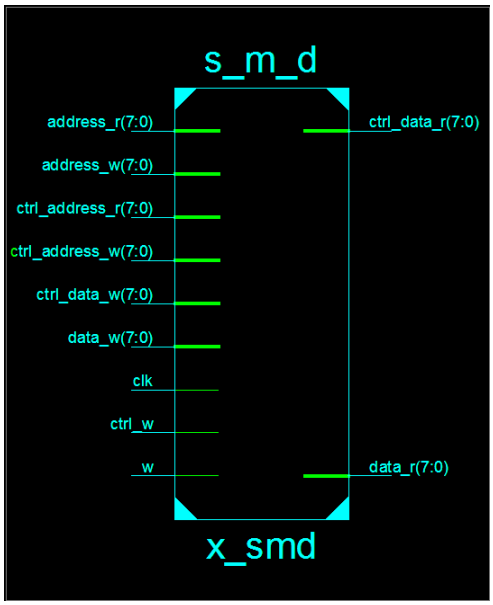


Figure 9

c) Bloc U

C'est une fonction (Figure 10) qui prend en entrée deux octets a et b de données et 5 bits de contrôle c, en fonction des bits de contrôle c elle produit en sortie un octet représentant (a and b), (a or b), (not a), (a <<1) ou bien (a or 0x80). Ce bloc va jouer le rôle d'une UL (Unité logique)

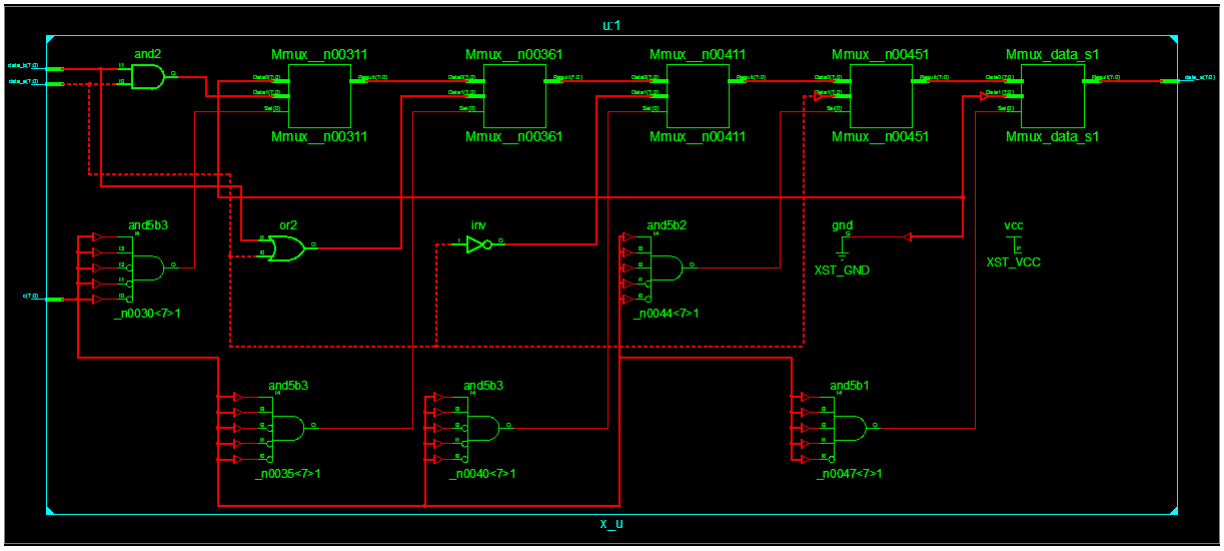


Figure 10

d) Bloc r

C'est une mémoire de 8 octets (Figure 11), 3 bits d'entrés pour la sélection l'adresse, un bit de contrôle lecture écriture. Ce bloc va jouer le rôle de 8 registres.

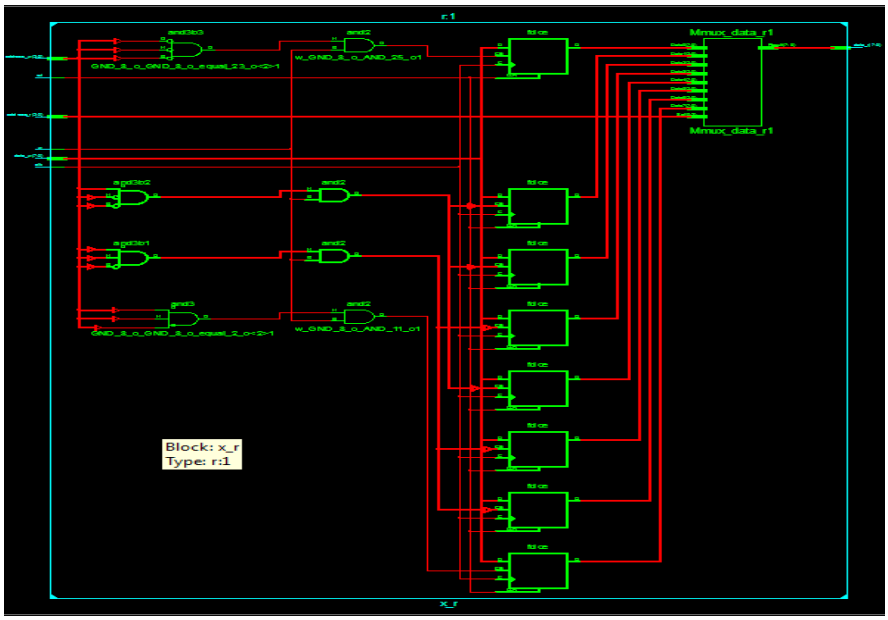


Figure 11

e) Bloc Accu

C'est un registre de 1 octet (Figure 12) qui va jouer le rôle d'accumulateur.

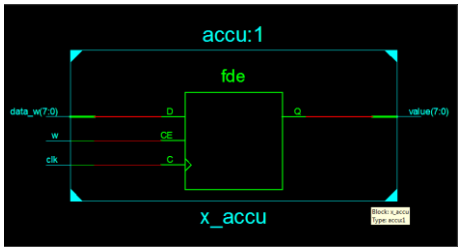


Figure 12

f) Bloc Ip

Ce bloc (Figure 13) contient un registre de 1 octet initialisé à 0, en sortie une incrémentation de 1 de cette valeur. La valeur du registre peut être modifiée en écriture. Ce bloc va jouer le rôle d'un pointeur d'instruction.

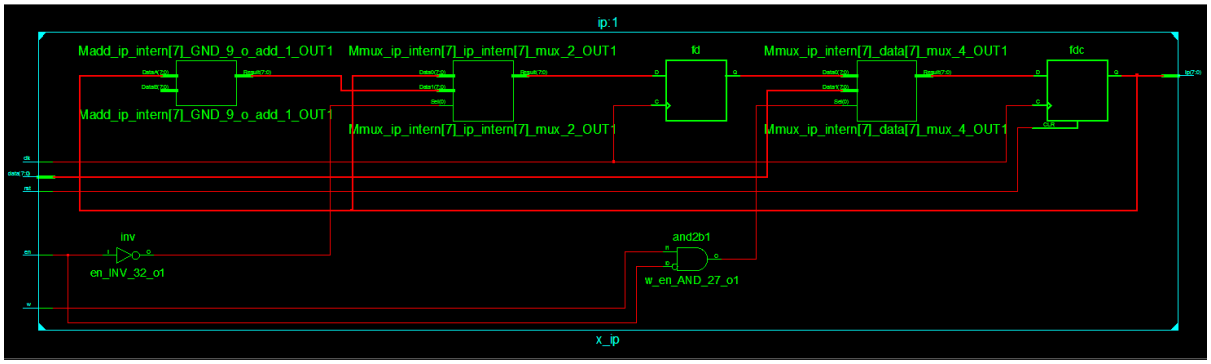


Figure 13

g) Assemblage des blocs

La Figure 14 montre l'interconnexion des différents blocs. Les différentes portes logiques et les multiplexeurs servent au décodage des instructions et le contrôle des différents blocs.

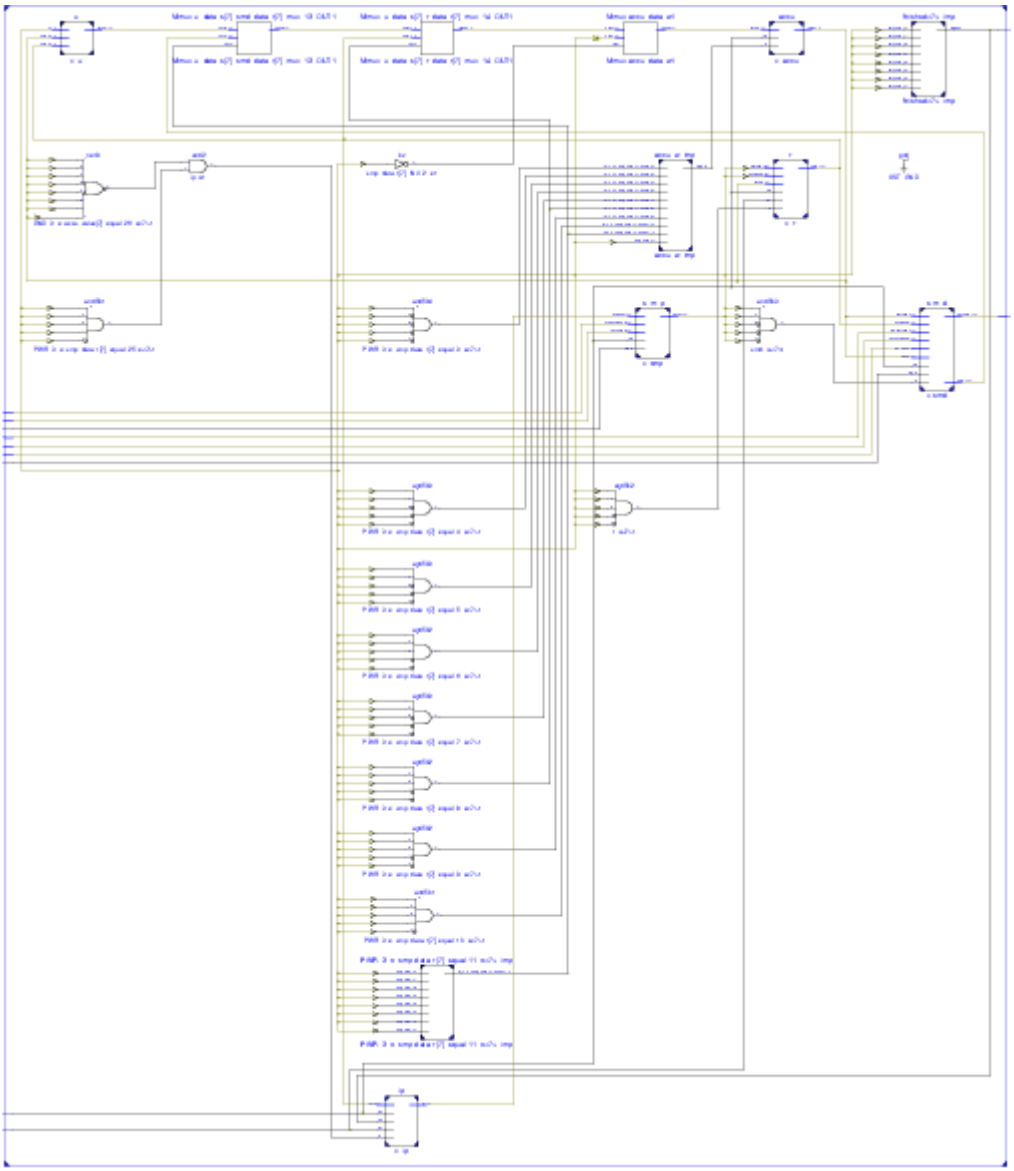


Figure 14

3) Description du fonctionnement de S

Globalement L'équipement S joue le rôle d'un processeur d'instructions (une instruction tient sur un octet). Il a 8 registres, un accumulateur, une unité logique, un compteur programme, une mémoire programme de 256 octets, et une mémoire de donnés de 2x256 octets.

Les mémoires data sont adressables indirectement à travers les registres. L'instruction du saut (jump) est exécuté si la valeur de l'accumulateur est zéro. Il a une instruction dédiée pour l'arrêt de l'exécution et signaler que le résultat de l'exécution est prêt à être lu par l'extérieur.

On n'a donc ce qu'il faut pour désassembler le programme smp et émuler son exécution.

4) Dé-assemblage de smp et émulation

La Figure 15 contient le listing du programme qui va nous permettre de désassembler les 231 instructions du programme smp, le programme désassemblé est listé dans l'Annexe 1.

```
import smp

i=0
for inst in smp.smp:
        print '0x%02x:           0x%02x: ' % (i, inst),
        i = i+1
        if (inst & 0x80 ) == 0:
                value= inst & 0x7F
                print 'mov 0x%02x accu' % value
        else:
                addr= inst & 0x07
                if (inst & 0xF8) == 0xa0:
                        print 'not ea'
                elif (inst & 0xF8) == 0x90:
                        print 'or r[%x] accu' % addr
                elif (inst & 0xF8) == 0x88:
                        print 'and r[%x] accu' % addr
                elif (inst & 0xF8) == 0xd8:
                        print 'shl accu'
                elif (inst & 0xF8) == 0xe0:
                        print 'mask accu'
                elif (inst & 0xF8) == 0xd0:
                        print 'mov smd[accu] accu'
                elif (inst & 0xF8) == 0xc8:
                        print "end"
                elif (inst & 0xF8) == 0xc0:
                        print 'mov accu smd[r[%x]]' % addr
                elif (inst & 0xF8) == 0xb0:
                        print 'mov accu r[%x]' % addr
                elif (inst & 0xF8) == 0xa8:
                        print 'mov r[%x] accu' % addr
                elif (inst & 0xF8) == 0xb8:
                        print 'jmp r[%x]' % addr
                else:
                        print "can't decode"
```

Figure 15

La Figure 16 suivante liste un programme qui émule l'exécution du programme smp sur un bloc de donnés de 224 octets. L'exécution finit au bout de 348067 instructions.

```
import smp

smd = open("data", "rb").read()[0 :  224]
smd="".join((key,chr(len(smd)),smd))
key="".join([chr(i+1) for i in range(16)])
res=[0 for it in range(256)]
IP=0
r=[0,0,0,0,0,0,0,0]
i=0

while smp.smp[IP] != 0xc8:
        jmp=0
        i=i+1
        inst=smp.smp[IP]
        print '0x%02x:           0x%02x: ' % (i, inst),
        if (inst & 0x80 ) == 0:
                accu= inst & 0x7F
                print 'mov 0x%02x accu' % accu
        else:
                addr= inst & 0x07
                if (inst & 0xF8) == 0xa0:
                        accu=(~accu)&0xFF
                        print 'not accu ; accu=0x%02x' % accu
                elif (inst & 0xF8) == 0x90:
                        accu= (r[addr] | accu) &0xFF
                        print 'or r[%x] accu ;  r[%x]= %x, accu=0x%02x' % (addr, addr, r[addr], accu)
                elif (inst & 0xF8) == 0x88:
                        accu= (r[addr] & accu) &0xFF
                        print 'and r[%x] accu ;  r[%x]= %x, accu=0x%02x' % (addr, addr, r[addr], accu)
                elif (inst & 0xF8) == 0xd8:
                        accu= ((accu <<1) & 0xFF)
                        print 'shl accu ; accu=0x%02x' % accu
                elif (inst & 0xF8) == 0xe0:
                        accu= accu  | 0x80
                        print 'mask accu ; accu=0x%02x' % accu
                elif (inst & 0xFF) == 0xd0:
                        print "accu =smd[%2x]= " % accu,
                        accu=ord(smd[accu])
                        print "0x%2x" % accu
                elif (inst & 0xF8) == 0xc8:
                        print "end"
                elif (inst & 0xF8) == 0xc0:
                        res[r[addr]]=chr(accu)
                        print "res[%2x]= %2x" % (addr,accu)
                elif (inst & 0xF8) == 0xb0:
                        r[addr]=accu
                        print 'mov accu r[%x]; accu=0x%x r[%x]=0x%x' % (addr,accu,addr,r[addr])
                elif (inst & 0xF8) == 0xa8:
                        accu=r[addr]
                        print 'mov r[%x] accu; accu=0x%x r[%x]=0x%x' % (addr,accu,addr,r[addr])
                elif (inst & 0xF8) == 0xb8:
                        if (accu == 0x00):
                                IP=r[addr]
                                jmp=1
                                print 'jmp r[%x], r[%x]=%x' % (addr,addr,IP),
                                print [hex(ri) for ri in r]
                        else:
                                print 'DIDNT  jmp r[%x], r[%x]=%x accu=0x%x' % (addr,addr,IP,accu),
                                print [hex(ri) for ri in r]
                else:
                        print "can't decode"
        if (jmp != 1):
                IP=(IP+1)&0xFF
print "END"
print "%d instructions" % i
print res
```

Figure 16

5) *L'algorithme de déchiffrement*

L'analyse de la trace d'exécution produite précédemment ainsi que le programme désassemblé nous permettra de déterminer l'algorithme de déchiffrement qui consiste en un XOR de la clé de 16 octets

avec les blocs de données de 16 octets puis une rotation de chaque octet. La Figure 17 illustre une implémentation en python de l'algorithme de déchiffrement sur un bloc de 224 octets. On compare le résultat obtenu avec le résultat de l'émulation (Figure 16) pour la validation.

```
def rotate (byte):
        val = ((byte << 7) & 0x80) | ( (byte << 5)& 0x40) | ( (byte << 3)& 0x20) | ( (byte << 1)& 0x10) |
( (byte >> 7)& 0x01) | ( (byte >> 5)& 0x02) | ( (byte >> 3 )& 0x04) | ( (byte >> 1) & 0x08)
        return val

key = [chr(i+1) for i in range(16)]
d = open("data", "rb").read()
smd = d[0 : 224]
res=[0 for it in range(256)]

for i in range(0,224):
        r=(ord(smd[i]) ^ ord(key[i&0x0F]))
        res[i]=chr(rotate(r))

print res
```

Figure 17

6) *Détermination de la clé*

On va utiliser le fait que les données déchiffrées est un encodage en base 64, et que les 16 octets de la clé sont réutilisés cycliquement, pour déterminer les clés possibles. La Figure 18 illustre une implémentation en python de cette approche.

```
import binascii

def ro (byte):
        val = ((byte << 7) & 0x80) | ( (byte << 5)& 0x40) | ( (byte << 3)& 0x20) | ( (byte << 1)& 0x10) |
( (byte >> 7)& 0x01) | ( (byte >> 5)& 0x02) | ( (byte >> 3 )& 0x04) | ( (byte >> 1) & 0x08)
        return val


key = [0 for i in range(16)]
d = open("data", "rb").read()

for i in range(16):
        for can in range(256):
           bad=0
           for base in range(0,len(d)-15,16):
                val= ro ( ord(d[base+i])   ^ can )
                if (  ( val > 0x7e ) or (val < 0x20)  ):
                        if (val != 0x0a) and (val != 0x0d):
                                bad=1
           if (bad == 0):
                print "found possible value for key[%x] = %x" % (i, can)
                key[i]=can

print key
print binascii.b2a_hex("".join([chr(i) for i in key]))
```

Figure 18

Ce qui nous donne la clé :

e683dcbc16ef58c665ac23d31e6da125

On a la clé et l'algorithme de déchiffrement, c'est tout ce qu'il nous faut pour obtenir le fichier « atad » et passer à l'étape suivante.

III) Analyse du fichier atad

C'est un fichier postscript, une tentative d'exécution avec ghostscript sera infructueuse. Et produit le message d'erreur :

> *missing '--' preceding script file*
>
> *usage: gs -- script.ps key*

Il faut donc fournir une clé pour pouvoir déchiffrer ce que ce fichier contient. Il ne nous reste plus qu'analyser le script.

On n'a pas accès à un débogueur postscript; on se servira donc de l'interpréteur interactive gs (ghostscript) et de l'éditeur « émacs » qui a un mode postscript et une fonctionnalité utile qui consiste à envoyer une sélection de région arbitraire à l'interpréteur gs (Figure 19)

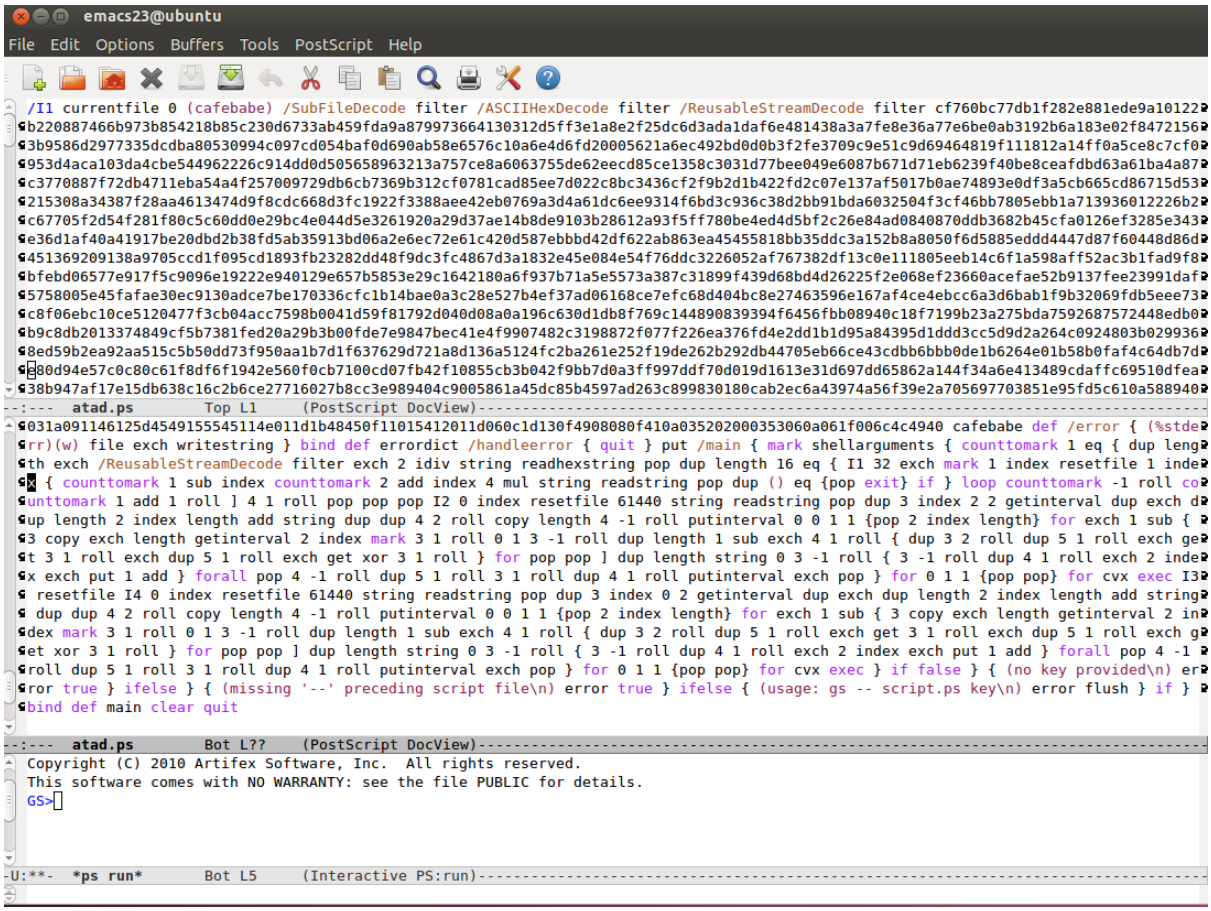


Figure 19

Le script atad contient principalement 4 blocs de donnés binaires I1, I2, I3, I4 et une fonction main.

1) L'analyse de la fonction Main

On utilise l'interpréteur interactif gs pour extraire les 4 blocks de donnés et les mettre dans 4 fichiers séparés qu'on appellera I1, I2, I3 et I4. On procède après à l'analyse de la fonction principale main qui est illustré dans la Figure 20 et Figure 21

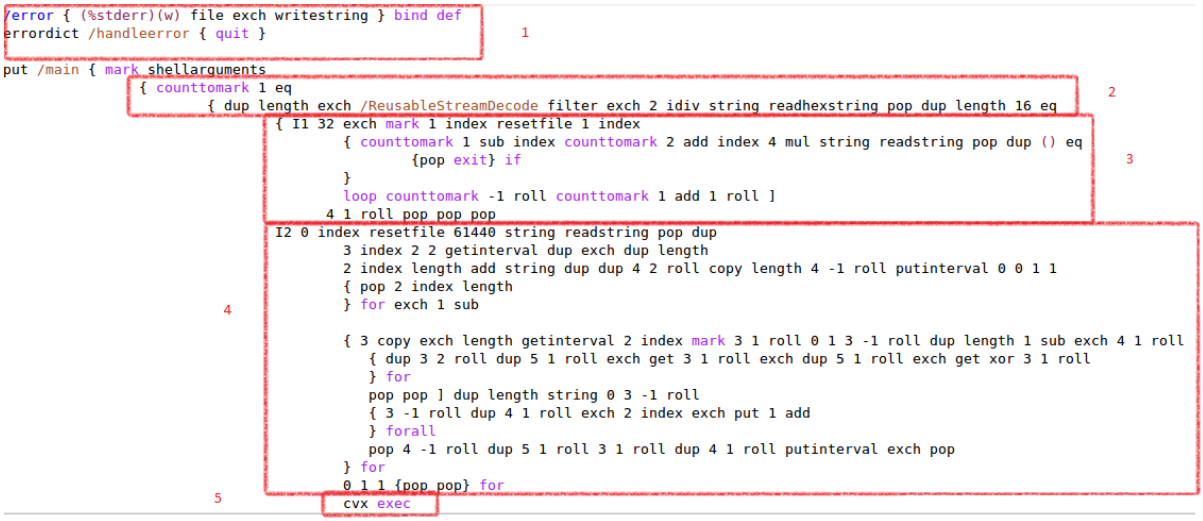


Figure 20

L'encadré 1 de la Figure 20 est une définition de l'impression des messages d'erreurs et le comportement du script en cas d'erreur (quit). La fonction main commence par empiler un mark et les arguments de la commande en ligne. L'encadré 2 teste la présence d'argument (la clé) et teste sa longueur qui doit être égale à 16 octets. Si tout est bon, l'encadré 3 crée un tableau à partir du bloc I1. A ce stade la pile est comme suit :

GS>pstack

[I1 en Tableau]

(Clé de 16 octets)

-mark-

L'encadré 4, récupère l'octet 3 et l'octet4 de la clé (k[2]k[3]) les duplique pour une créer une clé de 4 octets qui servira à déchiffrer le bloc I2.

L'encadré 5, change l'attribut du résultat pour qu'il soit exécutable et l'exécute.

L'encadré 1 de la Figure 21 est exactement le même algorithme de déchiffrement de I2 qui va être utilisé pour déchiffrer I4 et l'exécuter sauf que cette fois c'est l'octet 1 et l'octet2 de la clé (k[0]k[1]) qui sont utilisés pour générer la clé de déchiffrement.

On procède donc à l'analyse de l'algorithme de déchiffrement.

```
I4 0 index resetfile 61440 string readstring pop dup
        3 index 0 2 getinterval dup exch dup length
        2 index length add string dup dup 4 2 roll copy length 4 -1 roll putinterval 0 0 1 1
        {pop 2 index length} for
         exch 1 sub
        { 3 copy exch length getinterval 2 index mark 3 1 roll 0 1 3 -1 roll dup length 1 sub exch 4 1 roll
           { dup 3 2 roll dup 5 1 roll exch get 3 1 roll exch dup 5 1 roll exch get xor 3 1 roll
           } for
           pop pop ] dup length string 0 3 -1 roll
          { 3 -1 roll dup 4 1 roll exch 2 index exch put 1 add
          } forall
          pop 4 -1 roll dup 5 1 roll 3 1 roll dup 4 1 roll putinterval exch pop
        } for
        0 1 1 {pop pop} for
        cvx exec
    } if
                false
            }
        { (no key provided\n) error true
        } ifelse
    }
  { (missing '--' preceding script file\n) error true
  } ifelse

 { (usage: gs -- script.ps key\n) error flush
 }  if
} bind def
main clear quit
```

Figure 21

2) Algorithme de déchiffrement des blocs I2 e tI4

La Figure 22 illustre l'algorithme utilisé pour déchiffrer I2 et I4

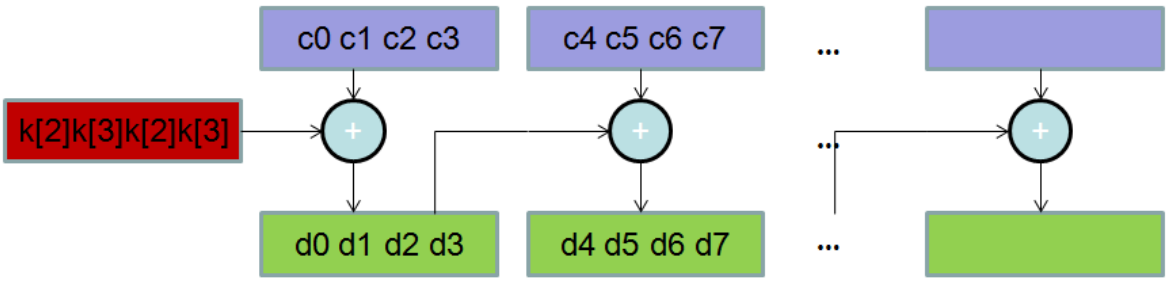


Figure 22

On utilisera le fait que les données déchiffrées sont un script exécutable pour déterminer les clés susceptibles d'être les bonnes. Après quelques essais, il s'avère qu'un « grep » sur l'opérateur « string » identifie les bonnes clés. La Figure 23 contient le programme python utilisé pour déterminer les clés et déchiffrer I2 et I4.

```
from  binascii import b2a_hex

def decr(filename):
  for ka in range(256):
    for kb in range(256):
        bad=0
        key=[ka,kb,ka,kb]
        res=""
        for i in range(0,len(c)-1,4):
                ci=c[i:i+4]
                val=[key[inc] ^ ord(ci[inc]) for inc in range(0,4)]
                key=val
                for tes in range (0,4):
                        res=res+chr(key[tes] )

        if (res.find('string') == -1) :
                bad=1
        if (bad == 0):
          print "found possible pair for ka,kb ka= %x  kb=%x" % (ka, kb)
          out = open(filename+"".join(["_ka_",b2a_hex(chr(ka)),"_kb_",b2a_hex(chr(kb))]),"wb")
          out.write(res)
          out.close()
  return

c = open("I2", "rb").read()
decr("I2_decrypted")
#c = open("I4", "rb").read()
#decr("I4_decrypted")
```

Figure 23

On obtient donc I2_decrypted et I4_decrypted ainsi que les 4 premiers octets de la clé globale de 16 octets :

k[0]=0xba ; k[1]=0xc9 ; k[2]=0xf7 ; k[3]=0xa8

3) L'analyse de I2_decrypted et de I4_decrypted

La Figure 24 est le contenu d'I2 déchiffré. On reconnait une implémentation de l'algorithme de hachage md5. C'est la fonction « calc » qui sera utilisé dans I4 déchiffré pour vérifier les checksums md5.

```
 20 dict begin /T [ 8#32732522170 8#35061733526 8#4410070333 8#30157347356 8#36537007657 8#10741743052 8#25014043023 8#37521512401 8#15140114330 8
8#21321173657 8#37777655661 8#21127153676 8#15344010442 8#37546070623 8#24636241616 8#11155004041 8#36607422542 8#30020131500 8#4627455121 8#351
55543652 8#32613610135 8#221012123 8#33050363201 8#34764775710 8#4170346746 8#30315603726 8#36465206607 8#10526412355 8#25170764405 8#3747372177
0 8#14733601331 8#21512446212 8#37776434502 8#20734373201 8#15547260442 8#37571234014 8#24457565104 8#11367547651 8#36656645540 8#27657736160 8#
5046677306 8#35250223772 8#32473630205 8#442016405 8#33165150071 8#34666714745 8#3750476370 8#30453053145 8#36412221104 8#10312577627 8#25345021
647 8#37444720071 8#14526654703 8#21703146222 8#37773772175 8#20541056721 8#15752077117 8#37613163340 8#24300241424 8#11602010641 8#36724677202
8#27516571065 8#5265751273 8#35341551621 ] def /F [ { c d /xor b /and d /xor } { b c /xor d /and c /xor } { b c /xor d /xor } { d /not b /or c /
xor } ] def /R [ 8#7 8#414 8#1021 8#1426 8#2007 8#2414 8#3021 8#3426 8#4007 8#4414 8#5021 8#5426 8#6007 8#6414 8#7021 8#7426 8#405 8#3011 8#5416
 8#24 8#2405 8#5011 8#7416 8#2024 8#4405 8#7011 8#1416 8#4024 8#6405 8#1011 8#3416 8#6024 8#2404 8#4013 8#5420 8#7027 8#404 8#2013 8#3420 8#5027
 8#6404 8#13 8#1420 8#3027 8#4404 8#6013 8#7420 8#1027 8#6 8#3412 8#7017 8#2425 8#6006 8#1412 8#5017 8#425 8#4006 8#7412 8#3017 8#6425 8#2006 8#
5412 8#1017 8#4425 ] def /W 1 31 bitshift 0 gt def /A W { /add } { /ma } ifelse def /t W { 1744 } { 1616 } ifelse array def /C 0 def 0 1 63 { /i
 exch def /r R i get def /a/b/c/d 4 i 3 and roll [ /d/c/b/a ] { exch def } forall t C [ a F i -4 bitshift get exec a A /x r -8 bitshift /get A T
 i get A W { 1 32 bitshift 1 sub /and } if /dup r 31 and /bitshift /exch r 31 and 32 sub /bitshift /or b A /def ] dup length C add /C exch def p
utinterval } for 1 1 C 1 sub { dup 1 sub t exch get /def cvx eq {pop} {t exch 2 copy get cvx put} ifelse } for W /mt t end cvx bind def not { /m
a { 2 copy xor 0 lt { add } { 16#80000000 xor add 16#80000000 xor } ifelse } bind def } { /ma { add 16#0FFFFFFF and } bind def } ifelse /calc {
 20 dict begin /a 8#14721221401 def /b 8#35763325611 def /c 8#23056556376 def /d 8#2014452166 def /x 16 array def /origs exch def /oslen origs l
ength def /s oslen 72 add 64 idiv 64 mul dup /slen exch def string def s 0 origs putinterval s oslen 16#80 put s slen 8 sub oslen 31 and 3 bitsh
ift put s slen 7 sub oslen -5 bitshift 255 and put s slen 6 sub oslen -13 bitshift 255 and put 0 64 slen 64 sub { dup 1 exch 63 add { s exch get
 } for 15 -1 0 { x exch 6 2 roll 3 { 8 bitshift or } repeat put } for a b c d mt d ma /d exch def c ma /c exch def b ma /b exch def a ma /a exch
 def } for 16 string [ [ a b c d ] { 3 { dup -8 bitshift } repeat } forall ] 0 1 15 { 3 copy dup 3 1 roll get 255 and put pop } for pop end } bi
nd def
 %
```

Figure 24

Le bloc I4 déchiffré est listé dans la Figure 25. C'est la partie du script qui va déchiffrer le Bloc I1.

```
 0 0 0 0 2 2 16 4 sub { 6 index exch 4 getinterval 10240 { 0 0 1 3 { 3 -1 roll dup 4 1 roll exch get exch 8 bitshift add } for exch pop dup -2 bi
tshift exch dup -3 bitshift 1 index -7 bitshift xor exch dup 4 1 roll xor xor 1 and 31 bitshift exch -1 bitshift or 4 string exch 3 -1 0 {3 copy
 exch 255 and put pop -8 bitshift} for pop dup <55555555> le {1} { dup <aaaaaaaa> le {-1} {0} ifelse } ifelse 4 -1 roll add 5 index length add 5
 index length mod 3 1 roll 0 0 1 3 { 3 -1 roll dup 4 1 roll exch get exch 8 bitshift add } for exch pop dup -2 bitshift exch dup -3 bitshift 1 i
ndex -7 bitshift xor exch dup 4 1 roll xor xor 1 and 31 bitshift exch -1 bitshift or 4 string exch 3 -1 0 {3 copy exch 255 and put pop -8 bitshi
ft} for pop dup <55555555> le {1} { dup <aaaaaaaa> le {-1} {0} ifelse } ifelse 3 -1 roll add 5 index 0 get length 4 idiv add 5 index 0 get lengt
h 4 idiv mod exch 0 0 1 3 { 3 -1 roll dup 4 1 roll exch get exch 8 bitshift add } for exch pop dup -2 bitshift exch dup -3 bitshift 1 index -7 b
itshift xor exch dup 4 1 roll xor xor 1 and 31 bitshift exch -1 bitshift or 4 string exch 3 -1 0 {3 copy exch 255 and put pop -8 bitshift} for p
op 6 -2 roll 2 copy 8 2 roll get 4 index 4 mul 7 index 5 index get 4 index 4 mul 4 5 copy dup 4 1 roll getinterval 4 1 roll getinterval exch dup
 length string 0 3 -1 roll { 3 copy put pop 1 add } forall pop exch 3 -1 roll pop 4 -2 roll 3 -1 roll putinterval putinterval 5 index 5 index ge
t 4 index 4 mul 4 getinterval 1 index 0 0 1 1 {pop 2 index length} for exch 1 sub { 3 copy exch length getinterval 2 index mark 3 1 roll 0 1 3 -
1 roll dup length 1 sub exch 4 1 roll { dup 3 2 roll dup 5 1 roll exch get 3 1 roll exch dup 5 1 roll exch get xor 3 1 roll } for pop pop ] dup 2
 length string 0 3 -1 roll { 3 -1 roll dup 4 1 roll exch 2 index exch put 1 add } forall pop 4 -1 roll dup 5 1 roll 3 1 roll dup 4 1 roll putinte
rval exch pop } for pop pop 5 1 roll 2 copy 7 -3 roll pop pop } repeat pop 4 index 0 1 index { length add } forall string 0 3 2 roll { 3 copy pu
tinterval length add } forall pop calc I3 16 string readstring pop ne {0 1 1073741823 {pop} for (Key is invalid. Exiting ...\n) error flush quit
 } if } for pop pop pop pop (output.bin) (w) file exch 1 index resetfile {1 index exch writestring} forall closefile
 %%%
```

Figure 25

Ce script applique le même algorithme de déchiffrement 6 fois pour déchiffrer I1 en utilisant une clé à 4 octets. Il commence par utiliser les octets k[2]k[3]k[4]k[5], ensuite il se décale de 2 octets, pour que dans l'itération finale ce sont les octets k[12]k[13]k[14]k[15] qui sont utilisés.

La section suivante décrit l'algorithme lui-même.

4) L'analyse de l'algorithme de déchiffrement du bloc I1

L'algorithme consiste en 6 itérations avec à chaque fois une nouvelle clé mais l'état est conservé d'une itération à une autre.

Chaque itération consiste en 10240 étapes, une clé est dérivée pour chaque étape à partir de la dernière clé utilisée.

Chaque étape consiste en un swap de position entre deux blocs de données de 4octets, ainsi qu'un XOR de la clé de l'étape avec un des blocs swapé. Un pointeur sur l'autre bloc swapé est maintenu.

Les données à déchiffrer (I1) sont organisés en 77 bloc de 32 bloc de 4 octets.

L'état d'une étape est dérivé de l'état de l'étape précédente. Un état consiste en une clé de 4 octets k, et le pointeur vers le dernier bloc de 4 octets swapé : pl et pla (Figure 26)

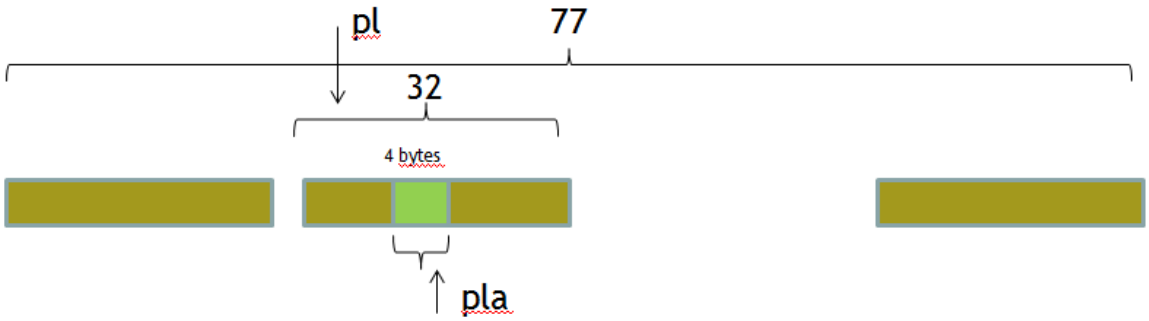


Figure 26

A chaque étape, la clé subit 3 décalages. Un décalage est illustré dans la Figure 27

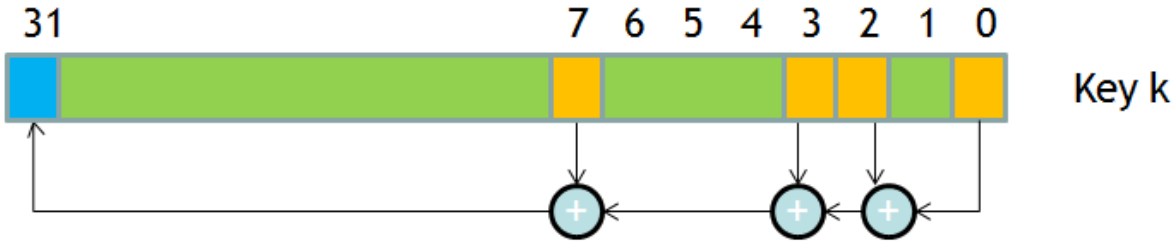


Figure 27

Le premier décalage de la clé sert à déterminer la prochaine valeur de pl (+1, -1 ou la même) en comparant la clé avec 0x55555555 et 0xaaaaaaaa.

Le deuxième décalage de la clé sert à déterminer la prochaine valeur de pla (+1, -1 ou la même) en comparant la clé avec 0x55555555 et 0xaaaaaaaa.

Et enfin le troisième décalage de la clé produit la clé utilisée pour le Xor.

L'état d'une nouvelle itération est l'état de la dernière étape de l'itération précédente (mais une nouvelle clé).

A la fin de chaque itération une vérification du checksum des données intermédiaires est effectuée. Le bloc I3 contient les 6 checksums utilisés.

5) Détermination de la clé

On va cette fois écrire notre programme en langage C pour des meilleures performances. On va l'exécuter 6 fois en changeant les états initiaux de chaque exécution qui sont dérivés de l'exécution qui précède. Chaque exécution correspond à une itération de l'algorithme.

Chaque exécution nous permettra de déterminer 2 octets de la clé. La Figure 28 montre l'implémentation d'une étape.

```c
#include <stdio.h>
#include <openssl/md5.h>
#include <unistd.h>

unsigned int I1[77][32];
unsigned int Init_I1[77][32];

unsigned int k, pl, pla;

unsigned char out[MD5_DIGEST_LENGTH];
//unsigned char target[]= {0x89 ,0x90, 0xc9, 0xa0, 0xb1, 0x3f, 0xe7, 0x3e, 0x35, 0xa6, 0xc5, 0x42, 0x23,
0x24, 0xdc, 0x6c};
//unsigned char target[]={0x33, 0x8f, 0x25 ,0x66 ,0x7e ,0xb4 ,0xec, 0x47 , 0x76, 0x3d, 0xab, 0x51, 0xc3,
0xfa, 0x41, 0xcb};
//unsigned char target[]={0xa3, 0x29 ,0xe1, 0x85, 0x36, 0Xb8, 0x31, 0x59,  0xb3, 0xa6, 0x90, 0xa0, 0x26,
0x5e, 0xc5, 0x19};
//unsigned char target[]={0xaa, 0xe9, 0x4f ,0x0e, 0x71 ,0x53, 0x76, 0xc4, 0xf0, 0x87, 0xbc, 0xcc, 0xdd,
0x0b, 0xe3 ,0xb4};
//unsigned char target[]={0xa1, 0x14, 0xf8, 0xbe, 0x74, 0x61, 0x42, 0xc4, 0x49,0x78, 0xfa, 0xa7, 0x6d,
0xae, 0x62, 0xcf};
//unsigned char target[]={ 0x19, 0x7d, 0x7b, 0xce, 0x4e, 0xb3, 0x8d, 0xd6,  0x8c, 0x8c, 0xe5, 0xf6, 0x9f,
0x32, 0x6e, 0x1e};
unsigned char target[]={ 0xff, 0xce ,0xae ,0x3f, 0x72, 0xf8, 0xea, 0xa3, 0x8e, 0x01, 0x9a, 0x59, 0xb1,
0xdc, 0x09, 0x97 };

void round_enc()
{
  unsigned char tm3,tm2,tm1,tm0;
  unsigned int  ka,kb,kc;
  unsigned char Kc[4];
  unsigned int l,la;
  char t,ta;
  unsigned int temp;

  ka = (( ( (k>>2)^(k>>3)^(k>>7)^k ) & 0x1) <<31 ) | (k>>1);
  if (ka <= 0x55555555) t=1;
  else  if (ka <= 0xaaaaaaaa) t=-1; else t=0;

  l= ( t+pl + 77) % 77;

  kb = (( ( (ka>>2)^(ka>>3)^(ka>>7)^ka ) & 0x1) <<31 ) | (ka>>1);
  if (kb <= 0x55555555) ta=1;
  else if (kb <= 0xaaaaaaaa) ta=-1; else ta=0;

  la=(ta+pla+32) % 32;

  kc = (( ( (kb>>2)^(kb>>3)^(kb>>7)^kb ) & 0x1) <<31 ) | (kb>>1);
  Kc[0]= (kc>>24)&0xFF ; Kc[1]= (kc>>16)&0xFF ; Kc[2]= (kc>>8)&0xFF; Kc[3]= kc & 0xFF;

  temp=I1[pl][pla];
  I1[pl][pla]=I1[l][la];
  I1[l][la]=temp;

  tm0= (I1[pl][pla] >>24)&0xFF ; tm1= (I1[pl][pla]>>16)&0xFF ;
  tm2= (I1[pl][pla]>>8)&0xFF;    tm3= I1[pl][pla] & 0xFF;

  I1[pl][pla]=((tm0 ^ Kc[0]) <<24) | ( (tm1 ^ Kc[1]) <<16) | ( (tm2 ^ Kc[2])<<8 ) | ( tm3 ^ Kc[3] );

  k=kc;    pl=l;   pla=la;
  return;
}
```

Figure 28

Figure 29 est la fonction qui charge les donnés intermédiaire de l'itération.

```
void read_I1() {
  unsigned int m,n;
  unsigned char tm3,tm2,tm1,tm0;
  //FILE * f=fopen("I1","rb");
  //FILE * f=fopen("I1_k4_k5","rb");
  //FILE * f=fopen("I1_k6_k7","rb");
  //FILE * f=fopen("I1_k8_k9","rb");
  //FILE * f=fopen("I1_k10_k11","rb");
  FILE * f=fopen("I1_k12_k13","rb");

  for (m=0;m<77;m++)
    for(n=0;n<32;n++){
        fread(&tm0,1,1,f);fread(&tm1,1,1,f);fread(&tm2,1,1,f);fread(&tm3,1,1,f);
        I1[m][n]=(tm0<<24)|(tm1<<16)|(tm2<<8)|tm3;
      }
  fclose(f);
  return;
}
```

Figure 29

La Figure 30 est l'implémentation d'une itération.

On obtient successivement :

k[4]=0x72 ; k[5]=0x1f ;

k[6]=0xad ; k[7]=0x3c ;

k[8]=0x9f ; k[9]=0xcf ;

k[10]=0x27 ; k[11]=0x1e ;

k[12]=0xed ; k[13]=0x9a ;

k[14]=0xbb ; k[15]=0xc8 ;

```
int main() {
  int m,n,j;
  FILE * f;
  unsigned char tm3,tm2,tm1,tm0;
  unsigned int  k3,k4;
  unsigned char clear[9856];
  unsigned char match;

  read_I1();
  for (m=0;m<77;m++)
    for(n=0;n<32;n++)
      Init_I1[m][n]=I1[m][n];
  //pla=0;pl=0;
  //pl=0x4b;pla=0xa;
  //pl=0x46;pla=0x1a;
  //pl=0xd ;pla=0x8;
  //pl=0x1a ;pla=0x1f ;
  pl=0xf ,pla=0x17;

  for (k3=0;k3<256;k3++){
      printf("trying k3=0x%x\n",k3);
      for (k4=0;k4<256;k4++){
          //pla=0;pl=0;
          //pl=0x4b;pla=0xa;
          //pl=0x46;pla=0x1a;
          //pl=0xd ;pla=0x8;
          //pl=0x1a ;pla=0x1f ;
          pl=0xf ,pla=0x17;

          for (m=0;m<77;m++)
            for(n=0;n<32;n++)
              I1[m][n]=Init_I1[m][n];
          //k=0xf7a80000 | (k3<<8) | k4;
          //k=0x721f0000 | (k3<<8) | k4;
          //k=0xad3c0000 | (k3<<8) | k4;
          //k=0x9fcf0000 | (k3<<8) | k4;
          //k=0x271e0000 | (k3<<8) | k4;
          k=0xed9a0000 | (k3<<8) | k4;

          for(j=0;j<10240;j++)
            round_enc();

          for (m=0;m<77;m++)
            for(n=0;n<32;n++){
              clear[m*128+n*4]=(I1[m][n] >>24)&0xFF;
              clear[m*128+4*n+1]=(I1[m][n] >>16)&0xFF;
              clear[m*128+4*n+2]=(I1[m][n] >>8) &0xFF;
              clear[m*128+4*n+3]=(I1[m][n])     &0xFF;
              }
          MD5(clear, 9856, out);
          match=1;
          for(n=0; n<MD5_DIGEST_LENGTH; n++){
              if ( out[n] != target[n])
                match=0;
            }
          if(match == 1){
              printf("Found a Hit k3=0x%x k4=0x%x pl=0x%x pla=0x%x \n", k3,k4,pl,pla);
              //f=fopen("I1_k4_k5","wb");
              //f=fopen("I1_k6_k7","wb");
              //f=fopen("I1_k8_k9","wb");
              //f=fopen("I1_k10_k11","wb");
              //f=fopen("I1_k12_k13","wb");
              f=fopen("I1_k14_k15","wb");
              fwrite( &clear, 1, 9856, f);
              fclose(f);
              exit(0);
    }   }    } return 0; }
```

Figure 30

La bonne clé est donc :

bac9f7a8721fad3c9fcf271eed9abbc8

Et en exécutant:

$gs -- atad bac9f7a8721fad3c9fcf271eed9abbc8

On obtient un fichier “output.bin”

IV) Analyse du fichier output.bin

C'est un carnet d'adresse en format vCard, sans tarder on l'ouvre avec Thunderbird (Figure 31)

L'adresse émail recherchée est codée avec une suite de 60 noms de fonction. On reconnait des appels système Linux.

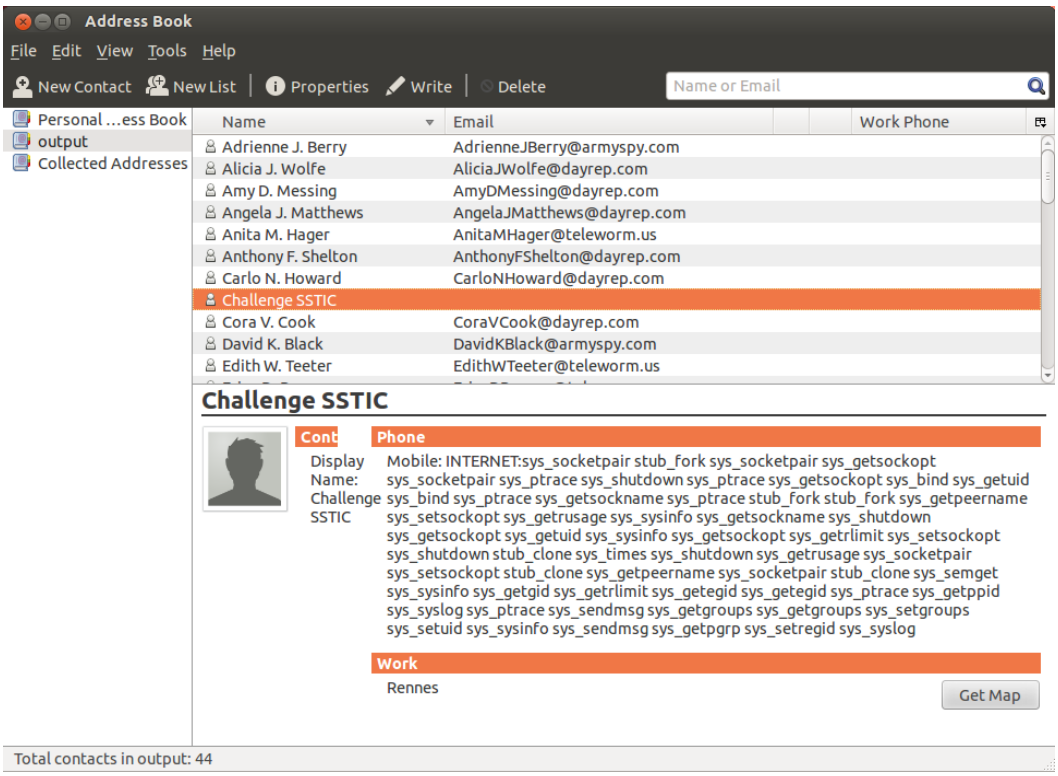


Figure 31

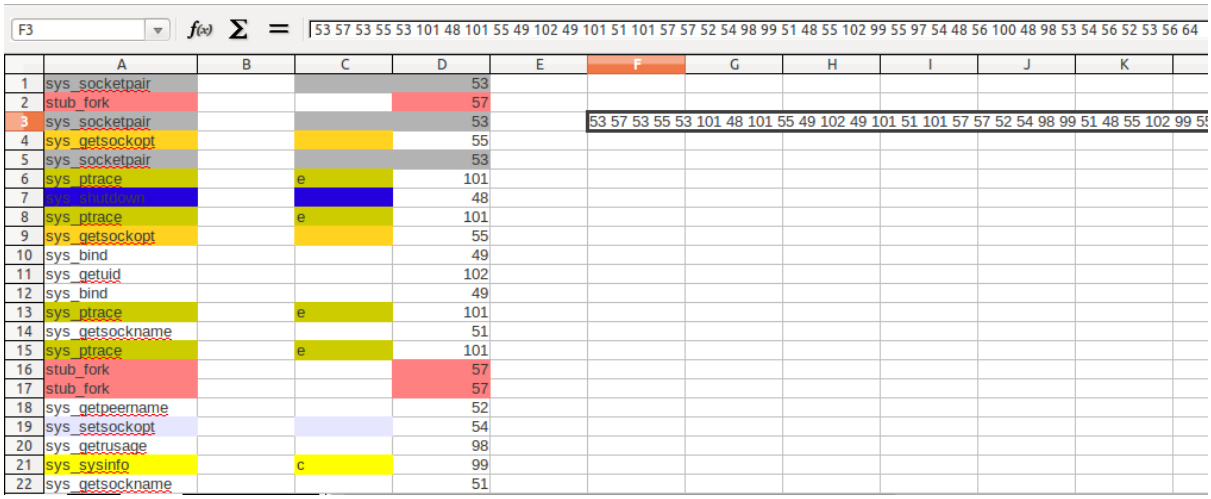
On sait déjà que l'adresse email est de la forme …@challenge.sstic.org, la Figure 32 suggère que chaque appel système est une substitution de caractère ascii.

41	sys_semget		@
42	sys_sysinfo		c
43	sys_getgid		h
44	sys_getrlimit		a
45	sys_getegid		l
46	sys_getegid		l
47	sys_ptrace		e
48	sys_getppid		n
49	sys_syslog		g
50	sys_ptrace		e
51	sys_sendmsg		.
52	sys_getgroups		s
53	sys_getgroups		s
54	sys_setgroups		t
55	sys_setuid		l
56	sys_sysinfo		c
57	sys_sendmsg		.
58	sys_getpgrp		o
59	sys_setregid		r
60	sys_syslog		g

Figure 32

Les appels systèmes linux sont organisés dans le noyau dans une table, on remplacera donc les appels systèmes par leur numéro d'index dans la table. A la toute première étape du challenge on avait une indication sur la version du système linux utilisé ce qui nous permettra d'utiliser une bonne version du fichier *Linux/arch/x86/include/asm/unistd_64.h* (http://lxr.free-electrons.com/source/arch/x86/include/asm/unistd_64.h?v=2.6.32#L127)

La Figure 33 capture une partie du décodage, on remarque que le numéro d'index des appels systèmes correspond au code ascii du caractère qu'il remplace.



F3 | 53 57 53 55 53 101 48 101 55 49 102 49 101 51 101 57 57 52 54 98 99 51 48 55 102 99 55 97 54 48 56 100 48 98 53 54 56 52 53 56 64

	A	B	C	D	E	F	G	H	I	J	K
1	sys_socketpair			53							
2	stub_fork			57							
3	sys_socketpair			53		53 57 53 55 53 101 48 101 55 49 102 49 101 51 101 57 57 52 54 98 99 51 48 55 102 99 5					
4	sys_getsockopt			55							
5	sys_socketpair			53							
6	sys_ptrace		e	101							
7	sys_shutdown			48							
8	sys_ptrace		e	101							
9	sys_getsockopt			55							
10	sys_bind			49							
11	sys_getuid			102							
12	sys_bind			49							
13	sys_ptrace		e	101							
14	sys_getsockname			51							
15	sys_ptrace		e	101							
16	stub_fork			57							
17	stub_fork			57							
18	sys_getpeername			52							
19	sys_setsockopt			54							
20	sys_getrusage			98							
21	sys_sysinfo		c	99							
22	sys_getsockname			51							

Figure 33

Au final on obtient l'adresse émail recherchée :

```
>>> em=[53, 57, 53, 55, 53, 101, 48, 101, 55, 49, 102, 49, 101, 51, 101 ,57 ,57 ,52, 54 ,98 ,99, 51, 48, 55 ,102 ,99, 55, 97, 54 ,48 ,56, 100 ,48 ,98 ,53 ,54 ,56, 52 ,53, 56, 64]

>>> "".join([chr(i) for i in em])

'59575e0e71f1e3e9946bc307fc7a608d0b568458@'
```

L'adresse complète est donc :

```
59575e0e71f1e3e9946bc307fc7a608d0b568458@challenge.sstic.org
```

V) Conclusion

Ce fut un exercice plaisant avec une variété de thèmes. Merci aux organisateurs de ce challenge sstic 2013.

Annexe 1

Programme smp Désassemblé

```
0x00:   0x00:  mov 0x00 accu
0x01:   0xb0:  mov accu r[0]
0x02:   0x10:  mov 0x10 accu
0x03:   0xd0:  mov smd[accu] accu
0x04:   0xb7:  mov accu r[7]
0x05:   0xa8:  mov r[0] accu
0x06:   0xa0:  not ea
0x07:   0xb6:  mov accu r[6]
0x08:   0x0e:  mov 0x0e accu
0x09:   0xb5:  mov accu r[5]
0x0a:   0x71:  mov 0x71 accu
0x0b:   0xb4:  mov accu r[4]
0x0c:   0x00:  mov 0x00 accu
0x0d:   0xbc:  jmp r[4]
0x0e:   0x01:  mov 0x01 accu
0x0f:   0xb6:  mov accu r[6]
0x10:   0x16:  mov 0x16 accu
0x11:   0xb5:  mov accu r[5]
0x12:   0x71:  mov 0x71 accu
0x13:   0xb4:  mov accu r[4]
0x14:   0x00:  mov 0x00 accu
0x15:   0xbc:  jmp r[4]
0x16:   0x52:  mov 0x52 accu
0x17:   0xb6:  mov accu r[6]
0x18:   0xaf:  mov r[7] accu
0x19:   0xbe:  jmp r[6]
0x1a:   0x11:  mov 0x11 accu
0x1b:   0xb7:  mov accu r[7]
0x1c:   0xa8:  mov r[0] accu
0x1d:   0xb6:  mov accu r[6]
0x1e:   0x24:  mov 0x24 accu
0x1f:   0xb5:  mov accu r[5]
0x20:   0x71:  mov 0x71 accu
0x21:   0xb4:  mov accu r[4]
0x22:   0x00:  mov 0x00 accu
0x23:   0xbc:  jmp r[4]
0x24:   0xaf:  mov r[7] accu
0x25:   0xb1:  mov accu r[1]
0x26:   0xa9:  mov r[1] accu
0x27:   0xd0:  mov smd[accu] accu
0x28:   0xb6:  mov accu r[6]
0x29:   0x0f:  mov 0x0f accu
0x2a:   0x88:  and r[0] accu
0x2b:   0xd0:  mov smd[accu] accu
0x2c:   0x96:  or r[6] accu
0x2d:   0xb6:  mov accu r[6]
0x2e:   0xa9:  mov r[1] accu
0x2f:   0xd0:  mov smd[accu] accu
0x30:   0xb7:  mov accu r[7]
0x31:   0x0f:  mov 0x0f accu
0x32:   0x88:  and r[0] accu
0x33:   0xd0:  mov smd[accu] accu
0x34:   0x8f:  and r[7] accu
0x35:   0xb7:  mov accu r[7]
0x36:   0xaf:  mov r[7] accu
0x37:   0xa0:  not ea
0x38:   0x8e:  and r[6] accu
0x39:   0xb7:  mov accu r[7]
0x3a:   0x40:  mov 0x40 accu
0x3b:   0xb6:  mov accu r[6]
0x3c:   0x53:  mov 0x53 accu
0x3d:   0xb5:  mov accu r[5]
0x3e:   0x00:  mov 0x00 accu
0x3f:   0xbd:  jmp r[5]
0x40:   0xaf:  mov r[7] accu
0x41:   0xc1:  mov accu smd[r[1]]
0x42:   0x01:  mov 0x01 accu
0x43:   0xb6:  mov accu r[6]
0x44:   0xa8:  mov r[0] accu
0x45:   0xb7:  mov accu r[7]
0x46:   0x4c:  mov 0x4c accu
0x47:   0xb5:  mov accu r[5]
0x48:   0x71:  mov 0x71 accu
0x49:   0xb4:  mov accu r[4]
0x4a:   0x00:  mov 0x00 accu
0x4b:   0xbc:  jmp r[4]
0x4c:   0xaf:  mov r[7] accu
0x4d:   0xb0:  mov accu r[0]
0x4e:   0x02:  mov 0x02 accu
0x4f:   0xb6:  mov accu r[6]
0x50:   0x00:  mov 0x00 accu
0x51:   0xbe:  jmp r[6]
0x52:   0xc8:  end
0x53:   0x01:  mov 0x01 accu
0x54:   0xb5:  mov accu r[5]
0x55:   0x00:  mov 0x00 accu
0x56:   0xb4:  mov accu r[4]
0x57:   0x6d:  mov 0x6d accu
0x58:   0xb3:  mov accu r[3]
0x59:   0xad:  mov r[5] accu
0x5a:   0xbb:  jmp r[3]
0x5b:   0x66:  mov 0x66 accu
0x5c:   0xb3:  mov accu r[3]
0x5d:   0xac:  mov r[4] accu
0x5e:   0xd8:  shl accu
0x5f:   0xb4:  mov accu r[4]
0x60:   0xad:  mov r[5] accu
0x61:   0x8f:  and r[7] accu
0x62:   0xbb:  jmp r[3]
0x63:   0x01:  mov 0x01 accu
0x64:   0x94:  or r[4] accu
0x65:   0xb4:  mov accu r[4]
0x66:   0xad:  mov r[5] accu
0x67:   0xd8:  shl accu
0x68:   0xb5:  mov accu r[5]
0x69:   0x57:  mov 0x57 accu
0x6a:   0xb3:  mov accu r[3]
0x6b:   0x00:  mov 0x00 accu
0x6c:   0xbb:  jmp r[3]
0x6d:   0xac:  mov r[4] accu
0x6e:   0xb7:  mov accu r[7]
0x6f:   0x00:  mov 0x00 accu
0x70:   0xbe:  jmp r[6]
0x71:   0x00:  mov 0x00 accu
0x72:   0xb1:  mov accu r[1]
0x73:   0xb3:  mov accu r[3]
0x74:   0x01:  mov 0x01 accu
0x75:   0xb2:  mov accu r[2]
0x76:   0x63:  mov 0x63 accu
0x77:   0xe0:  mask accu
0x78:   0xb4:  mov accu r[4]
0x79:   0xaa:  mov r[2] accu
0x7a:   0xbc:  jmp r[4]
0x7b:   0x2c:  mov 0x2c accu
0x7c:   0xe0:  mask accu
0x7d:   0xb4:  mov accu r[4]
0x7e:   0xaf:  mov r[7] accu
0x7f:   0x8a:  and r[2] accu
0x80:   0xbc:  jmp r[4]
0x81:   0x16:  mov 0x16 accu
0x82:   0xe0:  mask accu
0x83:   0xb4:  mov accu r[4]
0x84:   0xae:  mov r[6] accu
0x85:   0x8a:  and r[2] accu
0x86:   0xbc:  jmp r[4]
0x87:   0x0f:  mov 0x0f accu
0x88:   0xe0:  mask accu
0x89:   0xb4:  mov accu r[4]
```

```
0x8a:   0xa9:  mov r[1] accu
0x8b:   0xbc:  jmp r[4]
0x8c:   0xab:  mov r[3] accu
0x8d:   0x92:  or r[2] accu
0x8e:   0xb3:  mov accu r[3]
0x8f:   0xaa:  mov r[2] accu
0x90:   0xb1:  mov accu r[1]
0x91:   0x59:  mov 0x59 accu
0x92:   0xe0:  mask accu
0x93:   0xb4:  mov accu r[4]
0x94:   0x00:  mov 0x00 accu
0x95:   0xbc:  jmp r[4]
0x96:   0x22:  mov 0x22 accu
0x97:   0xe0:  mask accu
0x98:   0xb4:  mov accu r[4]
0x99:   0xa9:  mov r[1] accu
0x9a:   0xbc:  jmp r[4]
0x9b:   0xaa:  mov r[2] accu
0x9c:   0xb1:  mov accu r[1]
0x9d:   0x59:  mov 0x59 accu
0x9e:   0xe0:  mask accu
0x9f:   0xb4:  mov accu r[4]
0xa0:   0x00:  mov 0x00 accu
0xa1:   0xbc:  jmp r[4]
0xa2:   0xab:  mov r[3] accu
0xa3:   0x92:  or r[2] accu
0xa4:   0xb3:  mov accu r[3]
0xa5:   0x00:  mov 0x00 accu
0xa6:   0xb1:  mov accu r[1]
0xa7:   0x59:  mov 0x59 accu
0xa8:   0xe0:  mask accu
0xa9:   0xb4:  mov accu r[4]
0xaa:   0x00:  mov 0x00 accu
0xab:   0xbc:  jmp r[4]
0xac:   0x48:  mov 0x48 accu
0xad:   0xe0:  mask accu
0xae:   0xb4:  mov accu r[4]
0xaf:   0xae:  mov r[6] accu
0xb0:   0x8a:  and r[2] accu
0xb1:   0xbc:  jmp r[4]
0xb2:   0x3e:  mov 0x3e accu
0xb3:   0xe0:  mask accu
0xb4:   0xb4:  mov accu r[4]
0xb5:   0xa9:  mov r[1] accu
0xb6:   0xbc:  jmp r[4]
0xb7:   0xaa:  mov r[2] accu
0xb8:   0xb1:  mov accu r[1]
0xb9:   0x59:  mov 0x59 accu
0xba:   0xe0:  mask accu
0xbb:   0xb4:  mov accu r[4]
0xbc:   0x00:  mov 0x00 accu
```

```
0xbd:   0xbc:  jmp r[4]
0xbe:   0xaa:  mov r[2] accu
0xbf:   0x93:  or r[3] accu
0xc0:   0xb3:  mov accu r[3]
0xc1:   0x00:  mov 0x00 accu
0xc2:   0xb1:  mov accu r[1]
0xc3:   0x59:  mov 0x59 accu
0xc4:   0xe0:  mask accu
0xc5:   0xb4:  mov accu r[4]
0xc6:   0x00:  mov 0x00 accu
0xc7:   0xbc:  jmp r[4]
0xc8:   0x57:  mov 0x57 accu
0xc9:   0xe0:  mask accu
0xca:   0xb4:  mov accu r[4]
0xcb:   0xa9:  mov r[1] accu
0xcc:   0xbc:  jmp r[4]
0xcd:   0xaa:  mov r[2] accu
0xce:   0x93:  or r[3] accu
0xcf:   0xb3:  mov accu r[3]
0xd0:   0x00:  mov 0x00 accu
0xd1:   0xb1:  mov accu r[1]
0xd2:   0x59:  mov 0x59 accu
0xd3:   0xe0:  mask accu
0xd4:   0xb4:  mov accu r[4]
0xd5:   0x00:  mov 0x00 accu
0xd6:   0xbc:  jmp r[4]
0xd7:   0x00:  mov 0x00 accu
0xd8:   0xb1:  mov accu r[1]
0xd9:   0xaa:  mov r[2] accu
0xda:   0xd8:  shl accu
0xdb:   0xb2:  mov accu r[2]
0xdc:   0xa9:  mov r[1] accu
0xdd:   0xd8:  shl accu
0xde:   0xb1:  mov accu r[1]
0xdf:   0x76:  mov 0x76 accu
0xe0:   0xb4:  mov accu r[4]
0xe1:   0x00:  mov 0x00 accu
0xe2:   0xbc:  jmp r[4]
0xe3:   0xab:  mov r[3] accu
0xe4:   0xb7:  mov accu r[7]
0xe5:   0x00:  mov 0x00 accu
0xe6:   0xbd:  jmp r[5]
```