Challenge SSTIC 2013
Approche Thales SecureLab

Jean-Yves Burlett <jean-yves.burlett@thalesgroup.com>

28 mars 2013

1 Introduction

Comme chaque année, l'objectif du challenge consiste à retrouver et contacter une adresse email du domaine challenge.sstic.org dans un fichier fourni. L'extraction de cette adresse email s'est déroulée en 4 étapes.

2 Extraction d'une clef AES transmise par canal caché

Le challenge SSTIC 2013 commence par la récupération du fichier du challenge dump.bin (http ://static.sstic.org/challenge2013/dump.bin)

L'utilisation de la commande file donne très rapidement un aperçu du contenu de ce fichier. C'est un capture réseau qui sera par la suite ouverte dans wireshark.

```
1 $ file dump.bin
dump.bin: tcpdump capture file (little-endian) - version 2.4 (Ethernet,
     capture length 65535)
```

2.1 Analyse rapide du fichier PCAP

Le fichier dump.bin est donc ouvert dans wireshark. Cette capture présente plusieurs types de paquets dont des paquets icmp et des paquets ftp particulièrement remarquables. Mais la première chose à considérer est l'indice qui nous est offert par les concepteurs du challenge dans les premiers paquets de la capture.

```
Bonjour,
J'ai egare la cle pour dechiffrer mon carnet d'adresses.
Tu pourrais m'aider a la retrouver ? J'ai besoin de recuperer une adresse email
a l'interieur.
Pour t'aider, je t'envoie :
- une archive chiffree en AES par FTP
- la cle AES par canaux caches
voici l'iv utilise pour AES : 76C128D46A6C4B15B43016904BE176AC
voici le checksum de l'archive pour verifier le dechiffrement :
61c9392f617290642f9a12499de6b688
merci

PS :
Indication pour les canaux caches : 1 bit de canal cache temporel
concatene a 3 bits de canal cache non temporel.
```

Figure 1 – Indices en début de capture réseau

Au vu de ces indices, il est possible d'imaginer ce qui nous attend pour la première étape de ce challenge. Il faut extraire le fichier chiffré de la capture et trouver une clef transmise par canal caché pour déchiffrer ce fichier. Nous savons que ce fichier est une archive tar.gz grâce au nom qui est présent dans les paquets ftp.

```
USER sstic
PASS sstic
PWD
FEAT
HELP SITE
CLNT NcFTP 3.2.2 linux-x86-glibc2.9
TYPE I
REST 1
REST 0
SIZE sstic.tar.gz-chiffre
MDTM sstic.tar.gz-chiffre
PASV
STOR sstic.tar.gz-chiffre
SITE UTIME sstic.tar.gz-chiffre 20130318113720 20130318113533
    20130318113533 UTC
MDTM 20130318113533 sstic.tar.gz-chiffre
QUIT
```

Il faut donc dans un premier temps récupérer le fichier qui a été transmis par ftp :

```
$ tcpflow -r dump.bin -C "tcp port 60733" > sstic.tar.gz-chiffre.b64
```

Le fichier est formaté en base64 :

```
$ head sstic.tar.gz-chiffre.b64
```


```
lscdhg9i6334AVND30wuytHRs54ffHW3++P2RyisiF5TpF6TirU431NKjChOr0Ar
wtKUwx7JKRZ0wpRzrRbjUNlzpI5WRHEB8l8gJxu6noVAYbXCPIBCtJPlcvULcZLC
aF0glAf3dEpUxsTQ3+URUy4SPj7mXL2iRTPwyh7CTldU4fdL3h2Ydy/8M+lSnOd8
xgL4fapcLvyYk7k2yFN/3S+xbgSbvkhWPTtdhmdkAZ/5nKdJZdIG0aXEotnPOrIY
M/gmOOzb0eFED+jaA6cJ4xQGDDSYVFYWjvCAiNRygr8R6laRM3CmFeXCVrEgqu5m
9YdG8VyHPgNp2a5sZedp3qbcUocSBieULIxn0kvUMxnha/rxwT8O6MWbB8pnQ9Fp
ioNO8RcE5f+zUcmjhEZ2U68b507GGk+UZAjpPJCrmrOe4Iz1HBDJgd1Cu2mPQS9T
qbeJ/T6TaxuHQTHoylLclCBEnU/jia+S0590PCA437v5q5DV9LlT34Ifn0/w8uOh
q1aL/JC+tlw5moel2LtYk7jFivkjPNaVtGgdtRG74PQeHNXmp4eXgC0SoTcbPcpY
```

```
11 +4XXkDHYP5EiTJt6cuhQ/0C0pijzX0TZye/Rsau+oLVz2eljcc0H5v+FO0nJ6B/M
```


Il est donc possible d'obtenir le fichier binaire chiffré avec la commande suivante :

```
1 $ base64 -d sstic.tar.gz-chiffre.b64 > sstic.tar.gz-chiffre
```

Maintenant que le fichier chiffré est en notre possession nous allons pouvoir nous concentrer sur la recherche de la clef.

2.2 Analyse des paquets ICMP pour trouver la clef

La première chose qui attire l'attention dans la capture wireshark est la présence de 65 paquets icmp.

6	1.030155	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0xf132, seq=1/256, ttl=30
7	3.042155	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x0233, seq=1/256, ttl=30
8	5.055383	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x1333, seq=1/256, ttl=40
9	7.068428	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x2433, seq=1/256, ttl=30
10	9.082048	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x3533, seq=1/256, ttl=20
11	11.095455	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x4633, seq=1/256, ttl=10
12	12.108802	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x5533, seq=1/256, ttl=30
13	14.122642	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x6633, seq=1/256, ttl=30
14	15.136050	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x7533, seq=1/256, ttl=10
15	16.149389	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x8433, seq=1/256, ttl=20
16	18.162944	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x9533, seq=1/256, ttl=20
17	19.176277	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0xa433, seq=1/256, ttl=40
18	21.189804	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0xb533, seq=1/256, ttl=10
19	23.203362	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0xc633, seq=1/256, ttl=10
20	25.216709	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0xd733, seq=1/256, ttl=20
21	27.230168	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0xe833, seq=1/256, ttl=10
22	29.243681	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0xf933, seq=1/256, ttl=20
23	30.258967	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x0834, seq=1/256, ttl=10
24	31.275616	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x1734, seq=1/256, ttl=30
25	32.287502	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x2634, seq=1/256, ttl=20
26	34.303755	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x3734, seq=1/256, ttl=10
27	35.319402	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x4634, seq=1/256, ttl=10
28	37.336051	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x5734, seq=1/256, ttl=30
29	39.352397	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x6834, seq=1/256, ttl=30
30	41.370406	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x7934, seq=1/256, ttl=40
31	42.386615	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x8834, seq=1/256, ttl=20

FIGURE 2 – Paquets ICMP de la capture réseau

D'après l'indice on cherche une clef AES et on a récupère 4 bits en même temps. Il est donc aisément possible d'imaginer que nous allons récupérer 4 bits par paquet icmp. 4 * 64 = 256 bits soit une longueur de clef standard pour l'AES. Nous disposons donc d'un paquet en trop, ce qui peut laisser penser qu'au moins un bit de clef nécessitera un calcul entre deux paquets, ce qui est cohérent avec la mention d'1 bit de canal caché temporel dans les indices.

Un indice supplémentaire est la mention de l'IV (Initialisation Vector) de l'AES. Cela permet de restreindre les modes de fonctionnement de l'AES à tester. Notamment, les modes classiques CBC, OFB et CFB. CBC étant le mode le plus courant, il est très probable que ce soit celui utilisé.

En observant les paquets ICMP, nous pouvons identifier plusieurs sources d'entropie pour les bits de clef :

- intervalle de temps entre chaque paquet ICMP ;
- valeur du champ IP TTL ;

– valeur du champ IP Differentiated Services (utilisé pour signaler les niveaux de service (QoS) demandés à l'infrastructure).

2.2.1 Extraction des bits d'entropie

Ces valeurs brutes sont extraites à l'aide de tshark :

```
1 $ tshark -r dump.bin -T fields -e ip.ttl icmp > list-icmp-ttl
  $ tshark -r dump.bin -T fields -e frame.time_delta icmp > list-frame-
      delta
3 $ tshark -r dump.bin -T fields -e ip.dsfield icmp > list-ip-dsfield
```

Le champ TTL prend 4 valeurs différentes (2 bits d'entropie) ainsi qu'une valeur spécifique au 65e paquet. Le champ TTL n'étant pas dépendant du temps, il s'agit probablement d'un indice pour ignorer le dernier paquet.

Les intervalles de temps entre chaque paquet ICMP sont soit de 1, soit de 2 secondes. Cela permet de coder un bit de clef en utilisant les 65 paquets (64 différences entre 65 paquets).

Le champ DS prend lui aussi deux valeurs différentes plus une valeur spécifique pour le dernier paquet. Il permet donc, en éliminant le dernier paquet, de coder 1 bit de clef.

2.2.2 Génération de clef et test

Chacun des symboles extraits des paquets ICMP peut être interprété de plusieurs façons. De la même façon, chacun des 4 bits peut être permuté. Il va donc falloir tester un certain nombre de permutations pour la construction de la clef AES.

Il faut également construire un test d'arrêt pour isoler les bonnes clefs potentielles des clefs incorrectes. Le discriminant parfait est le condensat MD5 fourni dans les indices mais le tester nécessite de déchiffrer l'intégralité du message, de supprimer le padding éventuel et de calculer son MD5. C'est un test coûteux.

Dans un premier temps, nous allons partir de l'hypothèse que le nom du fichier n'est pas un piège et que celui-ci aura la structure d'un fichier GZip. Cela nous permet de filtrer les clefs qui permettent de décoder l'entête GZip standard. Si l'entête GZip (0x1f8b) est décodée, alors seulement nous tenteront le déchiffrement et la validation du MD5.

Le code du script de test de clef est fourni en annexe 7.1, page 16. La clef est trouvée très rapidement avec le mode CBC d'AES-256.

La combinaison de clef gagnante est la concaténation de blocs de 4 bits constitués comme suit :

– le délai entre deux paquets ICMP -1 seconde
– les 2 bits de champ TTL permutés $(0, 1, 2, 3) => (1, 3, 2, 0)$
– le bit obtenu de `((dsfield >> 1)-1) xor 1`

La clef de l'archive est :

```
1 dd 8c f2 d5 2e 69 aa fb 73 4e 3a cd 0e 4a 69 e8
  3e d9 3b c4 87 0e cd 0d 5b 6f aa d8 6a 63 ae 94
```

Le fichier extrait est une archive tar.gz contenant les données de l'étape suivante.

3 Rétro-ingéniérie et émulation d'un processeur FPGA

3.1 Découverte des fichiers de l'étape 2

L'étape 2 commence à partir de l'archive récupérée en déchiffrant le fichier de l'étape 1 :

```
$ tar xzvf sstic.tar.gz
archive/
archive/smp.py
archive/data
archive/s.ngr
archive/decrypt.py
```

Il est intéressant d'analyser chaque fichier. Le fichier data est un fichier binaire ne comportant pas de chaine de caractère marquante, il est chiffré. Le fichier smp.py est une liste python de 231 octets. Il sera utilisé dans la suite de l'étape 2. Le fichier decrypt.py est un programme python destiné à déchiffrer le fichier data à l'aide d'une clef.

```
#!/usr/bin/python

import sys
import base64
import md5

import dev
import smp

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

key = int(sys.argv[1], 16)
key = [(key >> (i * 8)) & 0xff for i in range(16)]

result = []
d = open("data", "rb").read()
dev.init("sp.ngr")
for i in range(0, len(d), 224):
  smd = d[i : (i + 224)]
  smd = (key, len(smd), smd)
  dev.send_smd(smd)
  dev.send_smp(smp.smp)
  dev.start()
  dev.wait_finished()
  result = result + dev.get_data()

print "".join(result)
result_md5 = md5.new()
result_md5.update("".join(result))
result_md5 = result_md5.digest()
result_md5 = [ord(x) for x in result_md5]
target_md5 = "6c0708b3cf6e32cbae4236bdea062979"
target_md5 = [int(target_md5[x:(x + 2)], 16) for x in range(0, len(
    target_md5), 2)]
print(["%02x" % x for x in target_md5])
```

```
print(["%02x" % x for x in result_md5])
if result_md5 != target_md5:
  print("Bad key...")
  sys.exit(1)

result = base64.b64decode("".join(result))
d = open("atad", "wb")
d.write(result)
d.close()
sys.exit(0)
```

Listing 1 – decrypt.py

Ce script python va donc demander une clef de 16 octets et s'en servir pour déchiffrer le fichier data. Pour celà, il y a des interactions avec le quatrième fichier de l'archive sp.ngr par le biais du module python dev qui n'est pas fourni. Il va donc falloir analyser le fichier s.ngr, et développer un module python pour communiquer avec ce dernier.

3.2 Analyse du fichier s.ngr

Il semble donc que le fichier s.ngr va jouer un rôle très important dans cette étape du challenge. Nous essayons donc d'obtenir plus d'informations :

```
$ file s.ngr
s.ngr: data
$ head -2 s.ngr
XILINX-XDB 0.1 STUB 0.1 ASCII
XILINX-XDM V1.6e
```

Des recherches sur Internet montrent que les fichiers ngr sont des fichiers destinés à être lu avec le RTL Viewer de Xilinx ISE. C'est un des fichiers générés par Xilinx [1] lors de la synthèse des sources VHDL. Il va donc falloir télécharger Xilinx ISE et charger ce fichier dans le RTL Viewer pour avoir une idée de ce qui nous attend. Après plusieurs manipulations visant à faire charger le fichier au RTL Viewer, nous obtenons le résultat suivant :

1. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_using_xst_for_synthesis.htm

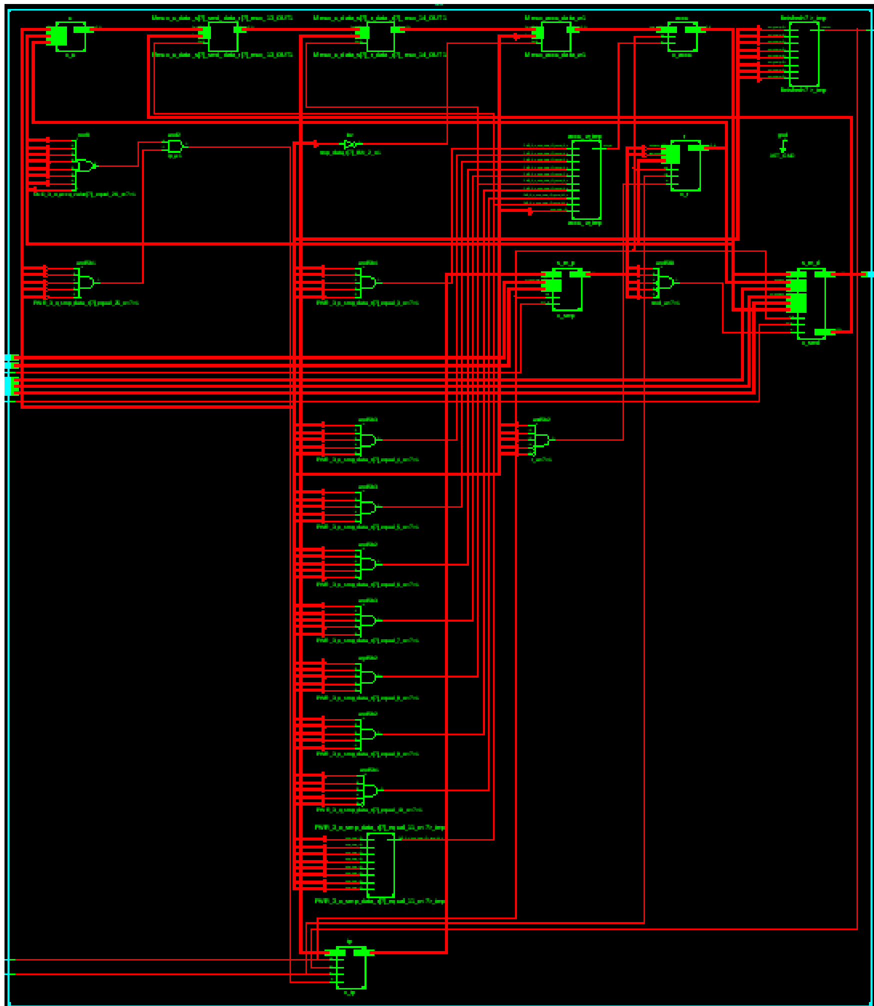

FIGURE 3 – Visualisation du fichier s.ngr

Nous avons donc un schéma de FPGA à base de portes logiques qui a plusieurs entrées et sorties :

- Entrées
  - Bus de 8 bits
    - smp_ctrl_address_w
    - smp_ctrl_data_w
    - smd_ctrl_address_r
    - smd_ctrl_address_w
    - smd_ctrl_data_w
  - Signal de 1 bit
    - smp_ctrl_w
    - smd_ctrl_w
    - clk
    - rst

  séparément
- Sorties

- Bus de 8 bits
  - smd_ctrl_data_r
- Signal de 1 bit
  - finished

Pour comprendre le fonctionnement de l'architecture qui nous est proposée, nous allons analyser les éléments présents dans le schéma séparément (nous les appelerons module).

3.2.1 Analyse des éléments du FPGA

Module u : Unité Arithmétique et Logique

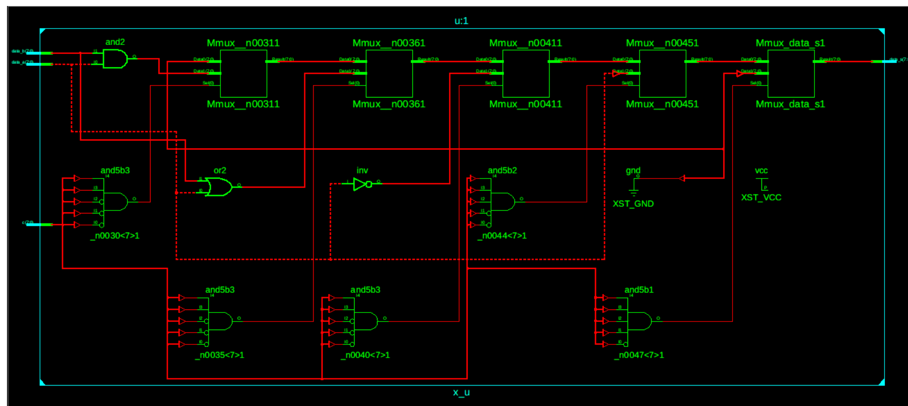

FIGURE 4 – Module u du FPGA

Ce module a trois entrées (data_a(7..0), data_b(7..0), c(7..0)) et une sortie (data_s(7..0)). Il comporte des portes and5 qui vont tester la valeur des 5 bits de poids fort de c et des muxs qui prennent deux entrées et ressortent l'une ou l'autre en fonction du sélecteur (résultat d'un and5). Ainsi, en fonction de la valeur des 5 bits de poids fort de c, une opération va être effectuée sur data_a et data_b.

On peut résumer ceci de cette manière :

- c=0b10001xxx =>data_s = data_a & data_b
- c=0b10010xxx =>data_s = data_a | data_b
- c=0b10100xxx =>data_s = not(data_a)
- c=0b11100xxx =>data_s = 0x80|data_a // met le bit de poids fort à 1
- c=0b11011xxx =>data_s = data_a «1

Le module u est donc une Unité Arithmétique et Logique simplifiée

Module ip : Pointeur d'instruction

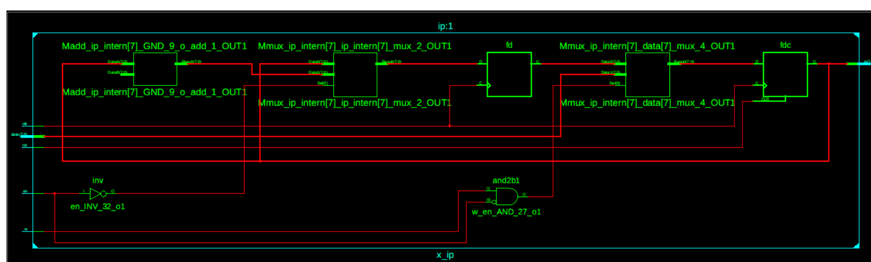

Figure 5 – Module ip du FPGA

Ce module comporte cinq entrées (data(7..0), clk, rst, en, w) et une sortie (ip(7..0)). Après analyse des muxs et des portes logiques présentes à l'intérieur du module, il est possible de déduire le fonctionnement suivant :
- Si 'en' vaut 1, la sortie reste constante.
- Si 'en' vaut 0.
  - Si 'w' vaut 0, la sortie est incrémentée de 1.
  - Si 'w' vaut 1, la sortie correspond à l'adresse passée en entrée dans data.

Le fonctionnement de ce module, est donc (comme son nom l'indique) celui d'un pointeur d'instruction. Il incrémente l'adresse de l'instruction courante de 1 ou lui donne une valeur passée en argument.

Module r : Registre de 8 octets

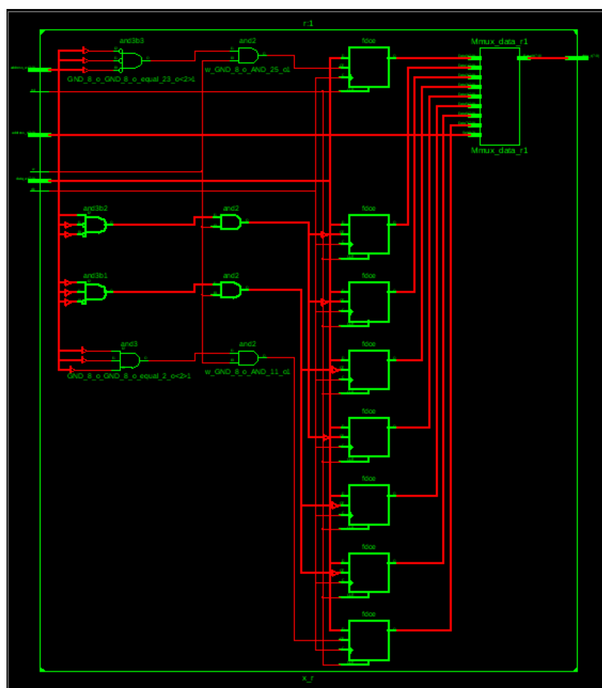

Figure 6 – Module r du FPGA

Ce module présente six entrées (address_w(2..0), address_r(2..0), data_w(7..0), w, clk, rst) et une sortie (data_r(7..0)). Il utilise 8 balances flip flop qui changent de valeur uniquement lorsqu'une de ses entrées est activée. Cette entrée est un 'et logique' entre w et une valeur particulière comprise entre 0 et 7 selon la bascule. Les valeurs stockées dans chaque bascule arrivent sur un mux, le selecteur du mux est l'entrée address_r. Il est possible de distinguer deux modes de fonctionnement :
- Si 'w' vaut 1 =>address_w code un entier entre 0 et 8. La bascule activée par cet entier prend la valeur de data_w.
- Si 'w' vaut 0 =>La sortie (data_r) prend la valeur de la bascule correspondant à la valeur de address_r.

Le fonctionnement de ce module correspond donc à celui d'un registre : il est possible d'écrire à une adresse parmi 8 et de lire cette valeur plus tard.

Module smp

Le module smp étant trop volumineux pour être présenté visuellement, il sera analysé de façon écrite uniquement.

Ce module présente une entrée interne (address_r(7..0)), et quatre entrées (ctrl_address_w(7..0), ctrl_data_w(7..0), ctrl_w, clk) qui viennent directement de l'extérieur (envoyées depuis le module python dans notre cas). Ces entrées serviront à envoyer des données pour l'initialisation du FPGA. Le module comporte aussi une sortie (data_r(7..0)).

En regardant le fonctionnement interne du module, on trouve 256 portes AND, 256 bascules, un mux de 256 entrées. Le module smp est donc une mémoire de 256 octets qui peut être écrite uniquement depuis l'extérieur du FPGA grâce aux entrées ctrl_address_w, ctrl_data_w et ctrl_w. En revanche, elle peut être lue uniquement de façon interne. En effet, on peut lire sur la sortie data_r la valeur de la mémoire à l'adresse address_r.

Module smd

Comme le module précédent, smd est trop volumineux pour être présenté visuellement.

Ce module reprend les caractéristiques du module précédent mais présente plus d'entrées et de sorties afin d'être accessibles depuis l'extérieur, et en interne. Ainsi, il a les entrées suivantes branchées directement à l'extérieur : smd_ctrl_address_r(7..0), smd_ctrl_address_w(7..0), smd_ctrl_data_w(7..0), ctrl_w, clk. Le module smd présente aussi des entrées internes : address_r(7..0), address_w(7..0), data_w(7..0). Et il présente deux sorties dont une interne (data_r(7..0)) et une externe (ctrl_data_r(7..0)).

Les éléments qui composent se module sont les mêmes que ceux de smd, il s'agit une nouvelle fois d'une mémoire de 256 octets. Cependant cette fois ci, cette mémoire est accessible en écriture et lecture depuis l'extérieur du FPGA, mais aussi depuis l'intérieur du FPGA.

Fonctionnement général

Après avoir analysé l'architecture de certains modules plus complexes, il convient d'élargir la compréhension à un cadre plus général.

Certains éléments n'ont pas encore été mentionnés et jouent un rôle dans le fonctionnement global :

- accu : accu joue le rôle de variable. selon les autres paramètres, il prend comme valeur la sortie de l'UAL, une valeur lue dans la mémoire smd, ou une valeur lue dans le registre r.
- accu_w_imp : cet élément est un 'ou logique' entre 10 entrées qui va positionner le bit w de l'accu afin de savoir si sa valeur doit évoluer.
- portes and : il y a plusieurs portes and5 qui vont tester des valeurs sur les 5 bits de poids fort de la sortie de smp. Selon ces valeurs, des bits vont être positionnés pour d'autres modules.

Instruction binaire	Opération réalisée
0xxxxxxx	accu = 0b0xxxxxxx
10000xxx	accu = 0
10001xxx	accu = accu & r[0bxxx]
10010xxx	accu = accu | r[0bxxx]
10011xxx	accu = 0
10100xxx	accu = not accu
10101xxx	accu = r[0bxxx]
10110xxx	r[0bxxx] = accu
10111xxx	if accu==0 : jmp r[0bxxx]
11000xxx	smd[r[0bxxx]] = accu
11001xxx	exit
11010xxx	accu = smd[accu]
11011xxx	accu = accu « 1
11100xxx	accu = accu | 0x80

TABLE 1 – Code machine du processeur

En possession de tous ces éléments il est possible de comprendre le fonctionnement du FPGA : celui d'un processeur.

- Les deux mémoires vont être initialisées
  - smp contient une liste d'instructions : c'est le programme que va exécuter le processeur.
  - smd contient des données qui vont être modifiées.
- Le pointeur d'instruction donne une adresse à smp, et récupère l'instruction en cours.
- l'instruction en cours (5 bits de poids fort) est testée par toutes les portes and et des bits sont positionnés :
  - Si l'instruction affecte accu, le bit w de accu est positionné.
  - Si l'instruction affecte r, le bit w de r est positionné.
  - Si l'instruction affecte smd, le bit w de smd est positionné.
- Dans la plupart des cas, les instructions sont codées sur les 5 bits de poids fort, et les 3 bits de poids faible sont utilisées pour fournir des valeurs.
- Si l'instruction commence par 0, les 7 bits de poids faible sont chargés dans accu

A partir de ces informations, il est possible de déduire le code machine.

3.2.2 Code machine du processeur

Le code machine étant trouvé, l'étape suivante est de se pencher sur l'émulation de ce processeur et sur ses interactions avec le programme decrypt.py.

3.3 Emulation du processeur FPGA

Après avoir remonté l'architecture du processeur, nous devons produire un simulateur pour exécuter son code. Il s'agit d'un processeur Harvard, avec données et code séparés. Les données sont dans SMD et le programme dans SMP.

Le fichier decrypt.py utilise des primitives simples pour accéder au processeur :

– chargement de code ;
– chargement de données ;
– récupérations de données ;
– démarrage du processeur.

Le simulateur reprend de manière simplifiée l'architecture du processeur avec deux tableaux de mémoire, un vecteur de registres et un équivalent fonctionnel des instructions processeur. Son code est disponible à l'annexe 7.2.1.

La spécificité identifiée lors du fonctionnement de la simulation, c'est que decrypt.py charge des données accompagné d'une clef et que lors de la relecture, celle-ci ne doit pas être renvoyée.

Une exécution du code processeur sert à déchiffrer un bloc de 224 octets de données.

3.4 Recherche de la clef de déchiffrement

Une fois le module python dev développé, le programme decrypt.py fourni dans l'archive est fonctionnel. Il faut donc travailler à la recherche de la clef de déchiffrement. Pour celà, un indice est fourni par le script python :

```
1 result = base64.b64decode("".join(result))
```

Le fichier déchiffré sera donc au format base64. C'est à dire que le charset d'arrivée sera [A-Za-z0-9+/=\n]. L'autre indice donné est le format d'entrée de la clef qui est en hexadécimal, la clef en paramètre correspondra donc au charset [a-f0-9], ce qui réduit le nombre de possibilités lors de l'approche en force brute.

Pour obtenir plus d'informations, il est intéressant d'analyser les différences entre fichiers déchiffrés avec des clefs différentes. Le script decrypt.py est adapté afin de déchiffrer seulement le premier block de 224 octets. Il est alors possible de générer ce bloc pour la clef 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA' et la clef 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB'. Le programme vbindiff montre leurs différences :

```
dump-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0000 0000: 7E A4 05 10 74 E5 01 07  90 0E DB F2 4F 8D 82 9C  ~...t... ....O...
0000 0010: 53 C3 16 04 74 E6 0E 51  B8 27 DF F6 77 8E 86 98  S...t..Q .'..w...
0000 0020: 6B C3 24 04 76 F1 0E 40  A6 53 C7 F7 7F 8E BC 82  k.$.v..@ .S......
0000 0030: 68 C1 3C 04 64 90 76 5D  A9 33 D3 F3 4C B4 A8 C1  h.<.d.v] .3..L...
0000 0040: 68 CC 27 0F 71 92 09 62  A2 50 FD D4 27 B0 97 A7  h.'.q..b .P..'...
0000 0050: 06 C6 29 3E 57 C0 7D 64  9F 29 D6 C4 5D 81 98 A3  ..)>W.}d .)..]...
0000 0060: 5E F7 07 29 4B F7 22 60  C2 03 A3 D8 44 81 97 A7  ^..)K."` ....D...
0000 0070: 66 F0 26 22 51 FB 18 07  B6 3A C6 D0 5B B9 97 A4  f.&"Q... .:..[...
0000 0080: 55 CE 03 04 4E C6 08 60  8A 29 D6 D0 40 AD AA A8  U...N..` .)..@...
dump-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
0000 0000: FE A4 05 10 74 E5 01 07  90 0E DB F2 4F 8D 82 9C  ....t... ....O...
0000 0010: D3 C3 16 04 74 E6 0E 51  B8 27 DF F6 77 8E 86 98  ....t..Q .'..w...
0000 0020: EB C3 24 04 76 F1 0E 40  A6 53 C7 F7 7F 8E BC 82  ..$.v..@ .S......
0000 0030: E8 C1 3C 04 64 90 76 5D  A9 33 D3 F3 4C B4 A8 C1  ..<.d.v] .3..L...
0000 0040: E8 CC 27 0F 71 92 09 62  A2 50 FD D4 27 B0 97 A7  ..'.q..b .P..'...
0000 0050: 86 C6 29 3E 57 C0 7D 64  9F 29 D6 C4 5D 81 98 A3  ..)>W.}d .)..]...
0000 0060: DE F7 07 29 4B F7 22 60  C2 03 A3 D8 44 81 97 A7  ...)K."` ....D...
0000 0070: E6 F0 26 22 51 FB 18 07  B6 3A C6 D0 5B B9 97 A4  ..&"Q... .:..[...
0000 0080: D5 CE 03 04 4E C6 08 60  8A 29 D6 D0 40 AD AA A8  ....N..` .)..@...
```

Figure 7 – Différences entre deux fichiers déchiffrés

Sur cette comparaison, il est possible de constater qu'un changement d'un octet dans la clef de 16 octets affecte seulement un octet sur 16 dans le fichier déchiffré. Cet élément est très important car il est possible de trouver chaque octet de la clef de manière indépendante. Cela réduit le nombre de possibilités à tester de 256^{16} à 256.

A partir de ces informations, il est possible de trouver la clef assez rapidement. Nous avons procédé de la façon suivante :

- Chiffrer le premier bloc avec 256 clefs contenant 16 fois le même octet.
- Si tous les bits $n+k\times 16$ appartiennent au charset base64 ajouter la valeur testée à l'ensemble de possibilités pour le nième octet.
- A la fin du premier bloc, relancer sur le deuxième bloc, avec seulement l'ensemble des valeurs possibles pour le premier bloc.
- Itérer jusqu'à l'obtention d'une clef unique.

Le script associé est à l'annexe 7.2.2. La clef obtenue est :

```
1 25 a1 6d 1e d3 23 ac 65 c6 58 ef 16 bc dc 83 e6
```

Une fois cette clef obtenue, nous pouvons exécuter la simulation sur l'intégralité des données. Nous obtenons en sortie un fichier `atad` qui s'avère être un document Postscript.

4 Document Postscript chiffré

Le document Postscript obtenu est structuré en 2 grandes parties :

- un bloc de flux binaires encodés en hexadécimal ;
- du code Postscript de décodage.

Ces 4 flux contiennent du code Postscript chiffré pour deux d'entre eux, un document chiffré dont la sortie claire doit être obtenue avec la bonne clef et une table de condensats MD5 qui sert à valider le contenu clair déchiffré.

Postscript est un langage standardisé. Il s'agit d'un langage à pile exploitant aussi la notion de flux comme stockage hors pile de données pour éviter de saturer la mémoire allouée à la pile.

Nom	Description
I1	Document chiffré
I2	Code Postscript chiffré
I3	Table de condensats MD5
I4	Code Postscript chiffré

Table 2 – Descriptions des flux binaires du document Postscript

Le code Postscript présent utilise une procédure non standard typique de l'interpréteur Ghostscript : `shellarguments`.

Un essai d'exécution avec Ghostscript renvoie le résultat suivant :

```
1 gs -dNODISPLAY atad.ps
  GPL Ghostscript 9.06 (2012-08-08)
```

```
3 Copyright (C) 2012 Artifex Software, Inc.  All rights reserved.
  This software comes with NO WARRANTY: see the file PUBLIC for details.
5 missing '--' preceding script file
  usage: gs -- script.ps key
7
  $ gs -dNODISPLAY -- atad.ps
9 GPL Ghostscript 9.06 (2012-08-08)
  Copyright (C) 2012 Artifex Software, Inc.  All rights reserved.
11 This software comes with NO WARRANTY: see the file PUBLIC for details.
  no key provided
13 usage: gs -- script.ps key

15 $ gs -dNODISPLAY -- atad.ps 123
  GPL Ghostscript 9.06 (2012-08-08)
17 Copyright (C) 2012 Artifex Software, Inc.  All rights reserved.
  This software comes with NO WARRANTY: see the file PUBLIC for details.
19
  $
```

Le script a besoin d'une clef, fournie en paramètre sur 16 octets codés en hexadécimal (`readhexstring pop dup length 16 eq` appliqué au flux du paramètre). Pour la trouver il va falloir étudier le fonctionnement interne du programme.

La première étape va consister à extraire les portions de code issues des flux binaires encodés dans le fichier.

4.1 Extraction du code chiffré

Le script présente deux appels à la fonction `exec`. Elle est précédée à chaque fois de la fonction `cvx`. Ce couple cvx exec sert à rendre exécutable puis exécuter du code Postscript stocké sous forme de littéral sur la pile. C'est là que nous devons instrumenter le code pour extraire ce code exécutable supplémentaire non visible directement dans le fichier.

En utilisant la fonction `error` déclarée dans le script, nous rajoutons une impression du code. `... cvx exec` devient `... dup error cvx exec`.

En faisant cela nous nous rendons compte que le code est chiffré. Les deux flux de code sont chiffrés avec deux clefs différentes. En testant octet par octet l'influence de la clef en paramètre, nous identifions que les 2 premiers octets servent à déchiffrer le premier flux de code et les 2 octets suivants l'autre flux.

Faute de déterminant plus précis, nous effectuons une recherche par force brute sur 16 bits en utilisant comme discriminant l'absence de caractères non imprimables ainsi que la présence de la chaine `" dup "`, qui est une des instructions les plus utilisées de Postscript, donc sa présence est très probable sur un échantillon de code Postscript représentatif.

Après quelques minutes de recherche, nous obtenons ainsi les 32 premiers bits de clef qui permettent d'extraire convenablement le code Postscript supplémentaire.

```
  $ gs -dNODISPLAY -- atad.ps bac9f7a8000000000000000000000000
2 GPL Ghostscript 9.06 (2012-08-08)
  Copyright (C) 2012 Artifex Software, Inc.  All rights reserved.
4 This software comes with NO WARRANTY: see the file PUBLIC for details.
  Key is invalid. Exiting ...
```

4.2 Étude du mécanisme de déchiffrement et validation

Le code Postscript étant correctement déchiffré, le script est capable de nous dire que notre clef est incorrecte. Le code déchiffré contient notamment une implémentation de MD5 en Postscript ainsi que du code de déchiffrement non identifié.

Sans étudier plus avant le mécanisme de déchiffrement, nous constatons qu'il utilise la clef en paramètre par blocs de 16 bits et compare son résultat de déchiffrement au md5 extrait d'un autre flux. Étant donné l'espace de clefs restreint à traiter, l'approche force brute semble la plus efficace en terme d'énergie humaine consommée.

4.3 Recherche de la clef complète

Le script Postscript est instrumenté pour éditer sur la sortie standard son md5 calculé ainsi que le résultat de son test. Un script python est créé pour générer 65536 scripts shells qui exécutent Ghostview avec le morceau de clef à tester. Cela permet de tester en parallèle un grand nombre de clefs sur plusieurs processeurs.

Une fois 16 bits de clef trouvé, on recherche le bloc de clef suivant. Soit 65536×6 tests. En étudiant plus avant le script, il aurait été possible de limiter à 65536 tests en testant tous les blocs en même temps. Cela aurait nécessiter de passer plus de temps à comprendre le script, ce qui n'a pas été jugé nécessaire.

Une fois la clef obtenue, le script génère un fichier `output.bin` qui se trouve être un fichier au format vCard.

5 Fichier vCard

Le fichier vCard obtenu contient des entrées génériques ainsi qu'un contact spécifique au challenge SSTIC.

```
1 BEGIN:VCARD
  VERSION:2.1
3 FN:Challenge SSTIC
  N:Challenge;SSTIC
5 ADR;WORK;PREF;QUOTED-PRINTABLE:;Campus Beaulieu;Rennes
  TEL;CELL:
7 EMAIL;INTERNET:sys_socketpair stub_fork sys_socketpair sys_getsockopt
      sys_socketpair sys_ptrace sys_shutdown sys_ptrace sys_getsockopt
      sys_bind sys_getuid sys_bind sys_ptrace sys_getsockname sys_ptrace
      stub_fork stub_fork sys_getpeername sys_setsockopt sys_getrusage
      sys_sysinfo sys_getsockname sys_shutdown sys_getsockopt sys_getuid
      sys_sysinfo sys_getsockopt sys_getrlimit sys_setsockopt
      sys_shutdown stub_clone sys_times sys_shutdown sys_getrusage
      sys_socketpair sys_setsockopt stub_clone sys_getpeername
      sys_socketpair stub_clone sys_semget sys_sysinfo sys_getgid
      sys_getrlimit sys_getegid sys_getegid sys_ptrace sys_getppid
      sys_syslog sys_ptrace sys_sendmsg sys_getgroups sys_getgroups
      sys_setgroups sys_setuid sys_sysinfo sys_sendmsg sys_getpgrp
      sys_setregid sys_syslog
```

```
END:VCARD
```

Dans un premier temps, nous avons remarqué que la fréquence des symboles correspond à celle d'une adresse mail terminant par @challenge.sstic.org.

Cela permet de deviner un certain nombre de caractères, mais pas leur intégralité.

Les symboles utilisés sont ceux déclarés dans les headers de Linux (entre autres, il s'agit pour la plupart d'appels système POSIX). En utilisant le fichier `include/asm-generic/unistd.h` distribué avec les sources de Linux, nous obtenons des nombre correspondant au code ASCII par lequel remplacer chaque symbole.

Cela nous permet déjà de constater que notre hypothèse sur la fréquence est bonne, ainsi que d'obtenir l'adresse email complète.

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

6 Remerciements

Merci à Vincent Fargues (`vincent.fargues@thalesgroup.com`) pour les appels à un ami, et plus largement à l'équipe de Thales SecureLab pour le soutien.

7 Annexes

7.1 Scripts de l'étape 1

7.1.1 Makefile

```
all: indices sstic.tar.gz

dump.bin:
	wget http://static.sstic.org/challenge2013/dump.bin

list-icmp-ttl: dump.bin
	tshark -r dump.bin -T fields -e ip.ttl icmp > $@

list-frame-delta: dump.bin
	tshark -r dump.bin -T fields -e frame.time_delta icmp |cut -c 1
	    > $@

list-ip-dsfield: dump.bin
	tshark -r dump.bin -T fields -e ip.dsfield icmp > $@

indices: dump.bin
	tcpflow -r dump.bin -C "tcp port 1234" > $@


sstic.tar.gz-chiffre.b64: dump.bin
	tcpflow -r dump.bin -C "tcp port 60733" > $@

sstic.tar.gz-chiffre: sstic.tar.gz-chiffre.b64
	base64 -d < $< > $@
```

```
25 sstic.tar.gz: sstic.tar.gz-chiffre list-icmp-ttl list-frame-delta list-
       ip-dsfield
           python extract-entropy-from-lists.py
```

Makefile

7.1.2 Recherche de clef

```
   #! /usr/bin/python
 2
   from Crypto.Cipher import AES
 4 import struct
   import md5
 6 import itertools

 8 total_checks = 0
   tested_keys=dict()
10
   def printstats(l):
12   print len(l)
     stats=dict()
14   for i in xrange(max(l)+1):
       stats[i]=0
16   for i in l:
      try:
18     stats[i] += 1
      except KeyError:
20     stats[i] = 1
     for i in stats.keys():
22    print "%02d: %-10s %d" % (i, "*"*stats[i], stats[i])

24 def trykey(k):
     global total_checks
26   global tested_keys
     key = k
28   if key in tested_keys:
       tested_keys[key] += 1
30     return False
     tested_keys[key] = 1
32   iv = "76C128D46A6C4B15B43016904BE176AC".decode("hex")
     cipher = AES.new(key, AES.MODE_CBC,iv)
34   fi = open("sstic.tar.gz-chiffre", "rb")
     msg = cipher.decrypt(fi.read(16))
36   fi.close()
     check = "61c9392f617290642f9a12499de6b688"
38   out=0
     if msg[:2] == "\x1f\x8b":
40     print "GZIP header"
       print msg[-32:].encode("hex")
42     out=check
     total_checks += 1
44
     if out == check:
46     from os import system
       system("openssl aes-256-cbc -d -in sstic.tar.gz-chiffre -out sstic.
           tar.gz -K %s -iv %s" % (key.encode("hex"), iv.encode("hex")))
48     ff=open("sstic.tar.gz","rb")
       c=md5.md5(ff.read()).hexdigest()
50     if c == check:
         print "***************** \o/ SUCCESS"
```

```
52        print "***************** \o/ SUCCESS"
          print "***************** \o/ SUCCESS"
54        print "***************** \o/ SUCCESS"
          print "***************** \o/ SUCCESS"
56        print "***************** \o/ SUCCESS"
          exit()
58
    print "Try Again"
60  return False

62 def keyfrombitlist(bl):
    S=str.join("",map(str,bl))
64  ho= ("%064s"%(hex(int(S,2))[2:][:-1])).replace(" ","0")
    print ho
66  pkey=ho.decode("hex")
    return pkey
68
   def try_multi_permuts(klist, plist):
70  k1 = klist
    permuts = plist
72  for (pa,pb,pc,pd) in permuts:
      L=[]
74    print "Permut= %s ---------------" % (str((pa,pb,pc,pd)))
      for kt in k1:
76      n=[kt[pa],kt[pb],kt[pc],kt[pd]]
        L+=n
78    #printstats(L)
      potkey = keyfrombitlist(L)
80    trykey(potkey)

82 f=open("list-frame-delta","r")
   deltas = map(lambda x: (int(x)-1)^0, map(str.strip, f.xreadlines())
       [1:])
84 f.close()
   print "deltas --"
86 print deltas
   printstats(deltas)
88
   deltasxor1 = list()
90 for d in deltas:
    deltasxor1 += [d^1]
92
   # -------------------------
94
   f=open("list-icmp-ttl","r")
96 ttls = map(lambda x: ((int(x[0]))-1)^0, map(str.strip, f.xreadlines())
       [:-1])
   f.close()
98
   print "ttls --"
100 print ttls
   printstats(ttls)
102
   # -------------------------
104
   f=open("list-ip-dsfield")
106 dsfs = map(lambda x: ((int(x)>>1)-1)^0, map(str.strip, f.xreadlines())
       [:-1])
   f.close()
108 print "dsfields --"
   print dsfs
110 printstats(dsfs)
```

```
112 dsfxor1 = list()
    for d in dsfs:
114   dsfxor1 += [d^1]

116 permuts=list()
    for p in itertools.permutations([0,1,2,3,4,5],4):
118   permuts += [p]

120 ttlo=ttls
    for p in itertools.permutations([0,1,2,3]):
122   ttls=list()
      print "MiX: (0, 1, 2, 3)=>%s" % (str(p))
124   for t in ttlo:
        n = p[t]
126     ttls += [n]
      k1= [(i,j>>1,j&1,k,j2,k2) for i,j,k,j2,k2 in zip(deltas,ttls,dsfs,
          deltasxor1,dsfxor1)]
128   print k1

130   try_multi_permuts(k1, permuts)

132 print "Total checks: %d" % (total_checks)
    print "Key count: %d" % (len(tested_keys))
134 print "Max key hit: %d" % (max(tested_keys.values()))
```

extract–entropy–from–lists.py

7.2 Scripts de l'étape 2

7.2.1 Simulateur du processeur

```
    """
2   @author jybu
    'mulator de folaie
4   """
    _r = [0] * 8
6   _acu = 0
    _ip = 0
8   _smd = [0] * 256
    smp = [0] * 256
10  _finished = 0
    _state = (_r,_acu,_ip,_smd,_finished)
12  offsetjump = dict()
    offsetjump = {0: 0, 64: 1, 102: 5, 44: 2, 109: 6, 113: 7, 83: 3, 87: 4}
14  coverage=dict()
    jumps=dict()
16

18  def do_print(s):
      print s
20
    def dont_print(s):
22    pass

24  printer=dont_print

26  def track_jump(f, t):
      return
28    global jumps
      try:
```

```
30      jumps[(f,t)] += 1
    except KeyError:
32      jumps[(f,t)] = 1

34  def track_coverage(code_addr):
      return
36    global coverage
      try:
38      coverage[code_addr] += 1
      except KeyError:
40      coverage[code_addr] = 1

42  def print_stats():
      return
44    global coverage
      global jumps
46    global smp

48    printer("coverage %d / %d" % (len(coverage.keys()), len(smp)))
      printer( "jumps: %d" % (len(jumps.keys())))
50    l = jumps.items()
      l.sort(lambda (a,b),(c,d): cmp(b,d))
52    for k,v in l:
        printer("%03d->%03d:  %5d" % (k[0],k[1],v))
54
    def _rd_acu(reg,(r,acu,ip,smd,finished)):
56    nr = r
      nr[reg] = acu
58    return ("r%d   = acu" % (reg), (nr,acu,ip,smd,finished))

60  def _smd_acu(reg,(r,acu,ip,smd,finished)):
      nsmd = smd
62    nsmd[r[reg]] = acu
    #  printer("SMD[%x] <- %x" % (r[reg], acu))
64    return ("smd[r%d]= acu" % (reg), (r,acu,ip,nsmd,finished))

66  def _acu_smd(reg,(r,acu,ip,smd,finished)):
    #  print "%x <- SMD[%x]" % (smd[acu], acu)
68    return ("acu  = smd[acu]", (r,smd[acu],ip,smd,finished))


70
    def _jump(reg,(r,acu,ip,smd,finished)):
72    global offsetjump
      if acu == 0:
74      try:
          offsetjump[r[reg]] += True
76      except KeyError:
          offsetjump[r[reg]] = 1
78
      return ("ip   = r%d (if acu==0)" % (reg), (r,acu,r[reg] if acu==0
          else ip,smd,finished))
80
    instructions = dict()
82  instructions[0b10000] = lambda reg,(r,acu,ip,smd,finished): ("XXX acu
        = 0 ; alt0", (r,0,ip,smd,finished))
    instructions[0b10001] = lambda reg,(r,acu,ip,smd,finished): ("acu &= r%
        d" % (reg), (r,(acu & r[reg])&0x0ff,ip,smd,finished))
84  instructions[0b10010] = lambda reg,(r,acu,ip,smd,finished): ("acu |= r%
        d" % (reg), (r,(acu | r[reg])&0x0ff,ip,smd,finished))
    instructions[0b10011] = lambda reg,(r,acu,ip,smd,finished): ("XXX acu
        = 0 ; alt1", (r,0,ip,smd,finished))
```

```
instructions[0b10100] = lambda reg,(r,acu,ip,smd,finished): ("acu  = ~
    acu", (r,(~acu & 0xff),ip,smd,finished))
instructions[0b10101] = lambda reg,(r,acu,ip,smd,finished): ("acu  = r%
    d" % (reg), (r,r[reg],ip,smd,finished))
instructions[0b10110] = _rd_acu
instructions[0b10111] = _jump #lambda reg,(r,acu,ip,smd,finished):
instructions[0b11000] = _smd_acu
instructions[0b11001] = lambda reg,(r,acu,ip,smd,finished): ("finished"
    ,(r,acu,ip,smd,1 if reg==0 else 0))
instructions[0b11010] = _acu_smd #lambda reg,(r,acu,ip,smd,finished):
    ("acu  = smd[acu]", (r,smd[acu],ip,smd,finished))
instructions[0b11011] = lambda reg,(r,acu,ip,smd,finished): ("acu  =
    acu << 1", (r,((acu << 1))&0x0ff,ip,smd,finished))
instructions[0b11100] = lambda reg,(r,acu,ip,smd,finished): ("acu &=
    mouaif ", (r,(0x80|acu)&0xff,ip,smd,finished))

def decode_and_execute_instr(instr, state):
  (r,acu,ip,smd,finished) = state
  reg = instr & 0b0111
  i = (instr >> 3) & 0b11111
  if i in instructions:
    return instructions[i](reg,state)
  if (i & 0b10000) == 0:
    imm = instr&0b01111111
    return ("acu  = 0x%x (%d)" % (imm,imm), (r,imm,ip,smd,finished))

  print "-INVAL-----------------"
  print " %d  %x  %s" % (instr,instr,bin(instr))
  return ("illegal", state)

def init(fname):
  global _state
  r = [0] * 8
  acu = 0
  ip = 0
  smd = [0] * 256
  smp = [0] * 256
  finished = 0
  _state = (r,acu,ip,smd,finished)

def send_smd((k,l,d)):
  global _state
  (r,acu,ip,smd,finished) = _state
  nd = d
  if l<224:
    nd += "\x00" * (224-l)
  smd = k+[l]+map(ord, list(nd))
#  print smd
  _state = (r,acu,ip,smd,finished)
#  print _state[3]
#  print "-blabla----"

def send_smp(ilist):
  _r = [0] * 8
  _acu = 0
  _ip = 0
  _smd = [0] * 256
  _finished = 0
  state2 = (_r,_acu,_ip,_smd,_finished)
  global smp
#  print "-*- Uploading code ... ------"
  for i in xrange(len(ilist)):
```

```
    smp[i] = ilist[i]
    (t,_) = decode_and_execute_instr(ilist[i], state2)
#     if i in offsetjump:
#         print "\nlabel%d:" % (i)
#     print "%03d: %02x %s" % (i,ilist[i], t)
#     if t.find("ip    =") == 0:
# print ("---------------------------")
#  print "-----------------------------"
  global offsetjump
#  print offsetjump
  #exit()

def start():
  global _state
  lstate=list(_state)
  lstate[4] = 0 #finished
  lstate[2] = 0 #ip
  _state=tuple(lstate)

def wait_finished():
  global _state
  state = _state
  (r,acu,ip,smd,finished) = state
  exe=0
  while not finished:
    isajump=False
    track_coverage(ip)
    i = smp[ip]
    (txt, nstate) = decode_and_execute_instr(i, state)
    l=list(nstate)
    if l[2] == ip:
      l[2] += 1
      l[2] &= 0xff
      nstate = tuple(l)
    else:
      track_jump(ip, l[2])
      isajump=True
    state = nstate
    printer( "%05d||%03d: %02x   %25s // acu=%02x r=%s smd[0x10:0x30]=%s" % (exe, ip, i, txt, state[1], " ".join(["%02x"%(x) for x in state[0]]), " "))#" ".join(["%02x"%(y) for y in smd[0x10:0x30]])) )
    (r,acu,ip,smd,finished) = state
    if isajump: printer("")
    exe +=1
    if False: #exe > 0xffff:
      global offsetjump
#       print "OOOffsets = %s" % (" ".join(["%02x(%03d)"%(j,j) for j in offsetjump.keys()]))
#       print offsetjump
      print_stats()
      exit()
      return
      raise Exception("plop")
  print "nb rounds %d" % (exe)

def get_data():
  global _state
  return map(chr, _state[3][0x11:0x11+224])
```

dev.py

7.2.2 Script de brute-force

```
#!/usr/bin/python


import dev
import smp

base64 = set()
b64s="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
    =+/\n\r "
for i in list(b64s):
  base64.add(i)
if len(base64) != 65+3:
  raise Exception("Error")

d = open("data", "rb").read()

def decode_one_block_key(k,o):
  key = int(k, 16)
  key = [(key >> (i * 8)) & 0xff for i in range(16)]

  result = []
  dev.init("sp.ngr")
  print "---------"
  print o
  smd = d[o : (o + 224)]
  smd = (key, len(smd), smd)
  dev.send_smd(smd)
  dev.send_smp(smp.smp)
  dev.start()
  dev.wait_finished()
  result = result + dev.get_data()
  return result

def check_b64_proof(res):
  l=[True]*16
  for i in xrange(0,len(res)):
    r=res[i]
    if r not in base64:
      l[i%16] &= False
  return l


for p in range(0,224,16):
  print " ".join([("%02x" % (ord(x)) ) for x in d[p:p+16] ])
print "check"
keybytes = [None]*16
for b in xrange(16):
  keybytes[b] = set()


finish=0
charset = range(256)
first_octet=0
while finish==0:
  keybytes = [None]*16
  for b in xrange(16):
    keybytes[b] = set()
  print "CHARSET--------------"
  print charset
  for i in charset:
```

```
    print "- %02x" % (i)
61    r=decode_one_block_key(("%02x" % (i))*16,first_octet)
    l=check_b64_proof(r)
63    for b in range(len(l)):
      if l[b]:
65        keybytes[b].add(i)
    print "keybytes ----"
67    print keybytes
  charset=list()
69  finish=1
  first_octet +=224
71  for i in range(16):
    charset+=list(keybytes[i])
73    if len(keybytes[i]) > 1:
      finish=0
75

77 print keybytes
```

decrypt_bf.py