Solution du challenge SSTIC 2013

Pierre Bienaimé

6 avril 2013

Table des matières

Introduction

J'aime le challenge SSTIC !

Chaque année, c'est une formidable occasion de découvrir qu'on est complètement nul dans beaucoup de domaines. Et chaque année, c'est un motivateur puissant qui nous force à assimiler de nouvelles connaissances afin de devenir un peu moins nul.

Regardons ensemble ce que nous réservait cette édition 2013 [1].

1 Niveau 1 : Pcap, AES et canal caché

1.1 Analyse préliminaire

Le challenge commence en récupérant un fichier *dump.bin* [2] qui est décrit comme étant une trace réseau. Le but du jeu, quant à lui, est toujours le même : analyser ce fichier pour y retrouver une adresse e-mail au format `...@challenge.sstic.org`.

Tout d'abord, il faut toujours garder à l'esprit que les concepteurs de challenge SSTIC font partie d'une espèce rare qui est réputée pour sa ruse, sa fourberie et son intelligence malveillante mise au service d'un sadisme informatique. De plus, puisque « *Rien n'est vrai, tout est permis* », il peut sembler sage de commencer par vérifier que cette trace réseau en est bien une.

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

Il s'agit donc bien d'un pcap. En regardant les chaînes de caractères contenues dans ce fichier avec la commande `strings`, on trouve une foultitude de choses intéressantes :

– Un petit texte d'introduction avec des consignes
– Une bannière de client FTP

```
CLNT NcFTP 3.2.2 linux-x86-glibc2.9
```

– Une connexion avec des identifiants hautement sécurisés

```
USER sstic
PASS sstic
```

1. **NB** : Cette solution faisant exactement 42 pages, peut-être résout-elle non seulement le challenge SSTIC mais également, à mon insu, tout le reste.
2. http://static.sstic.org/challenge2013/dump.bin

– Des références à un fichier nommé *sstic.tar.gz-chiffre*
– Pour finir, de longues chaînes encodées en base64.

Il est maintenant temps d'ouvrir notre pcap avec `wireshark` pour mieux comprendre à quoi correspond ce que l'on vient de trouver. Il renferme 228 paquets réseau que l'on peut facilement séparer en 3 parties :

1. La première partie est une connexion TCP qui fait transiter du texte brut vers le port 1234. On y retrouve notre consigne.
2. La deuxième partie est une suite de paquets *ICMP Echo Request*, autrement dit, des ping.
3. Enfin, la troisième partie de la capture est un transfert de fichier par FTP. Le fichier en question s'appelle, comme prévu, *sstic.tar.gz-chiffre*.

1.2 Dissertation philosophique autour de la consigne

Ou pas. Voici le contenu de cette fameuse consigne :

```
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.
```

Posséder une consigne dans un challenge SSTIC est un luxe certain. Celle-ci nous informe qu'une archive contenant un carnet d'adresses a été chiffrée en AES avant d'être transférée via FTP. La clé de déchiffrement, quant à elle, a été envoyée par « *canaux cachés* ».

1.3 Récupération du fichier transféré par FTP

Il existe certainement nombre de méthodes élégantes pour extraire d'un pcap un fichier transféré par FTP. Une option avancée ou un plugin Wireshark, un script Scapy ou encore une technique digne d'un épisode de « *Les Experts : Rennes* » qui consiste à rejouer le pcap vers un vrai serveur FTP.

Personnellement, j'ai opté pour la méthode plébiscitée par les étudiants contraints de réaliser un exposé, par les écrivains en manque d'inspiration et par les développeurs soucieux de glisser des erreurs dans leur code. Une bonne vieille méthode, inventée en 1973 par ce cher Larry Tesler à qui l'on doit tant : le copier-coller. Une technique qui s'est révélée bougrement [3] efficace puisqu'elle permet d'extraire le fichier en quelques secondes. Une fois le contenu du *Follow TCP stream* de la connexion de données FTP copié-collé dans une fichier, il suffit en effet de décoder le base64 avec par exemple une petite ligne de Python.

On jette ensuite un rapide coup d'œil au fichier *sstic.tar.gz-chiffre* fraîchement récupéré pour vérifier qu'il est bien chiffré, ce que l'absence de chaînes de caractères notables et la répartition uniforme des octets semblent tous deux confirmer.

1.4 Dissection du canal caché

La clé AES utilisée pour chiffrer l'archive a été transférée par canal caché. La capture contient justement une série de `ping` consécutifs. On peut donc supposer sans trop se risquer qu'ils ne sont pas là par hasard et qu'ils renferment le précieux sésame. De plus, la consigne nous donne un indice :

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

La capture contient 65 paquets ICMP, mais le dernier est un peu à part puisque c'est le seul à avoir un TTL à 1. Cela nous amène donc à 64 paquets. L'indice nous apprend que les morceaux de clé semblent se cacher par groupe de 4 bits. Si l'on suppute que chaque paquet ICMP cache un groupe de 4 bits, on obtient alors une clé de 256 bits, ce qui est une taille standard [4].

Voici un petit aperçu de ces paquets :

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0xf132, seq=1/256, ttl=30
2	2.012000	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x0233, seq=1/256, ttl=30
3	4.025228	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x1333, seq=1/256, ttl=40
4	6.038273	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x2433, seq=1/256, ttl=30
5	8.051893	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x3533, seq=1/256, ttl=20
6	10.065300	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x4633, seq=1/256, ttl=10
7	11.078647	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x5533, seq=1/256, ttl=30
8	13.092487	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x6633, seq=1/256, ttl=30
9	14.105895	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x7533, seq=1/256, ttl=10
10	15.119234	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x8433, seq=1/256, ttl=20
11	17.132789	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0x9533, seq=1/256, ttl=20
12	18.146122	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0xa433, seq=1/256, ttl=40
13	20.159649	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0xb533, seq=1/256, ttl=10
14	22.173207	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0xc633, seq=1/256, ttl=10
15	24.186554	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0xd733, seq=1/256, ttl=20
16	26.200013	192.168.1.13	192.168.1.12	ICMP	98	Echo (ping) request id=0xe833, seq=1/256, ttl=10

FIGURE 1 – Premiers paquets ICMP

3. Oui, je viens bien d'utiliser le mot « bougrement »
4. http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

Visuellement, un œil averti ne manquera pas de remarquer certains détails :
- Le temps entre deux paquets varie toujours soit d'environ une seconde, soit d'environ deux secondes.
- L'identifiant ICMP possède une valeur étrange, qui change à chaque fois.
- Le TTL change également en prenant 4 valeurs différentes : 10, 20, 30 et 40.
- Le curseur de ma souris a fait une timide mais remarquée apparition en bas à droite de la capture d'écran.

Pour disséquer plus facilement les paquets afin de trouver ce qui les différencie, on peut utiliser Scapy. La méthode `show()` s'avère pratique pour afficher tous les champs d'un paquet. Voici la sortie de cette commande sur le premier paquet ICMP :

```
>>> pcap[0].show()
###[ Ethernet ]###
  dst       = 90:e6:ba:4e:ce:db
  src       = 00:01:02:08:fa:cb
  type      = 0x800
###[ IP ]###
  version   = 4L
  ihl       = 5L
  tos       = 0x2
  len       = 84
  id        = 0
  flags     = DF
  frag      = 0L
  ttl       = 30
  proto     = icmp
  chksum    = 0xd93d
  src       = 192.168.1.13
  dst       = 192.168.1.12
  \options   \
###[ ICMP ]###
  type      = echo-request
  code      = 0
  chksum    = 0x71ce
  id        = 0xf132
  seq       = 0x1
###[ Raw ]###
  load      = '\x06\xfcFQZ\xad\x02\x00\x08\t\n\x0b\x0c\r\x0e
  \x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d
  \x1e\x1f !"#$%&\'()*+,-./01234567'
```

Pour déterminer ce qui varie d'un paquet à l'autre, on écrit un script qui va comparer les sorties générées par `show()`. Il se trouve que cette méthode produit directement un affichage sur la sortie standard. Pour récupérer cette sortie il est donc nécessaire de s'encombrer de quelques lignes supplémentaires.

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-

from scapy.all import rdpcap
import cStringIO, sys, difflib

sys.stdout = out = cStringIO.StringIO()
pcap = rdpcap("icmp.pcap")
pcap[0].show(indent=0)
ref = out.getvalue().splitlines()
compared = []
for pkt in pcap[1:]:
    out.reset()
    pkt.show(indent=0)
    compared.append(out.getvalue().splitlines())

sys.stdout = sys.__stdout__
for i, s in enumerate(compared):
    print "\npaquet %i: " % (i+1)
    diff = difflib.unified_diff(ref, s, n=0)
    for d in diff:
        if d.startswith("+ "):
            print d
```

Voici le diff généré par ce script pour les 5 premiers paquets :

```
paquet 1:
+  chksum    = 0x7c9f
+  id        = 0x233
+  load      = '\x08\xfcFQ<\xdc\x02\x00\x08\t\n\x0b\x0c\r\x0e
\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d
\x1e\x1f !"#$%&\'()*+,-./01234567'

paquet 2:
+  tos       = 0x4
+  ttl       = 40
+  chksum    = 0xcf3b
+  chksum    = 0xbd6b
+  id        = 0x1333
+  load      = '\n\xfcFQ\xe8\x0f\x03\x00\x08\t\n\x0b\x0c\r\x0e
\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d
\x1e\x1f !"#$%&\'()*+,-./01234567'
```

```
paquet 3:
+  tos       = 0x4
+  chksum    = 0xd93b
+  chksum    = 0xb638
+  id        = 0x2433
+  load      = '\x0c\xfcFQ\xdcB\x03\x00\x08\t\n\x0b\x0c\r\x0e
\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d
\x1e\x1f !"#$%&\'()*+,-./01234567'

paquet 4:
+  ttl       = 20
+  chksum    = 0xe33d
+  chksum    = 0x6f03
+  id        = 0x3533
+  load      = '\x0e\xfcFQ\x10x\x03\x00\x08\t\n\x0b\x0c\r\x0e
\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d
\x1e\x1f !"#$%&\'()*+,-./01234567'

paquet 5:
+  tos       = 0x4
+  ttl       = 10
+  chksum    = 0xed3b
+  chksum    = 0xfcce
+  id        = 0x4633
+  load      = '\x10\xfcFQo\xac\x03\x00\x08\t\n\x0b\x0c\r\x0e
\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d
\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Nous avons donc détecté de nouveaux champs qui varient :

- Le TOS (Type of Service [5]) est un champ de la couche IP qui en théorie sert à déterminer la priorité d'un datagramme IP. En pratique, le TOS n'est que rarement pris en compte donc la modification de sa valeur n'a pas d'incidence. C'est donc un bon candidat pour y cacher de l'information.
- Le checksum de la couche IP ainsi que celui de la couche ICMP varient. Mais ils sont à chaque fois valides. Ils ne peuvent donc pas servir de canal caché puisque leur valeur n'est pas librement manipulable par un utilisateur.
- Enfin, la charge utile change à chaque paquet. Il y a d'ailleurs souvent plus de 4 bits qui sont modifiés. Est ce que c'est normal ? Ou est-ce que c'est un vecteur utilisé pour cacher de l'information ? Pour le savoir, il va falloir s'instruire un peu plus sur le protocole ICMP.

En se documentant [6], on apprend que la charge utile par défaut d'un paquet *ICMP Echo Request* est de 56 octets, ce qui concorde avec notre cas. Les 8 premiers octets sont un *timestamp* et le reste est une sorte de padding cyclique qui démarre à `\x08` et dont la valeur de chaque octet suivant est incrémentée de 1.

5. http://en.wikipedia.org/wiki/Type_of_service
6. http://sd.wareonearth.com/~phil/net/ping/

Le padding est constant d'un paquet à l'autre. Quant au timestamp, il s'avère coïncider presque parfaitement avec le temps d'émission des paquets (stocké directement dans le format pcap). Presque. L'écart de quelques microsecondes est probablement du au temps nécessaire entre le calcul du payload ICMP et l'émission effective du paquet sur la carte réseau. La charge utile ne l'est donc pas pour nous !

En analysant plus en détail les valeurs prises par l'identifiant ICMP, on remarque que d'un paquet à l'autre, il est incrémenté soit de 17, soit de 15. Une bit peut donc être codé en utilisant les variations de valeur de l'identifiant. Cependant, en regardant de plus près, on s'aperçoit que chaque écart de 17 de l'identifiant coïncide avec un écart de 2 secondes entre les paquets (et réciproquement pour un écart de 15 et 1 seconde). Les informations apportées par la variation d'identifiant et la variation de temps sont donc exactement identiques.

Pour résumer, les 4 bits de clé cachés dans chaque paquet ICMP vont se trouver :
- Dans le champ TOS qui a ici deux valeurs possibles (2 et 4) ce qui permet de coder un bit.
- Dans le champ TTL qui a quatre valeurs possibles (10, 20, 30 et 40) ce qui permet de coder deux bits.
- Dans la variation de temps entre deux paquets et/ou la variation entre deux identifiants, qui permet de coder un bit.

1.5 Solution

La localisation des 256 bits de clé a été identifiée. Il reste cependant plusieurs inconnues à trouver pour déchiffrer notre fichier : l'ordre dans lequel ces 4 bits sont concaténés ensemble, quelle valeur correspond à quel bit et quel mode AES est utilisé. Pour résoudre ce problème, il n'y a d'autre solution que de tester toutes les possibilités. Voici le script Python utilisé pour extraire toutes les clés et déchiffrer le fichier :

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-

from scapy.all import rdpcap, IP
import Crypto, subprocess, itertools, sys

def extract_info(pkt):
    """ Retrieve time, tos and ttl of a packet"""
    return pkt.time, pkt[IP].tos, pkt[IP].ttl

def time_to_bit(delta_time, case=0):
    """ 2 possible cases """
    assert case in [0,1], "Time: invalid case " + str(case)
    assert delta_time in [1,2], "Time: invalid value " + str(delta_time)
    choices = list(itertools.permutations(["0","1"]))
    c = choices[case]
```

```
17      if delta_time == 1:
18          return c[0]
19      return c[1]
20
21  def tos_to_bit(tos, case=0):
22      """ 2 possible cases """
23      assert case in [0,1], "Tos: invalid case " + str(case)
24      assert tos in [2,4], "Tos: invalid value " + str(tos)
25      choices = list(itertools.permutations(["0","1"]))
26      c = choices[case]
27      if tos == 2:
28          return c[0]
29      return c[1]
30
31  def ttl_to_bit(ttl, case=0):
32      """ 24 possible cases """
33      assert case in range(24), "TTL: invalid case " + str(case)
34      assert ttl in [10, 20, 30, 40], "TTL: invalid value " + str(ttl)
35      choices = list(itertools.permutations(["00", "01", "10", "11"]))
36      c = choices[case]
37      if ttl == 10:
38          return c[0]
39      elif ttl == 20:
40          return c[1]
41      elif ttl == 30:
42          return c[2]
43      return c[3]
44
45  def swap_order(time, tos, ttl, case=0):
46      """ 6 possible cases """
47      assert case in range(6), "Concat: invalid case " + str(case)
48      choices = list(itertools.permutations([time, tos, ttl]))
49      return "".join(choices[case])
50
51  def get_key(pcap, time_case, tos_case, ttl_case, concat_order):
52      result = ""
53      old_time = None
54      for pkt in pcap:
55          time, tos, ttl = extract_info(pkt)
56          if old_time:
57              delta_time = int(round(time - old_time))
58              time_key = time_to_bit(delta_time, time_case)
59              tos_key = tos_to_bit(old_tos, tos_case)
60              ttl_key = ttl_to_bit(old_ttl, ttl_case)
61              result += swap_order(time_key, tos_key, ttl_key, concat_order)
62          old_time, old_tos, old_ttl = time, tos, ttl
63      key = ""
64      for i in range(0,256,8):
```

```
65          key += chr(int(result[i:i+8],2))
66      return key
67
68  def decrypt(key, cipher_mode):
69      iv = "76C128D46A6C4B15B43016904BE176AC".decode("hex")
70      mode = cipher_mode
71      aes = Crypto.Cipher.AES.new(key, mode, iv)
72      f = open("sstic.tar.gz-chiffre").read()
73      return aes.decrypt(f)
74
75  def check_result(r):
76      with open("/tmp/aes_result","wb") as f:
77          f.write(r)
78
79      h = Crypto.Hash.MD5.new(r).hexdigest()
80      if h == "61c9392f617290642f9a12499de6b688":
81          print "Found md5 hash !"
82          sys.exit(0)
83      try:
84          out = subprocess.check_output(["file", "/tmp/aes_result"])
85          if not out.endswith(": data\n"):
86              print out.strip()
87          if any(x in out for x in ["tar", "zip", "rar"]):
88              print "Found an archive !"
89              sys.exit(0)
90      except subprocess.CalledProcessError:
91          pass
92
93  def main():
94      pcap = rdpcap("icmp.pcap")
95      for time_case in range(2):
96          for tos_case in range(2):
97              for ttl_case in range(24):
98                  for concat_order in range(6):
99                      key = get_key(pcap, time_case, tos_case, ttl_case,
100                                    concat_order)
101                     for cipher_mode in range(1,6):
102                         r = decrypt(key, cipher_mode)
103                         check_result(r)
104
105 main()
```

Pour chaque clé et chaque mode AES, afin de tester si le déchiffrement a fonctionné, on vérifie que le hash MD5 du fichier déchiffré correspond bien à la valeur fournie par la consigne (61c9392f617290642f9a12499de6b688). Il se trouve que ça n'est jamais le cas !

On rajoute donc une deuxième condition d'arrêt qui dépend de l'affichage retourné par la commande `file` sur chaque fichier déchiffré. On trouve ainsi quelques fichiers prometteurs (comme par exemple des configurations Sendmail) mais qui se révèlent tous être des faux-positifs. Jusqu'à ce qu'enfin l'un des fichiers soit reconnu comme étant une archive.

```
$ python lvl1.py
[...]
/tmp/aes_result: gzip compressed data, was "archive.tar",
from Unix, last modified: Mon Mar 18 12:24:37 2013
Found an archive !
```

La date de l'archive est cohérente, il est donc fort probable que la bonne clé ait été trouvée et que le déchiffrement ait fonctionné. Reste à résoudre le mystère du hash MD5 incorrect. Quand on extrait notre archive gzip en ligne de commande, cela fonctionne mais nous apprend que des octets ont été ignorés :

```
$ gzip -d aes_result.gz

gzip: aes_result.gz: decompression OK, trailing garbage ignored
```

Tandis que quand on utilise le gestionnaire d'archive graphique par défaut de Gnome (File Roller), la décompression ne fonctionne pas et l'archive Tar située dans le conteneur gzip apparaît comme pesant 101,1Mo. Cette valeur est visiblement erronée puisque l'archive gzip ne pèse que 620ko. Tout ceci n'est pas sans nous rappeler la première étape du challenge SSTIC 2011 qui consistait à reconstituer une archive gzip coupée en plusieurs morceaux. Ce fut l'occasion de découvrir que dans le format gzip, la taille du fichier après décompression est stockée dans les quatre derniers octets du fichier.

Si File Roller affiche une taille incohérente et se prend les pieds dans le tapis au moment de la décompression, c'est donc que les derniers octets du fichier sont en cause.

```
$ hexdump aes_result.gz | tail -n 2
6d6d 5b6b dadb b6d6 b5b6 6dad 6b6d 4cfb
6ffb 0809 9bbd 2800 0032 0606 0606 0606
```

On comprend que lors du déchiffrement AES, un padding[7] PKCS#7 constitué de six octets `\x06` a été rajouté à la fin du fichier. On retrouve ainsi notre taille incohérente de 101Mo (0x06060606). Une fois ce padding supprimé manuellement, le nouveau hash MD5 devient identique à celui de la consigne et l'archive se décompresse sans soucis avec File Roller. L'heure du niveau 2 a sonné.

7. http://en.wikipedia.org/wiki/Padding_%28cryptography%29#PKCS7

2 Niveau 2 : Xilinx, FPGA et machine virtuelle

2.1 Analyse préliminaire

La décompression de l'archive nous offre quatre fichiers :
- data
- smp.py
- decrypt.py
- s.ngr

Le fichier *data* contient des données brutes qui semblent être chiffrées. Le fichier *smp.py* est un script python qui ne contient qu'un seul objet : une liste de 231 entiers compris entre 0 et 224.

Le fichier decrypt.py est nettement plus intéressant. Le voici :

```
#!/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()
```

```
33 result_md5 = [ord(x) for x in result_md5]
34 target_md5 = "6c0708b3cf6e32cbae4236bdea062979"
35 target_md5 = [int(target_md5[x:(x + 2)], 16) for x in range(0,
36                len(target_md5), 2)]
37 print(["%02x" % x for x in target_md5])
38 print(["%02x" % x for x in result_md5])
39 if result_md5 != target_md5:
40   print("Bad key...")
41   sys.exit(1)
42
43 result = base64.b64decode("".join(result))
44 d = open("atad", "wb")
45 d.write(result)
46 d.close()
47 sys.exit(0)
```

Ce script utilise le module python `dev`. Un module qui n'existe pas et dont on devine facilement que notre mission, si on l'accepte, va être de le créer. Des fonctions de ce module sont ensuite appelées avec comme paramètres le fichier *s.ngr*, la liste du fichier *smp.py*, le contenu du fichier *data* et une clé secrète. Ce module va ensuite générer un résultat dont la hash MD5 sera testé pour vérifier la validité de la clé. Pour finir, le résultat va être stocké dans un fichier nommé *atad*.

On remarque deux détails qui auront leur importance pour la suite :

- Il y a une typo dans l'instruction `dev.init("sp.ngr")` à la ligne 19 puisque le fichier *sp.ngr* n'existe pas. En effet, le fichier de l'archive s'appelle *s.ngr*. Cela laisse à penser que ce fichier n'est pas réellement utilisé et parsé pour initialiser le module `dev`. On nous informe simplement qu'il faudra en tenir compte... d'une manière pour l'instant obscure mais qui va s'éclaircir au fur et à mesure que mes neurones vont se mutiner.
- Le résultat est base64-décodé avant d'être stocké dans un fichier. On sait donc que tout ce qui sort du module doit être encodé en base64.

Pour finir cette analyse préliminaire, le fichier *s.ngr* est quant à lui bien énigmatique. Il ne s'agit pas d'un format de fichier standard et récupérer des informations pertinentes sur celui-ci est une tâche ardue.

```
$ file s.ngr
s.ngr: data

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

Les deux premières lignes du fichier contiennent un en-tête faisant référence à Xilinx [8] qui se révèle être la société qui a inventé puis qui s'est spécialisée dans les FPGA [9]. Un acronyme que j'avais déjà eu l'occasion d'entendre mais sans vraiment savoir à quoi il correspondait. Pour résumer en une phrase ce que j'en ai compris, un FPGA est un composant hardware programmable.

Comme l'indique l'en-tête, le reste du fichier est de l'ASCII (enfin presque...). Il y a en réalité plus de 3Mo de données, placées sur une seule ligne, sans retour chariot. Ouvrir le fichier avec `gedit` est une excellente idée si l'on a quelques vieux griefs contre cet éditeur de texte et qu'on souhaite se venger en le faisant souffrir.

Des recherches plus poussées sur le format de fichier NGR, agrémentées de mots-clés tels que *Xilinx* ou *FPGA* nous permettent de mettre la main sur un PDF [10] qui nous en apprend plus.

```
RTL View is a Register Transfer Level graphical representation
of your design. This representation (.ngr file produced by Xilinx
Synthesis Technology (XST)) is generated by the synthesis tool at
earlier stages of a synthesis process when technology mapping is
not yet completed. The goal of this view is to be as close as
possible to the original HDL code. In the RTL view, the design is
represented in terms of macro blocks, such as adders, multipliers,
and registers. Standard combinatorial logic is mapped onto logic
gates, such as AND, NAND, and OR
```

C'est un format propriétaire qui sert à représenter graphiquement la façon dont est programmé un FPGA. Il n'y a aucune documentation pour parser ou faire de la rétroconception sur ce format. De plus, il n'existe à priori pas de logiciels permettant de convertir un fichier NGR en code (comme du VHDL par exemple). Plusieurs forums font référence à un version gratuite de l'outil ISE, distribué par Xilinx, qui permet d'ouvrir les fichiers NGR. Nous partons donc en quête de ce logiciel.

2.2 Récupération du logiciel ISE

Ouvrir le fichier *s.ngr* est déjà un vrai challenge en soi. On se retrouve sur le site de Xilinx [11] à télécharger une suite logicielle pharaonique qui est découpée en quatre archives pour un total de plus de 8Go. Le téléchargement et l'installation prendront au final presque une journée entière.

Une fois l'installation achevée, on cherche le logiciel ISE parmi les 20Go de l'arborescence luxuriante qui a poussé sur le disque dur. On parvient finalement à lancer quelque chose qui nous permet d'ouvrir le fichier *s.ngr* et de naviguer de manière interactive à travers le schéma.

8. http://en.wikipedia.org/wiki/Xilinx
9. http://en.wikipedia.org/wiki/Field_programmable_gate_array
10. http://www.xilinx.com/support/documentation/user_guides/ug685.pdf
11. http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/v2012_4---14_4.html

2.3 Compréhension itérative du schéma

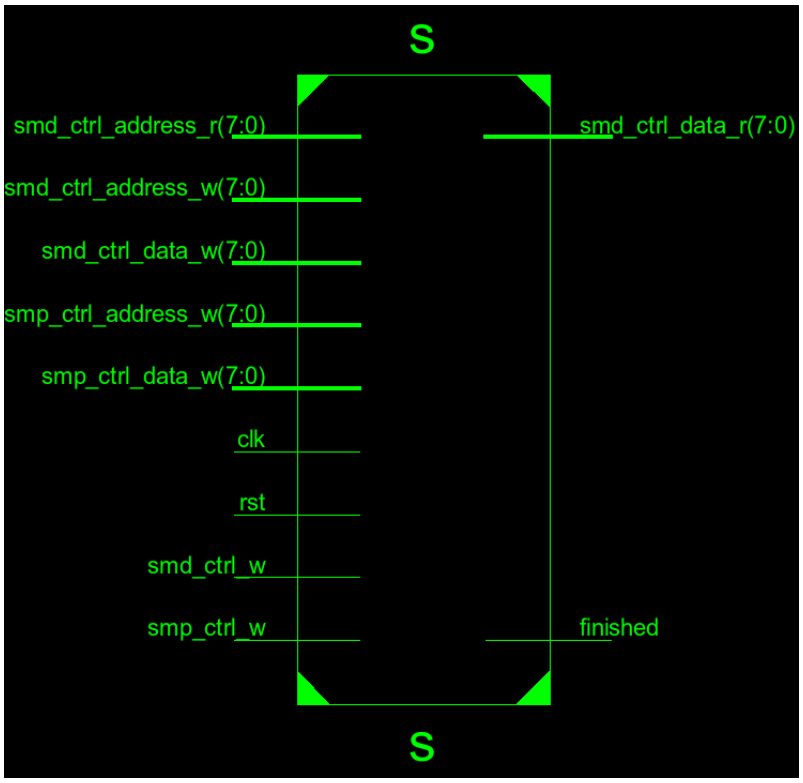
Je suis alors submergé par une succession d'émotions : la peur de commencer à comprendre ce qu'il va falloir faire pour résoudre cette épreuve. Puis le choc lorsque je réalise que c'est encore pire que ce que je pensais. Et enfin le désespoir lorsque je découvre que chaque élément du schéma contient lui-même un enchevêtrement complexe de circuits et de composants électroniques.

Après un premier niveau abordable, cette seconde épreuve est une véritable mise en exergue du sadisme des concepteurs. Un hymne à la cruauté. Un autel dressé au culte de la barbarie [12]. Mes nerfs commencent à vriller et je cherche une poule sur laquelle foncer en piqué diagonal. Puis je me reprends en main et après une bonne nuit de sommeil, je résume à tête reposée le travail à effectuer :

1. Implémenter un système qui simule le fonctionnement du FPGA décrit par le schéma, sans possibilités de tester la conformité de cette implémentation.
2. Deviner comment sont agencées les différentes données passées en entrée du module (SMP, data et la clé).
3. Cerise sur le gâteau, une fois que tout cela fonctionnera il faudra encore débusquer une faiblesse dans le mécanisme afin de trouver la clé secrète.

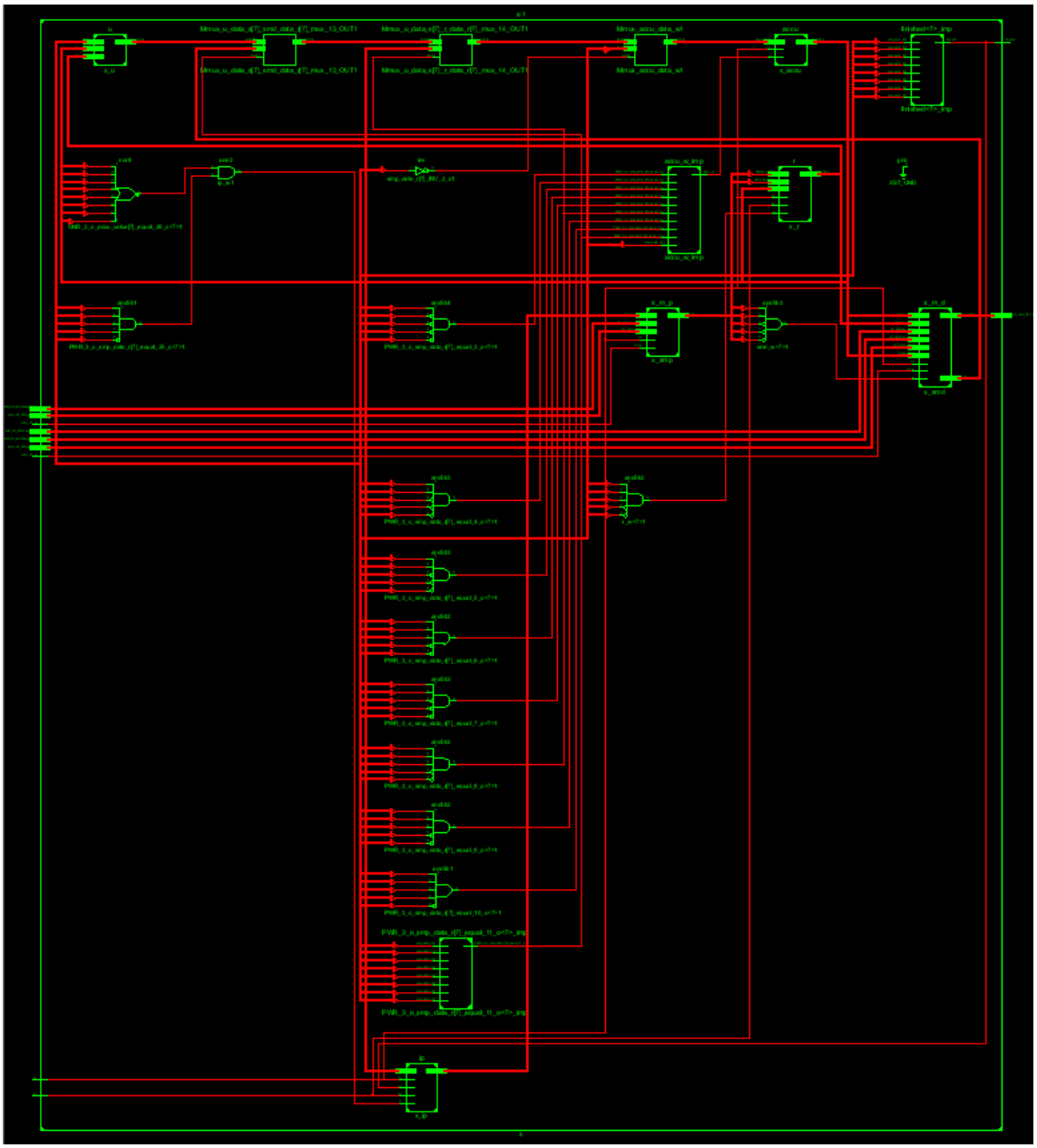
Facile !

Pour donner un petit aperçu, voici la vue qui représente uniquement les signaux et les bus d'entrée et de sortie du circuit :



12. http://fr.wikiquote.org/wiki/Kaamelott/Guethenoc

Voici ensuite la vue de ce circuit en n'affichant que les éléments haut-niveau. Sachant que chaque élément est ensuite composé d'un ensemble de signaux, de portes et de composants [13] :



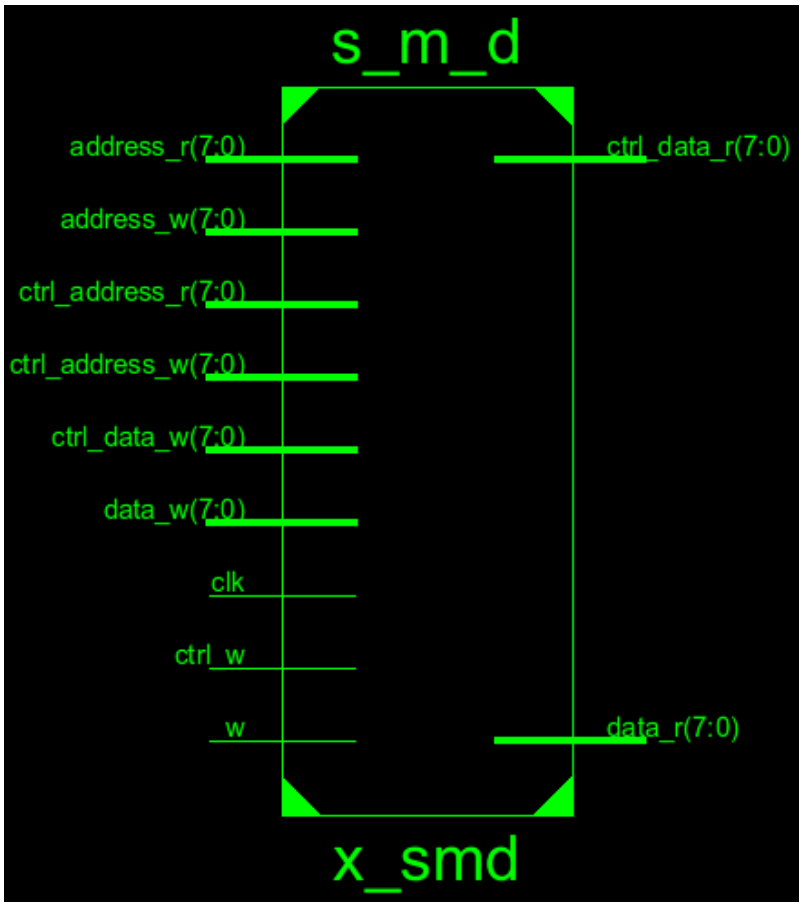
Après avoir longuement exploré les recoins du schéma d'un air idiot sans comprendre ce que j'avais sous les yeux, je me suis dit que la meilleure façon d'avancer (outre chercher de la documentation sur les composants électroniques représentés sur le schéma) était d'essayer de coder le fonctionnement décrit par le graphique. J'ai ainsi commencé une première fois avec une implémentation très bas niveau dans laquelle j'ai tenté de simuler chaque composant et chaque signal électrique.

13. DISCLAIMER : Le but de cette figure et de celles qui vont suivre n'est pas d'être lisibles mais de donner un aperçu au lecteur de ce à quoi ressemblent les schémas

Évidemment, ça n'a pas fonctionné. Mais ça m'a permis de mieux appréhender certaines notions, de comprendre à quoi servaient certains composants mis ensemble. Fort de cette nouvelle connaissance, j'ai refait une implémentation plus haut niveau. Elle non plus n'a pas fonctionné. D'autant que je ne savais toujours pas comment étaient censées être organisées les données en entrée du système. Mais en recodant chaque bloc, j'ai affiné ma compréhension. Après plusieurs itérations de cette méthodologie fastidieuse, j'ai fini par comprendre ce que faisait chaque bloc et ce que décrivait concrètement le schéma.

Il s'agit d'une sorte de machine virtuelle. Les entiers de la liste SMP sont des instructions qui vont être appliquées sur les données en entrée pour générer la sortie. Dans les sections suivantes, je vais m'attarder à décrire ce que fait chaque bloc de haut niveau.

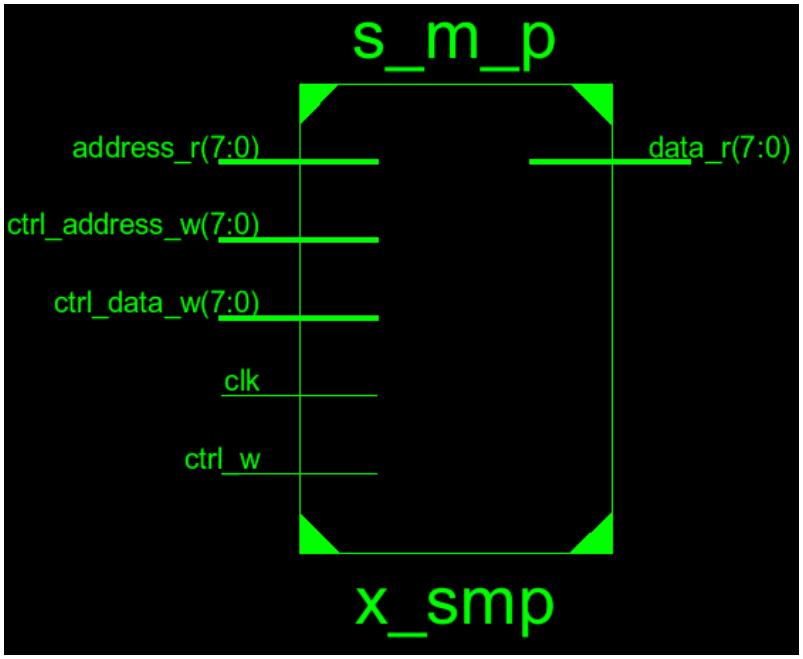
2.4 Bloc SMD



Tenter d'afficher l'ensemble des éléments contenus dans ce bloc provoque un déni de service. Il faut donc procéder morceau par morceau. Ce bloc renferme deux gros multiplexeurs de 256 entrées où chacune de ces entrées est raccordée à un circuit flip-flop. Il y a beaucoup de signaux électriques nécessaires pour le fonctionnement bas niveau, mais dont on peut faire abstraction une fois qu'on comprend le but du bloc. Un multiplexeur couplé à des circuits flip-flops sert à stocker un tableau de

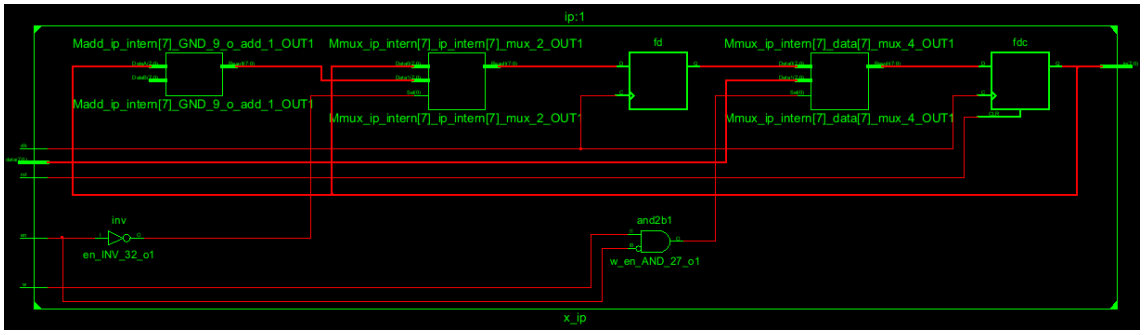
données. Au final, le bloc SMD se contente de stocker d'une part les données en entrée du système ($data + key$) et d'autre part les données déchiffrées qui seront retournées en sortie.

2.5 Bloc SMP



D'une manière similaire à SMD, le bloc SMP est gros ce qui le rend difficile à afficher et à analyser. Mais il sert tout simplement au stockage de la liste des 231 instructions du fichier *smp.py*.

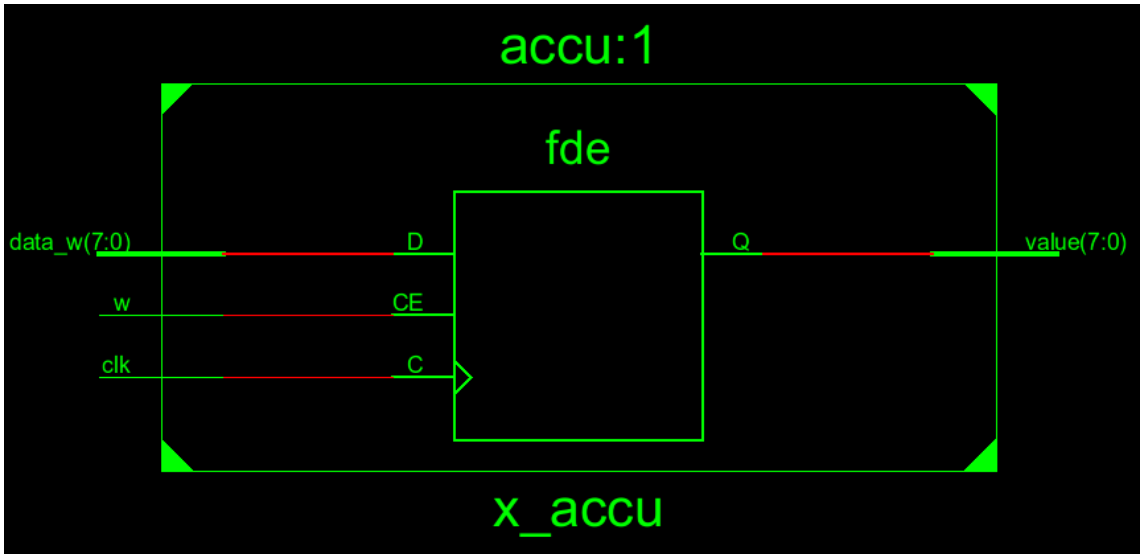
2.6 Bloc IP



De prime abord on pouvait supposer que le nom du bloc n'avait pas été choisi par hasard, mais difficile de décider s'il représentait plutôt *Internet Protocol* ou *Instruction Pointer*. En pratique, ce bloc sert bien à manipuler un pointeur d'instruction, c'est à dire l'index de la liste SMP où sera située la prochaine instruction. Dans le cas général, le pointeur est incrémenté de 1 à chaque itération. Mais il existe un cas particulier, quand l'instruction de SMP est comprise entre 184 et 191 et que la

valeur stockée dans le bloc Accu est 0. En effet, quand ces conditions sont respectées, le pointeur d'instruction prend à la place la valeur qui est en sortie du bloc R. En clair, il s'agit tout simplement d'un `Jump`.

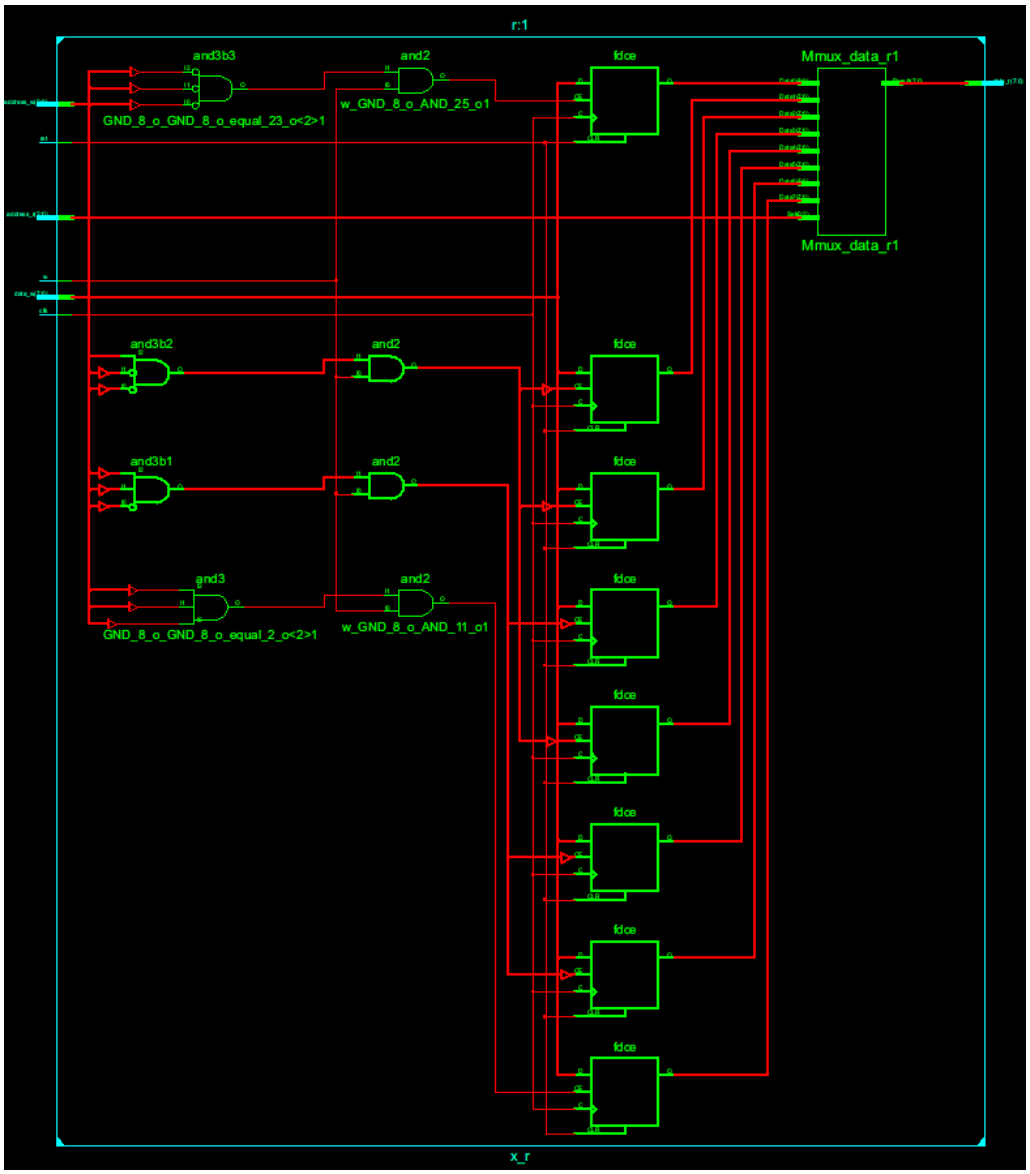
2.7 Bloc Accu



Ce bloc contient uniquement un circuit flip-flop qui sert à stocker une valeur. Sa particularité est que cette valeur peut être attribuée par à peu près tous les autres blocs. Elle est utilisée partout et change souvent. Pour faire une analogie avec le fonctionnement classique d'un CPU x86, pour moi, Accu correspondrait au registre EAX. C'est le registre principal de notre machine virtuel.

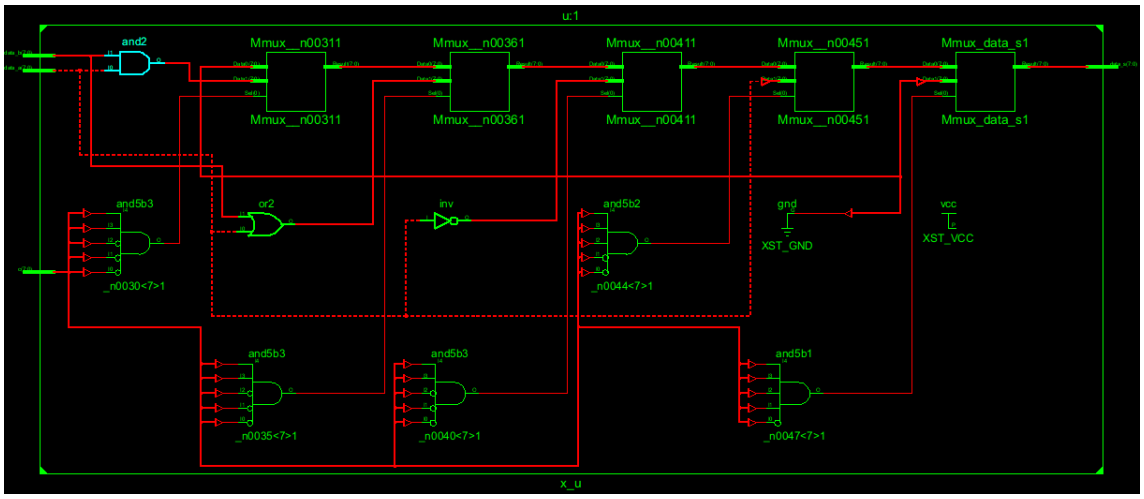
2.8 Bloc R

Ce bloc permet de stocker huit valeurs différentes d'Accu. En pratique, cela nous permet de disposer de huit registres supplémentaires afin de pouvoir stocker et manipuler des valeurs d'une manière plus durable.

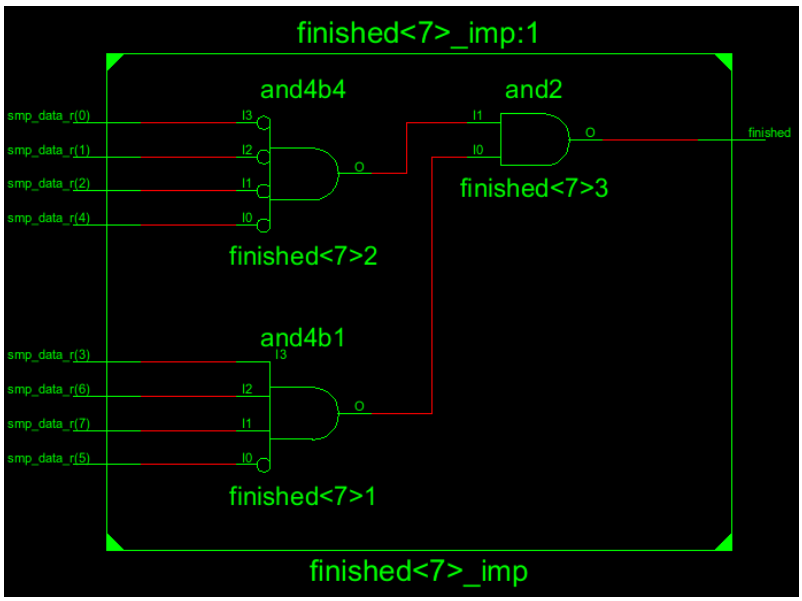


2.9 Bloc U

Le bloc U est le seul qui fasse réellement quelque chose. Selon la valeur de l'instruction SMP, il va appliquer des opérations logiques (OR, AND, NOT, XOR) entre les morceaux de données.



2.10 Bloc Finished



Ce bloc permet de stopper notre machine virtuelle quand la valeur de l'instruction SMP est 200.

2.11 Implémentation haut niveau

Ayant mieux compris le rôle de chaque bloc, il est maintenant possible de réaliser une implémentation très haut niveau du système. Implémentation qui, au final, est étonnement concise. Voici le code Python de ma classe Fpga.

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-

class Fpga:
    def __init__(self):
        self.eax = None
        self.r = list(None for x in range(8))
        self.finished = False

    def load_smp(self, smp):
        self.smp = smp

    def load_smd(self, smd):
        self.smd = smd

    def get_data(self):
        return list(chr(c) for c in self.smd[17:])

    def run(self, pause=True):
        self.finished = False
        self.ip = 0
        while not self.finished:
            ip = self.ip
            i = self.smp[self.ip]
            self.process_instruction(i)
            if ip == self.ip:
                self.ip += 1

    def jump(self, addr):
        self.ip = addr

    def ret(self):
        self.finished = True

    def process_instruction(self, i):
        idx = i & 7
        if i < 128:
            self.eax = i
        elif i >= 136 and i <= 143:
            self.eax = self.eax & self.r[idx]
        elif i >= 144 and i<= 151:
            self.eax = self.eax | self.r[idx]
        elif i >= 160 and i <= 167:
            self.eax = self.eax ^ 0xff
        elif i >= 168 and i <= 175:
             self.eax = self.r[idx]
```

```
47          elif i >= 176 and i <= 183:
48              self.r[idx] = self.eax
49          elif i >= 184 and i <= 191:
50               if self.eax == 0:
51                  self.jump(self.r[idx])
52          elif i >= 192 and i <= 199:
53              self.smd[self.r[idx]] = self.eax
54          elif i == 200:
55              self.ret()
56          elif i == 208:
57              self.eax = self.smd[self.eax]
58          elif i == 216:
59              self.eax = self.eax << 1 & 0xff
60          elif i == 224:
61              self.eax = self.eax | 128
62          else:
63              raise Exception("Unknown instruction: %i" % i)
```

Pour que ce code soit utilisable directement avec le fichier *decrypt.py*, on créé un module `dev` minimaliste qui fait la jonction :

```
1  #!/usr/bin/env python
2  #-*- coding:utf-8 -*-
3  import fpga
4
5  s = fpga.Fpga()
6
7  def init(ngr):
8      pass
9
10 def send_smd(smd):
11     k, size, sm = smd
12     smd = k + [size] + list(ord(c) for c in sm)
13     s.load_smd(smd)
14
15 def send_smp(smp):
16     s.load_smp(smp)
17
18 def start():
19     s.run()
20
21 def wait_finished():
22     pass
23
24 def get_data():
25     return s.get_data()
```

2.12 Débogage façon assembleur

L'implémentation présentée dans le paragraphe précédent est fonctionnelle, mais ce ne fut pas le cas du premier coup. Tout d'abord parce que je n'avais toujours pas de certitude quant à l'ordonnancement des données passées en entrée. Mais également car quelques petites subtilités du schéma m'avaient échappé.

Pour déboguer, le besoin d'avoir une syntaxe facilement compréhensible par un humain s'est rapidement fait ressentir. J'ai donc pris la peine de traduire chaque instruction de SMP dans une syntaxe Python proche de celle de l'assembleur. Voici le dictionnaire de correspondance :

```
ASM = {
    0: "eax = 0",
    1: "eax = 1",
    2: "eax = 2",
    14: "eax = 14",
    15: "eax = 15",
    16: "eax = 16",
    17: "eax = 17",
    22: "eax = 22",
    34: "eax = 34",
    36: "eax = 36",
    44: "eax = 44",
    62: "eax = 62",
    64: "eax = 64",
    72: "eax = 72",
    76: "eax = 76",
    82: "eax = 82",
    83: "eax = 83",
    87: "eax = 87",
    89: "eax = 89",
    99: "eax = 99",
    102: "eax = 102",
    109: "eax = 109",
    113: "eax = 113",
    118: "eax = 118",
    136: "eax = eax & r0",
    138: "eax = eax & r2",
    142: "eax = eax & r6",
    143: "eax = eax & r7",
    146: "eax = eax | r2",
    147: "eax = eax | r3",
    148: "eax = eax | r4",
    150: "eax = eax | r6",
```

```
34      160: "eax = eax ^ 0xff",
35      168: "eax = r0",
36      169: "eax = r1",
37      170: "eax = r2",
38      171: "eax = r3",
39      172: "eax = r4",
40      173: "eax = r5",
41      174: "eax = r6",
42      175: "eax = r7",
43      176: "r0 = eax",
44      177: "r1 = eax",
45      178: "r2 = eax",
46      179: "r3 = eax",
47      180: "r4 = eax",
48      181: "r5 = eax",
49      182: "r6 = eax",
50      183: "r7 = eax",
51      187: "if eax == 0 : jmp(r3)",
52      188: "if eax == 0 : jmp(r4)",
53      189: "if eax == 0 : jmp(r5)",
54      190: "if eax == 0 : jmp(r6)",
55      193: "smd[r1] = eax",
56      200: "ret()",
57      208: "eax = smd[eax]",
58      216: "eax = eax << 1 & 0xff",
59      224: "eax = eax | 128",
60  }
```

Grâce à ce dictionnaire, il a été possible d'exécuter chaque instruction pas à pas, en affichant à chaque fois l'état des différents registres, comme lors d'un débogage standard. Cela a permis de trouver les quelques petites erreurs d'implémentation restantes en remarquant l'appel à des instructions inutiles (car fausses) lors de l'exécution. De plus, en observant dans quel ordre sont lus les octets de SMD, il est possible d'en déduire comment sont organisées les données.

Ainsi, on comprend que les données stockées dans SMD sont rangées telles que l'annonce le fichier *decrypt.py* : tout d'abord les 16 octets de clé, ensuite un octet représentant la longueur des données, puis un morceau issu du fichier *data*.

2.13 Cassage de la clé

La simulation est fonctionnelle. Reste à trouver la clé ! Pour ce faire, on teste le déchiffrement du premier bloc de données avec différentes valeurs de clé :

- Uniquement des 00
- Uniquement des 01
- Le premier octet à 01 et les 15 autres à 00
- Le premier octet à 02 et les 15 autres à 00
- ...

On s'aperçoit immédiatement que quelque chose cloche :

- Le premier octet de la clé sert au déchiffrement des octets de *data* 1, 17, 33, ...
- De la même manière, le deuxième octet de la clé n'est utilisé que pour les octets de *data* 2, 18, 34, ...

Il est donc possible de bruteforcer chaque octet de clé indépendamment des autres. La condition pour savoir si l'octet testé est correct est simple : il faut que tous les octets déchiffrés soient du base64 valide.

Pour casser la clé, on va donc tester seulement 256 clés différentes (que des 00, que des 01, que des 02, etc) puis on analysera dans quels cas les données répondent aux critères définis ci-dessus. Comme le déchiffrement de l'ensemble du fichier *data* est long, on se limite ici aux 10 premiers blocs de données.

```
#!/usr/bin/python

import sys, base64, md5, re, dev, smp

secret_key = list(None for i in range(16))
b64_range = re.compile(r"[A-Za-z0-9/\+\n=]")
d = open("data").read()

# Testing 256 keys
for k in range(256):
    key = list(k for i in range(16))
    result = []

    # Start our Fpga with 10 blocks of data
    for i in range(0, 224*10, 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()

    # Analyzing results
    for i in range(16):
        r = result[i::16]
        if all(b64_regex.match(chr(x)) for x in r):
            print "Found part %i of the key ! ==> %s" % (i, hex(k))
            secret_key[i] = k
            print secret_key

print "Key cracked: " + repr(secret_key)
```

```
secret_key.reverse()
secret_key = "".join(chr(x).encode("hex") for x in secret_key)
print "Reversed order: " + secret_key
```

```
$ python crack.py
Key cracked: [230, 131, 220, 188, 22, 239, 88, 198, 101, 172, 35,
211, 30, 109, 161, 37]
Reversed order: 25a16d1ed323ac65c658ef16bcdc83e6
```

La clé trouvée, on peut maintenant déchiffrer l'ensemble du fichier *data* ce qui génère un fichier *atad* dans le répertoire courant. Les portes du niveau 3 sont grandes ouvertes.

3 Niveau 3 : PostScript

3.1 Analyse préliminaire

La fichier *atad* pèse 32Ko et contient du texte.

```
$ file atad
atad: ASCII text, with very long lines
```

Une chaîne de caractères présente à la fin du fichier et qui s'apparente à un message d'erreur nous met la puce à l'oreille sur ce qu'est *atad*.

```
{ (usage: gs -- script.ps key\n) error flush }
```

`gs`, alias `Ghostscript` est un « *PostScript and PDF language interpreter and previewer* ». De plus, l'extension du nom *script.ps* nous indique que le fichier *atad* est un script écrit en langage PostScript. Initialement, tout le contenu du fichier était sur une seule et unique ligne. Mais en grand seigneur et rien que pour toi, cher lecteur, en voici une version indentée :

```
/I1 currentfile 0 (cafebabe) /SubFileDecode filter
/ASCIIHexDecode filter /ReusableStreamDecode filter
cf760bc77db1f282e881ede9a10122b220887466b973b854218[...] cafebabe def

/I2 currentfile 0 (cafebabe) /SubFileDecode filter
/ASCIIHexDecode filter /ReusableStreamDecode filter
c598d7cc5b5354440b0613490c45483d4e7b0f6c0368120f100[...] cafebabe def

```

```
/I3 currentfile 0 (cafebabe) /SubFileDecode filter
/ASCIIHexDecode filter /ReusableStreamDecode filter
338f25667eb4ec47763dab51c3fa41cba329e18536b83159b3a[...] cafebabe def

/I4 currentfile 0 (cafebabe) /SubFileDecode filter
/ASCIIHexDecode filter /ReusableStreamDecode filter
8ae98ae900000000020002000316121411455556000085554001[...] cafebabe def

/error { (%stderr)(w) file exch writestring } bind def

errordict /handleerror { quit } put

/main
{ mark shellarguments
    { counttomark 1 eq
        { dup length exch /ReusableStreamDecode filter exch 2 idiv
            string readhexstring pop dup length 16 eq
            { I1 32 exch mark 1 index resetfile 1 index
                { counttomark 1 sub index counttomark 2 add index
                    4 mul string readstring pop dup () eq {pop exit} if
                } loop
                counttomark -1 roll counttomark 1 add 1 roll ] 4 1 roll
                pop pop pop

                I2 0 index resetfile 61440 string readstring pop dup 3
                index 2 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

                I3 resetfile
                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

```

```
57                  exch 1 sub
58                  { 3 copy exch length getinterval 2 index mark 3 1 roll
59                      0 1 3 -1 roll dup length 1 sub exch 4 1 roll
60                      { dup 3 2 roll dup 5 1 roll exch get 3 1 roll exch
61                          dup 5 1 roll exch get xor 3 1 roll
62                      } for
63                       pop pop ] dup length string 0 3 -1 roll
64                      { 3 -1 roll dup 4 1 roll exch 2 index exch put 1 add
65                      } forall
66                      pop 4 -1 roll dup 5 1 roll 3 1 roll dup 4 1 roll
67                      putinterval exch pop
68                  } for 0 1 1 {pop pop} for cvx exec
69              } if
70              false
71          }
72          { (no key provided\n) error true } ifelse
73      }
74      { (missing '--' preceding script file\n) error true } ifelse
75      { (usage: gs -- script.ps key\n) error flush } if
76  }
77  bind def
78  main
79  clear
80  quit
```

Même indenté, ce script reste compliqué à lire tant qu'on ne dispose pas d'un minimum de connaissances sur le langage PostScript. À noter que quatre chaînes au début du fichier (lignes 3, 7, 11 et 15) ont ici été tronquées pour des raisons d'affichage alors que dans les faits, elles sont extrêmement longues.

3.2 Le PostScript pour les nuls

La montée en compétence sur le langage PostScript fut relativement aisée. Ce qu'il faut en retenir, c'est que c'est un langage principalement destiné à décrire des pages dans le but de les imprimer, mais qu'il est malgré tout *Turing-Complete*[14]. Il utilise une notation polonaise inverse[15], c'est à dire que l'opérateur est situé à la fin. Par exemple, une addition `1+2` s'écrira `1 2 add` et une opération `3*(1+2)` se notera `3 1 2 add mul`.

Le langage utilise une pile et peut être interprété en flux, à la volée. On peut simplifier à l'extrême le principe de fonctionnement en disant que quand un entier ou une chaîne est rencontré, il est placé sur le dessus de la pile. Quand un opérateur est rencontré (une addition, un `if`, une boucle `for`, ...), il dépile le nombre d'arguments dont il a besoin puis il place le résultat final sur le haut de la pile. Par conséquent, un script PostScript est facile à interpréter par une machine alors que pour un humain, la notation polonaise inverse est plutôt contre-intuitive.

14. http://en.wikipedia.org/wiki/Turing_completeness
15. https://fr.wikipedia.org/wiki/Notation_polonaise_inverse

3.3 Méthodologie de rétroconception

Pour comprendre ce que fait ce script, on s'arme d'une documentation exhaustive [16] du langage PostScript ainsi que d'un arsenal impressionnant d'outils de débogage :

- L'outil `gs` dans sa version interpréteur de commande.
- L'édition à la main du code source pour afficher des informations utiles.

Ça peut paraître rudimentaire. Ça l'est. Mais ce fut suffisant. L'interpréteur de commandes permet d'exécuter des sous-parties du script de façon *standalone* afin de les comprendre plus facilement. En complément, il est très commode d'éditer le code source du script pour y récupérer des informations de debug. Par exemple, la commande `stack` affiche la pile. La commande `=` dépile l'élément du dessus de la pile et l'affiche sur la sortie standard s'il est d'un type basique (entier ou chaîne de caractères). La commande `==` fait la même chose, à la seule différence qu'elle produit un affichage même si l'élément dépilé est par exemple un tableau. Ainsi, pour afficher la valeur qui sera en haut de la pile à tel ou tel moment, on peut insérer les instructions `dup =` dans le code, pour dupliquer l'élément puis le dépiler et l'afficher. Pour finir, j'ai souvent utilisé l'enchaînement `stack quit` afin d'afficher la pile et de quitter immédiatement le programme (pratique dans les grosses boucles).

3.4 Cassage des quatre premiers octets de la clé

Voici une liste non exhaustive des différentes actions réalisées par ce script :

1. Les fonctions I1, I2, I3 et I4 sont déclarées. La syntaxe n'est pas très intuitive, mais l'unique but de ces fonctions est de fournir un accès aux données qu'elles contiennent, à savoir un gros bloc de données non-identifiées par fonction.
2. La clé passée en argument du script est récupérée et il est vérifié qu'elle fait bien 16 octets de long.
3. Les données de I1 sont placées sur la pile, dans un tableau, mais sans être exploitées pour l'instant.
4. Les données de I2 sont à leur tour placées sur la pile.
5. Le troisième et le quatrième octet de la clé sont récupérés et dédoublés.
6. Un XOR est effectué entre 4 octets de I2 et les 2*2 octets de la clé. Le résultat est stocké et lors de la seconde itération, le XOR est cette fois effectué entre le résultat intermédiaire et les données suivantes de I2, et ainsi de suite.
7. La commande `exec` est appelée sur le résultat final obtenu après tous ces XOR. C'est à dire que ce résultat sera du code PostScript valide, qui sera interprété à son tour.
8. Puis on remarque que le code est dupliqué et que les mêmes opérations vont être effectuées, mais cette fois avec I3 et I4. La seule différence est que pour le XOR, c'est le premier et le deuxième octet de la clé qui sont manipulés.

Pour casser ces quatre octets de clé, il n'est nullement besoin d'essayer d'inverser les opérations effectuées. En effet, ce qui est intéressant avec ces XOR, c'est que

16. de seulement 772 pages... http://www-cdf.fnal.gov/offline/PostScript/PLRM2.pdf

chaque octet de clé est utilisé indépendamment des autres pour chiffrer les données. De plus, le sommet de la pile au moment de l'appel des deux `exec` doit être une chaîne de caractères contenant du code PostScript valide. Plutôt que d'exécuter ce code, on va donc modifier notre script pour l'afficher à la place sur la sortie standard. Pour ce faire, il suffit de remplacer chacun des deux `exec` par un `=`.

```
--- } for 0 1 1 {pop pop} for cvx exec
+++ } for 0 1 1 {pop pop} for cvx =
```

Les chaînes affichées lors de l'exécution du script font respectivement 2888 et 2280 octets de long. On constate bien qu'elles varient en fonction des quatre premiers octets de la clé. Pour casser la clé plus facilement, l'astuce est de se rappeler que dans du code PostScript, le caractère le plus souvent utilisé est de loin l'espace. Ainsi, en testant 256 clés et en comptant à chaque fois le nombre d'espaces récupérés sur la sortie, il est possible de trouver simultanément les 4 premiers octets de la clé en seulement quelques lignes de Python et quelques secondes d'exécution. À noter que Ghostscript est utilisé avec des options qui désactivent l'affichage et optimisent la vitesse d'exécution. En effet, sans celles-ci, un aperçu est ouvert à chaque exécution ce qui en plus d'être lent fait planter à la longue mon gestionnaire de fenêtres.

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-
import subprocess

for k in range(256):
    key = chr(k).encode("hex")*4 + "01"*12
    o = subprocess.check_output(["gs", "-dNOPAUSE", "-dNODISPLAY",
            "-q", "--", "atad_bruteforce.ps", key])
    k3 = o[:2888:2]
    k4 = o[1:2888:2]
    k1 = o[2889::2]
    k2 = o[2890::2]

    for i, subset in enumerate([k1, k2, k3, k4]):
        if subset.count(" ") > 100:
            print "Found byte %i of key: %s" % (i+1, chr(k).encode("hex"))
```

```
$ python bruteforce.py
Found byte 4 of key: a8
Found byte 1 of key: ba
Found byte 2 of key: c9
Found byte 3 of key: f7
```

Le début de la clé secrète est donc `bac9f7a8`. Ce morceau permet de récupérer deux nouveaux scripts PostScript qu'il va falloir analyser afin de trouver la suite de la clé.

3.5 Cassage des 12 octets suivants

Le premier script généré sert à déclarer des fonctions (supposées de chiffrement/mélange), en particulier une fonction `calc`. Mais je n'ai pas eu besoin de l'analyser plus en détails. Le second script est plus utile. Voici son contenu, après une sommaire indentation :

```
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 bitshift 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 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
        3 -1 roll add 5 index 0 get length 4 idiv add 5 index 0 get length
        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 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 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 get 4 index 4 mul 4 getinterval 1 index 0 0 1 1
        {pop 2 index length} for
        exch 1 sub
        {
```

```
38              3 copy exch length getinterval 2 index mark 3 1 roll 0 1 3 -1
39              roll dup length 1 sub exch 4 1 roll
40              { dup 3 2 roll dup 5 1 roll exch get 3 1 roll exch dup 5 1 roll
41                exch get xor 3 1 roll
42              } for
43              pop pop ] dup length string 0 3 -1 roll
44              { 3 -1 roll dup 4 1 roll exch 2 index exch put 1 add } forall
45              pop 4 -1 roll dup 5 1 roll 3 1 roll dup 4 1 roll putinterval
46              exch pop
47          } for
48          pop pop 5 1 roll 2 copy 7 -3 roll pop pop
49      } repeat
50      pop 4 index 0 1 index { length add } forall
51      string 0 3 2 roll
52      { 3 copy putinterval length add } forall
53      pop
54
55      calc I3 16 string readstring pop ne
56      { 0 1 1073741823 {pop} for (Key is invalid. Exiting ...\n)
57        error flush quit
58      } if
59  } for
60  pop pop pop pop (output.bin) (w) file exch 1 index resetfile
61  {1 index exch writestring} forall closefile
```

Voici un résumé approximatif des opérations effectuées par ce script :

1. Le premier `getinterval` récupère 4 octets de clé. En l'occurrence les deux derniers ayant déjà été trouvés (f7a8) et les deux suivants, à trouver.
2. Une multitude d'opérations logiques sont appliquées sur la valeur de cette sous-partie de la clé.
3. Le résultat obtenu est comparé avec <55555555> et <aaaaaaaa>. En fonction de si le résultat est supérieur ou inférieur, une valeur (0, 1 ou -1) est placée sur la pile. Celle-ci influera sur les calculs suivants.
4. Puis on fait beaucoup d'autres choses toujours avec cette sous-clé. Et on répète toutes ces opérations 10240 fois.
5. Une valeur est calculée à partir des données de I3 et de la vilaine fonction `calc`. Puis cette valeur est comparée avec le résultat des 10240 itérations précédentes. S'ils sont identiques, on recommence tout, mais cette fois avec la sous-clé suivante (deux octets de la sous-clé qu'on vient de trouver + deux octets à deviner). Et ainsi de suite pour les 12 octets de clé restants.
6. Si la clé est trouvée dans son intégralité, un fichier *output.bin* est généré à partir du contenu de la pile.
7. Si par contre une des sous-clé n'est pas valide, on rentre dans une protection anti-bruteforce toute mignonne : une boucle `for`, qui au bout de 1073741823 itérations (donc plusieurs secondes d'exécution) nous informe que la clé est invalide.

J'avais commencé le fastidieux travail de débogage et de compréhension de ce script, d'une manière semblable à la méthodologie utilisée pour la première partie du niveau 3. J'avais pour objectif d'avoir une compréhension fine de l'algorithme utilisé pour mélanger et vérifier les morceaux de clé. J'avais commencé à le ré-implémenter en Python. Et puis j'ai fini par me rendre à l'évidence : après avoir terminé le niveau 2, mon cerveau avait fondu. Parvenir à tout comprendre était une peine perdue. Je me suis donc rabattu sur un objectif bien plus pragmatique : suivre le point d'entrée et le point de sortie des clés et essayer de trouver un moyen pour que l'ordinateur réfléchisse à ma place pendant qui j'irai me perdre dans les bras de Morphée.

Car oui, quand j'ai découvert que seulement deux nouveaux octets de clé étaient testés à chaque fois, c'est à la fois heureux et honteux que j'ai réalisé que cela ne faisait que 65536 cas à tester par morceau de clé. Reste à le répéter 6 fois et le tour est joué. Cependant, pour faire fonctionner le bruteforce, il va falloir réfléchir une dernière petite fois à la bonne manière de procéder. Tout d'abord, il est nécessaire de fusionner ce script avec le tout premier. Cette étape est assez simple, il suffit de remplacer le second `exec` par un `pop` et de copier-coller notre script à la suite. Ensuite, il faut contourner la protection anti-bruteforce et disposer d'un moyen de savoir si la clé testée est la bonne ou non. Voici le diff de ma modification :

```
calc I3 16 string readstring pop ne
--- { 0 1 1073741823 {pop} for (Key is invalid. Exiting ...\n)
---   error flush quit
--- } if
+++ {(Fail) = quit } {(Win) = } ifelse
```

De cette manière, si la sous-clé est fausse, on affiche « *Fail* » sur la sortie standard et on quitte le programme. Si elle est correcte, on affiche « *Win* » et on continue l'exécution pour tester le morceau suivant.

La dernière étape est de coder un script de bruteforce intelligent [17] qui soit capable de trouver tous les morceaux de clé tout seul pour que je puisse passer une bonne nuit de sommeil, plutôt qu'un script qui trouve un morceau, qui ait besoin d'être relancé pour la suite et qui donc m'oblige à me réveiller toutes les heures.

Soyez rassurés, ma tentative fut un lamentable échec puisque j'avais laissé une typo dans mon programme et qu'il s'est planté après avoir trouvé le premier morceau de clé. Mais voici à présent une version qui fonctionne :

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-
import subprocess, sys, multiprocessing, itertools, time

choices = list("".join(x) for x in itertools.product("0123456789abcdef",
        repeat=4))
```

17. La présence de ces deux mots côte à côte est fortement improbable

```
secret_key = multiprocessing.Manager().list()
secret_key.extend(["ba", "c9", "f7", "a8"])

def bruteforce(key, guess):
    c = 0
    l = len(key)
    padding = ""
    if l < 14:
        padding = (14 - l) * "01"
    key = "".join(key)
    for g in guess:
        c += 1
        k = key + g + padding
        o = subprocess.check_output(["gs", "-dNOPAUSE", "-dNODISPLAY",
                "-q", "--", "atad_bruteforce_bis.ps", k])
        if c % 100 == 0:
            print c
        if o.count("Win") != (l-4)/2:
            print "FOUND!!!!"
            print k
            secret_key.extend([ g[:2], g[2:] ])
            sys.exit(0)

while len(secret_key) < 16:
    l = len(secret_key)
    pool = []
    for i in range(6):
        p = multiprocessing.Process(target=bruteforce,
                args=(secret_key, choices[i*11000:(i+1)*11000]))
        p.start()
        pool.append(p)
    while True:
        time.sleep(10)
        if len(secret_key) > l or len(secret_key) > 16:
            for p in pool:
                if p.is_alive():
                    p.terminate()
            break
```

Pour savoir si la sous-clé testée est la bonne, on compte le nombre de chaînes « *Win* » sur la sortie standard. Le script lance six processus et vérifie ponctuellement si l'un d'entre eux a trouvé une sous-clé. Le cas échéant, les processus sont stoppés puis relancés avec la nouvelle clé afin de trouver le morceau suivant.

Au bout d'environ 7 heures d'exécution [18], la clé entière a été trouvée :

18. Durée qui pourrait être diminuée si l'on dispose de plusieurs machines car les calculs sont entièrement parallélisables

```
bac9f7a8721fad3c9fcf271eed9abbc8
```

En l'utilisant sur le script initial *atad*, un fichier *output.bin* est créé.

Victoire !!! Victoire ?!?

4 Niveau bonus

4.1 Récupération de l'adresse e-mail de validation

Non, pas tout à fait. Le fichier *output.bin* est au format vCard. Il peut être ouvert avec un éditeur de texte classique.

```
$ file output.bin
output.bin: vCard visiting card
```

On trouve à l'intérieur une liste de contacts avec pour chacun d'eux un nom, une adresse, un numéro de téléphone et un e-mail. L'un de ces contacts porte comme prénom « *Challenge SSTIC* ». Un prénom pour le moins original et peu usité mais qui, il faut l'avouer, est doté d'un certain charme. Pour un enfant, au quotidien, ça doit quand même être assez difficile à porter. Bref.

```
BEGIN:VCARD
VERSION:2.1
FN:Challenge SSTIC
N:Challenge;SSTIC
ADR;WORK;PREF;QUOTED-PRINTABLE:;Campus Beaulieu;Rennes
TEL;CELL:
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
```

En lieu et place de l'adresse e-mail tant désirée, on trouve une longue chaîne de caractères qui est manifestement une liste d'appels système. Je perçois ici tout à fait le léger ricanement des concepteurs qui se délectent de nous faire languir 20 minutes supplémentaires, nous qui pensions être arrivé au bout de nos peines. La souris était déjà prête à cliquer pour envoyer l'e-mail salvateur. Mais non, il va falloir fournir un dernier petit effort.

On devine facilement que c'est un encodage simple où chaque appel système correspond à un caractère. La première étape est de déterminer à quelle architecture ces appels système font référence. Certains *syscall* sont utilisés partout mais ont une valeur différente selon l'architecture, tandis que d'autres sont plus spécifiques. La manière la plus directe d'arriver à nos fins est de récupérer une liste unique des appels système utilisés, puis d'une manière un peu sale mais tellement efficace, de googler cette liste. Tous les liens pointent alors vers les appels système du noyau Linux 64 bits.

Le hasard fait bien les choses puisque j'utilise justement ce noyau. La liste des appels système peut donc être récupérée localement, dans le fichier */usr/include/x86_64-linux-gnu/asm/unistd_64.h*. Pour vérifier que ce sont les bons appels système, on peut s'aider du fait que l'adresse e-mail, une fois décodée, se terminera par *@challenge.sstic.org*.

On prend donc par exemple le syscall correspondant au dernier caractère, à savoir `sys_syslog` et on regarde sa valeur dans *unistd_64.h* :

```
#define __NR_syslog                          103
__SYSCALL(__NR_syslog, sys_syslog)
```

Cet appel système est associé à l'entier 103, ce qui est également le code ASCII du caractère "g". Tout va au mieux dans le meilleur des mondes. Pour forger un dictionnaire de conversion entre le nom d'un syscall et sa valeur entière, il suffit d'un *one-liner* Python avec une regex bien sentie :

```python
import re
print dict( (v, int(k)) for k, v in re.findall(
    r"#define .*?(\d+)\s*__SYSCALL.+?, (\w+)\)",
    open("/usr/include/x86_64-linux-gnu/asm/unistd_64.h").read()))
```

Ensuite, on parcourt la chaîne de la vCard et on remplace chaque syscall par le caractère ASCII correspondant à sa valeur. On obtient ainsi l'adresse e-mail de validation du challenge :

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

4.2 Allons un peu plus loin

Après avoir envoyé un e-mail à cette adresse puis avoir laissé exprimer ma joie en effectuant une danse rituelle autour de ma chaise de bureau, j'ai réalisé que ce dernier petit niveau bonus ne m'avait apporté aucune nouvelle connaissance utile. Mais il n'est pas encore trop tard pour y remédier.

En voyant l'adresse e-mail encodée, cela m'a fait penser aux codecs Python. Les codecs, ce sont ces choses qui peuvent être invoquées depuis un objet `string` grâce aux méthodes `encode` et `decode`. Les deux codecs qui sont probablement les plus couramment utilisés sont `hex` et `base64`. Mais il en existe d'autres, moins connus mais qui peuvent se révéler puissants, comme par exemple `rot13`, `uu` ou `zlib`.

```
>>> "toto".encode("base64")
'dG90bw==\n'
>>> "746f746f".decode("hex")
'toto'
>>> "toto".encode("rot13")
'gbgb'
>>> "toto".encode("uu")
'begin 666 <data>\n$=&]T;P  \n \nend\n'
>>> "toto".encode("zlib")
'x\x9c+\xc9/\xc9\x07\x00\x04x\x01\xc7'
```

Je me suis donc mis en tête de résoudre ce niveau un peu plus élégamment en apprenant comment faire pour créer mon codec `syscall`. La documentation [19] nous indique la démarche à suivre : il faut créer une fonction de recherche qui à partir d'un nom donné retourne un objet `CodecInfo`. Cet objet possédera un lien vers les fonctions `encode` et `decode` souhaitées. Pour finir, il faut enregistrer la fonction de recherche. Voici le code de mon codec `syscall` :

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import codecs

encoding_map = {
0:'sys_read', 1:'sys_write', 2:'sys_open', 3:'sys_close',
4:'sys_newstat', 5:'sys_newfstat', 6:'sys_newlstat', 7:'sys_poll',
8:'sys_lseek', 9:'sys_mmap', 10:'sys_mprotect', 11:'sys_munmap',
12:'sys_brk', 13:'sys_rt_sigaction', 14:'sys_rt_sigprocmask',
15:'stub_rt_sigreturn', 16:'sys_ioctl', 17:'sys_pread64',
18:'sys_pwrite64', 19:'sys_readv', 20:'sys_writev', 21:'sys_access',
22:'sys_pipe', 23:'sys_select', 24:'sys_sched_yield',
25:'sys_mremap', 26:'sys_msync', 27:'sys_mincore', 28:'sys_madvise',
```

19. http://docs.python.org/2/library/codecs.html

```
29:'sys_shmget', 30:'sys_shmat', 31:'sys_shmctl', 32:'sys_dup',
33:'sys_dup2', 34:'sys_pause', 35:'sys_nanosleep',
36:'sys_getitimer', 37:'sys_alarm', 38:'sys_setitimer',
39:'sys_getpid', 40:'sys_sendfile64', 41:'sys_socket',
42:'sys_connect', 43:'sys_accept', 44:'sys_sendto',
45:'sys_recvfrom', 46:'sys_sendmsg', 47:'sys_recvmsg',
48:'sys_shutdown', 49:'sys_bind', 50:'sys_listen',
51:'sys_getsockname', 52:'sys_getpeername', 53:'sys_socketpair',
54:'sys_setsockopt', 55:'sys_getsockopt', 56:'stub_clone',
57:'stub_fork', 58:'stub_vfork', 59:'stub_execve', 60:'sys_exit',
61:'sys_wait4', 62:'sys_kill', 63:'sys_newuname', 64:'sys_semget',
65:'sys_semop', 66:'sys_semctl', 67:'sys_shmdt', 68:'sys_msgget',
69:'sys_msgsnd', 70:'sys_msgrcv', 71:'sys_msgctl', 72:'sys_fcntl',
73:'sys_flock', 74:'sys_fsync', 75:'sys_fdatasync',
76:'sys_truncate', 77:'sys_ftruncate', 78:'sys_getdents',
79:'sys_getcwd', 80:'sys_chdir', 81:'sys_fchdir', 82:'sys_rename',
83:'sys_mkdir', 84:'sys_rmdir', 85:'sys_creat', 86:'sys_link',
87:'sys_unlink', 88:'sys_symlink', 89:'sys_readlink', 90:'sys_chmod',
91:'sys_fchmod', 92:'sys_chown', 93:'sys_fchown', 94:'sys_lchown',
95:'sys_umask', 96:'sys_gettimeofday', 97:'sys_getrlimit',
98:'sys_getrusage', 99:'sys_sysinfo', 100:'sys_times',
101:'sys_ptrace', 102:'sys_getuid', 103:'sys_syslog',
104:'sys_getgid', 105:'sys_setuid', 106:'sys_setgid',
107:'sys_geteuid', 108:'sys_getegid', 109:'sys_setpgid',
110:'sys_getppid', 111:'sys_getpgrp', 112:'sys_setsid',
113:'sys_setreuid', 114:'sys_setregid', 115:'sys_getgroups',
116:'sys_setgroups', 117:'sys_setresuid', 118:'sys_getresuid',
119:'sys_setresgid', 120:'sys_getresgid', 121:'sys_getpgid',
122:'sys_setfsuid', 123:'sys_setfsgid', 124:'sys_getsid',
125:'sys_capget', 126:'sys_capset', 127:'sys_rt_sigpending',
128:'sys_rt_sigtimedwait', 129:'sys_rt_sigqueueinfo',
130:'sys_rt_sigsuspend', 131:'stub_sigaltstack', 132:'sys_utime',
133:'sys_mknod', 134:'sys_ni_syscall', 135:'sys_personality',
136:'sys_ustat', 137:'sys_statfs', 138:'sys_fstatfs',
139:'sys_sysfs', 140:'sys_getpriority', 141:'sys_setpriority',
142:'sys_sched_setparam', 143:'sys_sched_getparam',
144:'sys_sched_setscheduler', 145:'sys_sched_getscheduler',
146:'sys_sched_get_priority_max', 147:'sys_sched_get_priority_min',
148:'sys_sched_rr_get_interval', 149:'sys_mlock', 150:'sys_munlock',
151:'sys_mlockall', 152:'sys_munlockall', 153:'sys_vhangup',
154:'sys_modify_ldt', 155:'sys_pivot_root', 156:'sys_sysctl',
157:'sys_prctl', 158:'sys_arch_prctl', 159:'sys_adjtimex',
160:'sys_setrlimit', 161:'sys_chroot', 162:'sys_sync',
163:'sys_acct', 164:'sys_settimeofday', 165:'sys_mount',
166:'sys_umount', 167:'sys_swapon', 168:'sys_swapoff',
169:'sys_reboot', 170:'sys_sethostname', 171:'sys_setdomainname',
172:'stub_iopl', 173:'sys_ioperm', 174:'sys_ni_syscall',
175:'sys_init_module', 176:'sys_delete_module', 177:'sys_ni_syscall',
```

```
178:'sys_ni_syscall', 179:'sys_quotactl', 180:'sys_ni_syscall',
181:'sys_ni_syscall', 182:'sys_ni_syscall', 183:'sys_ni_syscall',
184:'sys_ni_syscall', 185:'sys_ni_syscall', 186:'sys_gettid',
187:'sys_readahead', 188:'sys_setxattr', 189:'sys_lsetxattr',
190:'sys_fsetxattr', 191:'sys_getxattr', 192:'sys_lgetxattr',
193:'sys_fgetxattr', 194:'sys_listxattr', 195:'sys_llistxattr',
196:'sys_flistxattr', 197:'sys_removexattr', 198:'sys_lremovexattr',
199:'sys_fremovexattr', 200:'sys_tkill', 201:'sys_time',
202:'sys_futex', 203:'sys_sched_setaffinity',
204:'sys_sched_getaffinity', 205:'sys_ni_syscall',
206:'sys_io_setup', 207:'sys_io_destroy', 208:'sys_io_getevents',
209:'sys_io_submit', 210:'sys_io_cancel', 211:'sys_ni_syscall',
212:'sys_lookup_dcookie', 213:'sys_epoll_create',
214:'sys_ni_syscall', 215:'sys_ni_syscall',
216:'sys_remap_file_pages', 217:'sys_getdents64',
218:'sys_set_tid_address', 219:'sys_restart_syscall',
220:'sys_semtimedop', 221:'sys_fadvise64', 222:'sys_timer_create',
223:'sys_timer_settime', 224:'sys_timer_gettime',
225:'sys_timer_getoverrun', 226:'sys_timer_delete',
227:'sys_clock_settime', 228:'sys_clock_gettime',
229:'sys_clock_getres', 230:'sys_clock_nanosleep',
231:'sys_exit_group', 232:'sys_epoll_wait', 233:'sys_epoll_ctl',
234:'sys_tgkill', 235:'sys_utimes', 236:'sys_ni_syscall',
237:'sys_mbind', 238:'sys_set_mempolicy', 239:'sys_get_mempolicy',
240:'sys_mq_open', 241:'sys_mq_unlink', 242:'sys_mq_timedsend',
243:'sys_mq_timedreceive', 244:'sys_mq_notify',
245:'sys_mq_getsetattr', 246:'sys_kexec_load', 247:'sys_waitid',
248:'sys_add_key', 249:'sys_request_key', 250:'sys_keyctl',
251:'sys_ioprio_set', 252:'sys_ioprio_get', 253:'sys_inotify_init',
254:'sys_inotify_add_watch', 255:'sys_inotify_rm_watch'
}

decoding_map = dict((v,k) for k, v in encoding_map.iteritems())

class SyscallCodecException(Exception):
    pass

def syscall_encode(s):
    return " ".join(encoding_map[ord(c)] for c in s), len(s)

def syscall_decode(s):
    r = []
    for syscall in s.split():
        if syscall:
            try:
                r.append(chr(decoding_map[syscall]))
            except KeyError:
                raise SyscallCodecException(
```

```
                "Decoding error: not a valid sstic-syscall string")
    return "".join(r), len(s)

def _lookup(name):
    if name == "syscall":
        return codecs.CodecInfo(
            name = name,
            encode = syscall_encode,
            decode = syscall_decode)

codecs.register(_lookup)
```

Ensuite, on peut décoder l'adresse e-mail en appelant directement la méthode `decode("syscall")` sur la chaîne encodée :

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import syscall

mail="""sys_socketpair stub_fork sys_socketpair
sys_getsockoptsys_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"""

print mail.decode("syscall")
```

Et c'est ainsi que le cookie s'émiette [20].

Conclusion

Merci pour cette édition 2013 du challenge SSTIC que j'ai personnellement trouvé très bien dosée. Un premier niveau abordable qui permet au plus grand nombre de démarrer le challenge sans se casser immédiatement les dents. Un second niveau complexe mais qui m'a permis de découvrir un domaine qui m'était complètement inconnu. Un troisième niveau relativement long mais qui se fait bien et qui m'a

20. http://www.kaakook.fr/citation-7529

initié au langage PostScript. Et un dernier niveau en forme de clin d'œil. Je vais maintenant conclure avec ce mot de la fin :

```
sys_ftruncate sys_ptrace sys_setregid sys_sysinfo sys_setuid
sys_dup sys_llistxattr sys_setrlimit sys_dup sys_setgroups
sys_getpgrp sys_setresuid sys_getgroups sys_dup sys_ptrace
sys_setgroups sys_dup sys_llistxattr sys_setrlimit sys_dup
sys_getegid sys_getpid sys_getrlimit sys_getppid sys_getppid
sys_llistxattr sys_reboot sys_ptrace sys_dup sys_setsid
sys_setregid sys_getpgrp sys_sysinfo sys_getgid sys_getrlimit
sys_setuid sys_getppid sys_ptrace sys_sendmsg sys_dup
sys_fsync sys_getpid sys_ptrace sys_getgroups sys_setsid
sys_llistxattr sys_swapoff sys_setregid sys_ptrace sys_dup
sys_setreuid sys_setresuid sys_ptrace sys_dup sys_getresuid
sys_getpgrp sys_setresuid sys_getgroups sys_dup sys_getppid
sys_getpid sys_getrlimit sys_getresuid sys_ptrace sys_setfsuid
sys_dup sys_setsid sys_getrlimit sys_getgroups sys_dup
sys_llistxattr sys_reboot sys_setgroups sys_llistxattr
sys_reboot sys_dup sys_setuid sys_setpgid sys_setsid
sys_getpgrp sys_setregid sys_setgroups sys_setresuid
sys_getppid sys_llistxattr sys_reboot sys_getgroups sys_dup
sys_setsid sys_getrlimit sys_setregid sys_dup sys_setpgid
sys_getrlimit sys_dup sys_setsid sys_setregid sys_getpgrp
sys_setsid sys_ptrace sys_getppid sys_getgroups sys_setuid
sys_getpgrp sys_getppid sys_dup sys_llistxattr sys_setrlimit
sys_dup sys_getegid sys_getrlimit sys_dup sys_getgroups
sys_getpgrp sys_getegid sys_setuid sys_getegid sys_getpgrp
sys_setreuid sys_setresuid sys_setuid sys_ptrace sys_dup
stub_vfork sys_socket sys_dup sys_msgsnd sys_setgroups
sys_dup sys_getrusage sys_setregid sys_getrlimit sys_getresuid
sys_getpgrp sys_dup sys_ptrace sys_setgroups sys_dup sys_setpgid
sys_ptrace sys_setregid sys_sysinfo sys_setuid sys_dup
sys_llistxattr sys_setrlimit sys_dup sys_setgroups sys_getpgrp
sys_setuid sys_sendto sys_dup sys_getegid sys_ptrace sys_sysinfo
sys_setgroups sys_ptrace sys_setresuid sys_setregid sys_dup
sys_sysinfo sys_getpgrp sys_setresuid sys_setregid sys_getrlimit
sys_syslog sys_ptrace sys_setresuid sys_getresgid sys_sendto
sys_dup sys_setreuid sys_setresuid sys_setuid sys_dup
sys_getrlimit sys_dup sys_setsid sys_setregid sys_setuid
sys_getgroups sys_dup sys_getegid sys_getrlimit sys_dup
sys_setsid sys_ptrace sys_setuid sys_getppid sys_ptrace sys_dup
sys_times sys_ptrace sys_dup sys_times sys_llistxattr sys_reboot
sys_sysinfo sys_getpgrp sys_times sys_ptrace sys_setregid sys_dup
sys_sysinfo sys_ptrace sys_dup sys_setpgid sys_ptrace
sys_getgroups sys_getgroups sys_getrlimit sys_syslog sys_ptrace
sys_dup2
```