# Solution du challenge SSTIC 2017

Pierre Bienaimé

26 avril 2017

## Table des matières

# 1 Introduction

J'aime le challenge SSTIC !

Chaque année, c'est une formidable occasion de découvrir qu'on est complétement 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.

Le challenge SSTIC est important pour moi. Au delà de sa qualité et de son intérêt, j'y attache avant tout une valeur sentimentale. C'est grâce à lui, en 2011, que j'ai appris les bases de l'assembleur et que je me suis découvert une affinité pour la sécurité informatique et pour ses challenges. Aujourd'hui, après en avoir résolu 5 et participé à en concevoir un, je constate que j'ai encore appris beaucoup de choses et que l'excitation est toujours au rendez-vous.

Paternité et vie de famille oblige, chaque année c'est de plus en plus difficile de dégager du temps libre pour m'attaquer à ce challenge. J'en profite donc pour remercier ma femme (enfin surtout pour m'excuser...) qui a des raisons forts légitimes de me détester chaque année, autour des mois de mars et avril :)

Cette édition 2017, composée de 5 épreuves, est placée sur le thème de Rayman (j'y ai joué sur Playstation 1 ! ça remonte un peu...). L'objectif du challenge est de trouver une adresse email au format ...@sstic.org. Pour celà, il faudra récupérer des clés en terminant les épreuves. Cette année, petite nouveauté : des secrets intermédiaires facultatifs (baptisés LUM [1]) ont été cachés au fil des épreuves.

# 2 Electronic Flash

## 2.1 Présentation

Le challenge commence avec le fichier firmware.eml, un email stocké au format texte contenant deux pièces jointes.

Le corps du message nous indique qu'une des pièces jointes est une trace réalisée par un analyseur logique lors de l'écriture d'un firmware sur une flash NAND.

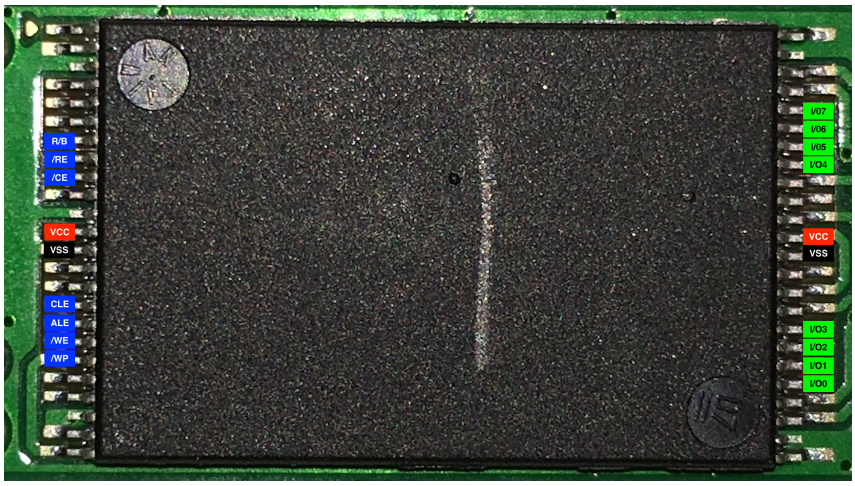
1. en référence aux objets que l'on doit collecter dans les jeux Rayman

```
Bonjour,

La chaine de production est prete pour produire votre nouvelle console de jeux.

Nous avons programme un premier modele avec le firmware que vous nous avez fourni.
Avant de lancer la production, merci de valider que le firmware ecrit sur ce premier modele est bien
celui que vous souhaitez deployer sur l'ensemble de vos consoles.
Pour cela, et afin de ne pas faire d'erreur, nous avons enregistre avec un analyseur logique la
programmation de la memoire flash NAND, vous trouverez en piece jointe cet enregistrement. Nous
utilisons le logiciel sigrok, vous pouvez recuperer le firmware avec le decodeur parallele sur
le bus de donnees. Nous utilisons uniquement l'interface graphique "pulseview", mais il me semble
qu'il y a aussi des bindings python et une interface CLI, qui vous permettront certainement
d'automatiser l'extraction.

Dans l'attente de votre validation.

Cordialement,
John
```

Pour extraire les pièces jointes, plutôt que de bricoler à la main avec les multi-part base64, pour une fois j'ai opté pour une solution propre [2] et j'ai utilisé munpack.

```
$ munpack firmware.eml
NAND_pinout.jpg (image/jpeg)
NAND_FLASH_writes_no_OOB_5MHz.sr (application/octet-stream)
```

La première image est un schéma représentant le rôle des différentes pins de la flash

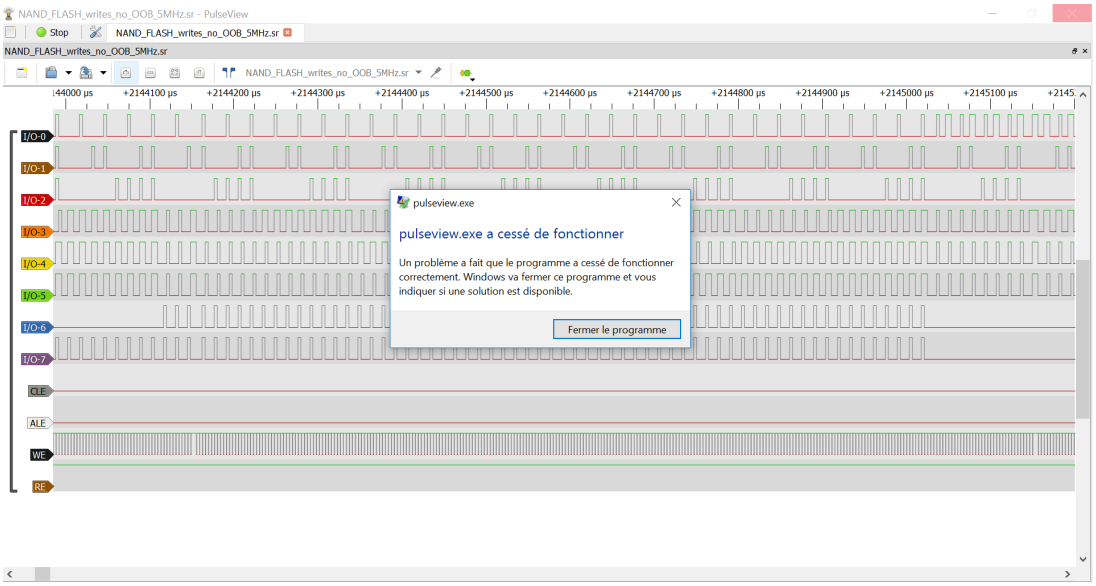



Le second fichier est une sauvegarde pour le logiciel sigrok. Pour examiner cette capture d'analyseur logique et en extraire le contenu écrit sur la flash NAND, l'email nous conseille d'utiliser le décodeur parallèle sur le bus de données du logiciel sigrok, ainsi que l'interface graphique Pulseview.

Je ne vais pas trop disserter sur les péripéties liées à la compilation de Pulseview sur ma distribution Linux. Ni sur la version Windows 10 qui plante toutes les minutes. Ni sur les bindings Python sigrok [3] qui pointaient sur une page blanche au moment de

2. En rédigeant cette solution, j'ai découvert accidentellement qu'on pouvait tout simplement double-cliquer sur le fichier eml pour l'ouvrir dans Thunderbird :)
3. http://sigrok.org/api/libsigrok/unstable/bindings/python/index.html

la résolution du challenge. Au final, je n'ai utilisé ni sigrok ni Pulseview pour résoudre cette épreuve, car la sauvegarde au format *.sr* s'est avérée suffisament pratique pour être utilisée directement.




## 2.2 Extraction du firmware

Le fichier *.sr* est une archive zip qui renferme 1,6 Go de fichiers contenant les données brutes des différents signaux. On y trouve également un fichier *metadata*.

```
probe1=I/O-0
probe2=I/O-1
probe3=I/O-2
probe4=I/O-3
probe5=I/O-4
probe6=I/O-5
probe7=I/O-6
probe8=I/O-7
probe9=CLE
probe10=ALE
probe11=WE
probe12=RE
```

Disclaimer : n'y connaissant pas grand-chose en analyseur logique, j'utilise peut-être du vocabulaire imprécis ou incorrect.

Ce fichier nous indique que l'analyseur logique a enregistré l'état de 12 signaux, dont 8 correspondent au bus de données (`I/O-0` à `I/O-7`). Pour comprendre la signification des signaux CLE, ALE, WE et RE, j'ai feuilleté la spécification Open Nand Flash Interface [4].

Il n'y a pas de clock. La synchronisation se fait grâce aux signaux RE (read) et WE (write). Pour lire ou écrire, il faut tout d'abord faire passer du courant sur ces deux pins, puis faire osciller l'une des deux. La valeur courante des 8 pins du bus de données va

4. http://www.onfi.org/~/media/onfi/specs/onfi_4_0-gold.pdf?la=en

alors servir à coder un octet. Les signaux CLE (command) et ALE (address) sont des sortes de flags permettant d'indiquer que le bus de données va coder respectivement des commandes ou des adresses mémoires.

Je n'ai pas vraiment compris si la synchronisation se fait sur le front descendant ou le front montant. Mais en tatonnant un peu, j'ai fini par obtenir le bon résultat.

La première étape a été de faire un prétraitement sur les 1.6 Go de données brutes du fichier de sauvegarde sigrok. Le format s'avère être assez simple : la valeur des signaux est codée sur deux octets. Le premier correspond au bus de données, le second aux signaux CLE, ALE, WE et RE. Cependant, à cause de l'échantillonnage, l'information utile se retrouve fortement dupliquée. J'ai donc fait un petit script qui lit les changements de valeur des signaux WE et RE pour supprimer toutes les informations redondantes. On passe ainsi de 1.6 Go à 69 Mo de données.

Ensuite, il suffit de parser les données et d'implémenter les commandes utilisées au fur et à mesure qu'on les rencontre, en se basant sur la spécification ONFI.

```python
CLE = 1 # command
ALE = 2 # address
WE = 4  # write
RE = 8  # read

def iterator():
    with open("nand.data") as f:
        while True:
            d = f.read(2)
            if not d:
                return
            flags = ord(d[1])
            yield flags, d[0]

def b2i(b):
    return int(str(b[::-1]).encode("hex"), 16)

class NandCommandHandler:
    def __init__(self):
        self.it = iterator()
        self.mem = {}

    def run(self):
        flags, byte = self.it.next()
        while True:
            if flags & CLE:
                r = self.handle_command(ord(byte))
                if r:
                    flags, byte = r
                else:
                    try:
                        flags, byte = self.it.next()
                    except StopIteration:
                        break
            else:
                raise Exception("Need a command (got flags=%X byte=%X)" % (flags, ord(byte)))
        self.get_firmware()

    def get_firmware(self):
        fw_name = "firmware.data"
        with open(fw_name, "wb") as f:
            offset = 0
            for k in sorted(self.mem):
                if k == offset:
```

```
45                      f.write(self.mem[k])
46                      offset += 1
47                  else:
48                      raise Exception("Missing firmware data at offset 0x%X" % offset)
49          print "Firmware saved into file: ", fw_name
50
51      def handle_command(self, cmd):
52          if cmd == 0xff: # RESET
53              print "RESET"
54          elif cmd == 0x90: # READ ID
55              msg = "READ ID (%s): %s"
56              addr = ord(self.get_addr(1))
57              if addr == 0:
58                  t = "Manufacturer ID"
59              elif addr == 0x20:
60                  t = "ONFI signature"
61              else:
62                  raise Exception("Bad READ ID type 0x%X" % addr)
63              data, next_cmd = self.get_data()
64              print msg % (t, data)
65              return next_cmd
66          elif cmd == 0x60: # BLOCK ERASE
67              addr = b2i(self.get_addr(3))
68              self.get_cmd(0xD0)
69              print "BLOCK ERASE %06X" % addr
70          elif cmd == 0x80: # PAGE PROGRAM
71              addr = self.get_addr(5)
72              column = b2i(addr[:2])
73              row = b2i(addr[2:])
74              data = self.get_data(end_cmd=0x10)
75              #print "PAGE PROGRAM column %04X, row %06X (%i bytes)" % (column, row, len(data))
76              if column:
77                  raise Exception("Todo: handle columns")
78              self.mem[row] = data
79          else:
80              raise Exception("Unknown command 0x%X" % cmd)
81
82      def get_addr(self, size):
83          addr = bytearray()
84          for i in range(size):
85              flags, byte = self.it.next()
86              if not flags & ALE:
87                  raise Exception("Not a valid address flags=%X byte=%X" % (flags, ord(byte)))
88              addr.append(byte)
89          return addr
90
91      def get_data(self, end_cmd=None):
92          data = bytearray()
93          for flags, byte in self.it:
94              if not flags & CLE and not flags & ALE:
95                  data.append(byte)
96              else:
97                  break
98          if end_cmd:
99              if not flags & CLE or ord(byte) != end_cmd:
100                 raise Exception("Need command %X (got flags=%X byte=%X)" % (end_cmd, flags, ord(byte)))
101             return data
102         return data, (flags, byte)
103
104     def get_cmd(self, cmd):
105         flags, byte = self.it.next()
106         if not flags & CLE or ord(byte) != cmd:
107             raise Exception("Need command %X (got flags=%X byte=%X)" % (cmd, flags, ord(byte)))
108
109 if __name__ == '__main__':
110     n = NandCommandHandler()
111     n.run()
```

```
$ python nand.py

RESET
READ ID (Manufacturer ID): SSTIC PSEUDO NAND   LUM{x.g215WiPCR}
BLOCK ERASE 000000
BLOCK ERASE 000800
BLOCK ERASE 000E00
BLOCK ERASE 000100
BLOCK ERASE 001000
BLOCK ERASE 000900
BLOCK ERASE 000400
BLOCK ERASE 000500
BLOCK ERASE 000700
BLOCK ERASE 000200
BLOCK ERASE 000C00
BLOCK ERASE 000B00
BLOCK ERASE 000A00
BLOCK ERASE 000D00
BLOCK ERASE 000F00
BLOCK ERASE 000600
BLOCK ERASE 000300
Firmware saved into file:  firmware.data
```

La première commande envoyée à la flash NAND sert à lire le Manufacturer ID, dans lequel un premier LUM a été caché. Ensuite, le firmware est écrit sur la flash avec une succession de commandes PAGE PROGAM (non affichées ici), entrecoupées de quelques BLOCK ERASE pour, j'imagine, initialiser les blocs de données qui vont être écrits.

Le firmware ainsi récupéré fait 34 Mo. Il s'agit d'une partition FAT32.

```
$ file firmware.data

firmware.data: DOS/MBR boot sector, code offset 0x58+2, OEM-ID "mkfs.fat", Media descriptor 0xf8,
sectors/track 63, heads 255, sectors 67584 (volumes > 32 MB) , FAT (32 bit), sectors/FAT 520,
serial number 0xf0fa166b, unlabeled
```

Quand on monte cette partition, on récupère un fichier challenges.zip ainsi que son hash md5 permettant de vérifier si l'écriture du firmware s'est bien passée. Évidemment, dans mon cas, l'archive zip semblait valide et se décompressait correctement, mais son hash md5 était faux... Le problème venait de deux ou trois bits incorrects, car il y a quelques cas aux limites où la valeur du bus de données varie pendant une même écriture.

C'est assez suspect de nous fournir une partition entière pour y stocker un seul fichier. On s'inspire donc de la solution de 0xf4b au challenge SSTIC 2015 pour regarder ça de plus près. On constate qu'un fichier *lum.txt* a été supprimé mais qu'il est encore présent sur la partition.

```
$ fls firmware.data

r/r 5:        challenges.zip
r/r 8:        challenges.zip.md5
r/r * 10:        lum.txt
v/v 1064195:        $MBR
v/v 1064196:        $FAT1
v/v 1064197:        $FAT2
d/d 1064198:        $OrphanFiles

$ icat firmware.data 10

LUM{AsPBdVWz95y}
```

L'archive challenges.zip contient un site web à faire tourner localement. Celui-ci démarre une machine virtuelle Linux utilisant l'architecture Openrisc. Cette VM nous donne accès à deux nouvelles épreuves : *don't let him escape* et *riscy zones*. Une fois celles-ci résolues, l'épreuve *unstable machine*s sera débloquée. Voici les deux README que l'on trouve à la racine de l'archive et à la racine de la machine virtuelle

```
Bonjour,

Voici la suite du challenge, les epreuves suivantes se derouleront dans un
navigateur.  Pour fonctionner, ce dossier doit etre propulse par un serveur
web, par exemple avec python SimpleHTTPServer :

$ python -m SimpleHTTPServer 8000

Pour travailler plus confortablement tu peux te connecter a la machine
virtuelle fonctionnant dans ton navigateur en SSH.
Une passerelle Websocket<->Interface TAP est necessaire pour fournir du reseau
a la machine virtuelle. Le projet "go-websockproxy" [1] permet de creer cette
passerelle, il remplit egalement la fonction de serveur web.

Le binaire necessite des privileges eleves (les capabilities CAP_NET_RAW et
CAP_NET_ADMIN ainsi que le droit d'executer la commande "ip" avec ces
capabilities) pour creer une interface TAP :
sudo go-websockproxy --listen-address=127.0.0.1:8090 --static-directory=$CHALLENGE_DIR
--tap-ipv4=10.42.42.1/30

Ensuite il ne reste plus qu'a ouvrir ton navigateur a l'adresse
http://127.0.0.1:8090/main.html. Lorsque la machine virtuelle a fini de
demarrer, il est possible de se connecter en SSH :
ssh user@10.42.42.2, le mot de passe est "sstic".

[1] https://github.com/gdm85/go-websockproxy
```

```
Bienvenue aventurier !

Le Grand Protoon a ete enleve, l'equilibre du monde est rompu. Les Electroons
ont ete enfermes dans diverses epreuves, retrouve-les, et libere le Grand
Protoon pour que le monde retrouve sa stabilite.

Les epreuves qui t'attendent sont dans le dossier "/challenges", certaines
d'entre elles doivent etre prealablement deverrouillees.

Lorsque tu auras reussi a liberer un Electroon, ajoute-le a l'aide de la
commande "/challenges/tools/add_key". Les epreuves verrouillees se
deverrouilleront automatiquement.

Dans chaque niveau, des Lums sont caches, retrouve-les ! La plupart sont
dissimules sur le chemin vers l'Electroon. Pour les collectionner, ajoute-les
avec la commande "/challenges/tools/add_lum". Il n'est pas essentiel de
collecter l'ensemble des Lums pour finir ta quete.

Bonne chance !
```


## 2.3 Résumé des secrets

```
LUM{x.g215WiPCR}
```

```
LUM{AsPBdVWz95y}
```

# 3 Don't let him escape

## 3.1 Présentation

```
$ ls dont_let_him_escape/

bpf.py
mini-libc.so
server.py
```

En parcourant les 3 fichiers de cette épreuve, on comprend assez rapidement que le fichier server.py démarre une socket à laquelle est attachée un filtre BPF (Berkeley Packet Filter). Le but du jeu va être d'envoyer des paquets réseau à ce serveur qui valident le filtre BPF.

Comme beaucoup de monde, je connais un petit peu BPF, puisque c'est la syntaxe qu'on utilise, entre autres, à la fin des commandes `tcpdump` pour faire du filtrage (tcpdump -i eth0 port 80). Ce que j'ignorais totalement, c'est qu'en écrivant un filtre BPF, ça génère du bytecode qui sera exécuté dans une machine virtuelle BPF. C'est assez génial :)

Le fichier server.py utilise un filtre BPF sous forme d'un gros blob binaire encodé en base64. Il va donc falloir trouver un moyen de le désassembler.

## 3.2 Analyse statique

Après une lecture attentive de la documentation liée à l'assembleur BPF [5] et la tentative infructueuse d'utiliser les bpftools [6] pour désassembler voire même jitter le programme BPF, j'ai commencé un désassemblage à la main. J'ai alors rapidement constaté que les opcodes ne collaient pas du tout. Bien que les fichiers server.py et bpf.py font partout référence à "BPF", nous sommes ici en présence d'eBPF (extended BPF), une version proposant un jeu d'instructions plus complet.

J'ai trouvé un désassembleur eBPF en Python [7], qui marchait presque. Il manquait simplement quelques instructions spéciales. On obtient alors 1500 lignes d'assembleur qu'on commence à reverser de manière statique.

Au début de l'execution, le registre r1 pointe vers le paquet réseau.

5. https://www.kernel.org/doc/Documentation/networking/filter.txt
6. https://github.com/cloudflare/bpftools
7. https://github.com/iovisor/ubpf

```
0000 bf16000000000000  mov   r6, r1
0008 b707000000000000  mov   r7, 0x0
0010 280000000c000000  ldh   r0, r6_packet[0xc]
0018 5500d00400080000  jne   r0, 0x800, exit_R7
0020 3000000017000000  ldb   r0, r6_packet[0x17]
0028 5500ce0411000000  jne   r0, 0x11, exit_R7
0030 2800000010000000  ldh   r0, r6_packet[0x10]    # IP length (IP header + udp + ...)
0038 bf08000000000000  mov   r8, r0
0040 300000000e000000  ldb   r0, r6_packet[0xe]     # IP IHL (probably 0x5)
0048 bf09000000000000  mov   r9, r0
0050 2800000024000000  ldh   r0, r6_packet[0x24]
0058 5500c80439050000  jne   r0, 0x539, exit_R7     # UDP Dest Port must be 1337
0060 18010000f8ffffff  lddw  r1, 0xfffffff8
0070 0f18000000000000  add   r8, r1                 # ip_len - 8
0078 6709000002000000  lsh   r9, 0x2
0080 570900003c000000  and   r9, 0x3c               # r9 is the IP header size (usually 20 bytes)
0088 1f98000000000000  sub   r8, r9                 # r8 is now the custom UDP payload size (must be 16)
0090 7b9ae8ff00000000  stxdw [r10-24], r9
0098 0709000016000000  add   r9, 0x16               # r9 is IP header size + 0x16 --> sizeof(Ether) +
00A0 b701000000000000  mov   r1, 0x0                # sizeof(IP) + sizeof(UDP header).
00A8 7b1af8ff00000000  stxdw [r10-8], r1            # So r9 points to custom UDP payload
00B0 1811000003000000  lddw  r1, 0x3
00C0 bfa2000000000000  mov   r2, r10
00C8 07020000f8ffffff  add   r2, 0xfffffff8
00D0 8500000001000000  call  0x1
00D8 5500010000000000  jne   r0, 0x0, loc_00E8
00E0 0500020000000000  ja    loc_00F8

              loc_00E8:
00E8 7901000000000000  ldxdw r1, [r0]
00F0 55015b0000000000  jne   r1, 0x0, loc_03D0

              loc_00F8:
00F8 b707000000000000  mov   r7, 0x0
0100 6708000020000000  lsh   r8, 0x20
0108 7708000020000000  rsh   r8, 0x20
0110 5508b10410000000  jne   r8, 0x10, exit_R7          # Custom UDP payload must be 0x10
0118 6709000020000000  lsh   r9, 0x20
0120 7709000020000000  rsh   r9, 0x20
0128 5090000000000000  ldb   r0, r6_packet[r9+0x0]      # 1st byte of custom payload
0130 bf08000000000000  mov   r8, r0
0138 79a9e8ff00000000  ldxdw r9, [r10-24]
0140 0709000017000000  add   r9, 0x17
0148 5090000000000000  ldb   r0, r6_packet[r9+0x0]      # 2nd byte of custom payload
0150 57080000ff000000  and   r8, 0xff
0158 5508a8044c000000  jne   r8, 0x4c, exit_R7          # 1st custom byte must be "L"
0160 57000000ff000000  and   r0, 0xff
0168 5500a60455000000  jne   r0, 0x55, exit_R7          # 2nd custom byte must be "U"
0170 79a9e8ff00000000  ldxdw r9, [r10-24]
0178 bf91000000000000  mov   r1, r9
0180 070100001e000000  add   r1, 0x1e
0188 7b1ae0ff00000000  stxdw [r10-32], r1               # addr of 9th byte
0190 bf98000000000000  mov   r8, r9
0198 070800001a000000  add   r8, 0x1a
01A0 7b8ac8ff00000000  stxdw [r10-56], r8               # addr of 5th byte
01A8 4080000000000000  ldw   r0, r6_packet[r8+0x0]
01B0 7b0ad0ff00000000  stxdw [r10-48], r0               # 5th byte
01B8 bf98000000000000  mov   r8, r9
01C0 0708000022000000  add   r8, 0x22
01C8 4080000000000000  ldw   r0, r6_packet[r8+0x0]
01D0 7b0ad8ff00000000  stxdw [r10-40], r0               # 13, 14, 15, 16th byte
01D8 79a1e0ff00000000  ldxdw r1, [r10-32]
01E0 4010000000000000  ldw   r0, r6_packet[r1+0x0]
01E8 bf08000000000000  mov   r8, r0
01F0 0709000018000000  add   r9, 0x18
01F8 4890000000000000  ldh   r0, r6_packet[r9+0x0]
0200 6708000020000000  lsh   r8, 0x20
0208 79a1d8ff00000000  ldxdw r1, [r10-40]
0210 4f18000000000000  or    r8, r1
```

```
0218 180100007d41664d  lddw  r1, 0x456443724d66417d     # "EdCrMfA}"
0228 5d188e0400000000  jne   r8, r1, exit_R7
0230 79a1d0ff00000000  ldxdw r1, [r10-48]
0238 6701000020000000  lsh   r1, 0x20
0240 7701000020000000  rsh   r1, 0x20
0248 55018a0451577642  jne   r1, 0x42765751, exit_R7    # "BvWQ"
0250 57000000ffff0000  and   r0, 0xffff
0258 550088047b4d0000  jne   r0, 0x4d7b, exit_R7        # "M{"
0260 b707000001000000  mov   r7, 0x1
0268 7b7af8ff00000000  stxdw [r10-8], r7
0270 b701000000000000  mov   r1, 0x0
0278 7b1af0ff00000000  stxdw [r10-16], r1
0280 1811000003000000  lddw  r1, 0x3
0290 bfa2000000000000  mov   r2, r10
0298 07020000f0ffffff  add   r2, 0xfffffff0
02A0 bfa3000000000000  mov   r3, r10
02A8 07030000f8ffffff  add   r3, 0xfffffff8
02B0 b704000000000000  mov   r4, 0x0
02B8 8500000002000000  call  0x2
```

Chose assez rigolote, le BPF permet de faire des appels à certaines fonctions prédéfinies du kernel. Dans ce programme, c'est utilisé pour mettre en place une sorte de machine à états et pour stocker des variables globales qui seront persistantes entre les paquets réseaux.

De manière statique, on trouve que le filtre BPF attend un paquet sur le port UDP 1337, avec une charge utile de 16 octets valant LUM{BvWQEdCrMfA}. Si cette condition est remplie, la machine change d'état et le filtre se met à la recherche d'un second paquet, qui cette fois devra contenir la clé.

Seul soucis, les opérations qui vérifient la clé sont complexes et on arrive vite aux limites de l'analyse statique. Il va falloir des outils plus puissants. C'est pourquoi j'ai pris mon courage à deux mains, j'ai préparé consciencieusement mon sceau de cailloux, et j'ai téléchargé Miasm2 [8].

## 3.3 Exécution symbolique

Mon plan est d'utiliser le moteur d'exécution symbolique de Miasm pour trouver la clé. Pour celà il faut d'abord convertir l'assembleur eBPF en langage intermédiaire Miasm.

Il existe une manière propre d'ajouter une nouvelle architecture dans Miasm. J'ai essayé, mais j'ai vite été à court de cailloux. J'ai donc opté pour la manière sale. J'ai repris le squelette du désassembleur ubpf, mais en plus de lui faire sortir les instructions au format texte, j'en ai profité pour générer à la volée des expressions en langage intermédiaire Miasm. Quand il y a un jump, je saute directement à l'offset correspondant de mon blob binaire et je re-désassemble l'instruction.

Ainsi, on ne pourra pas profiter de fonctionnalités avancées type DependencyGraph, puisque je n'ai même pas géré la notion de Basic Block. Mais on pourra quand même bénéficier du moteur d'exécution symbolique et de ses simplifications.

Après quelques sérieux tâtonnements pour trouver une façon viable de représenter un paquet réseau dans Miasm (j'ai eu pas mal de soucis d'endianness), j'ai pu vérifier

8. https://github.com/cea-sec/miasm

le fonctionnement de mon exécution symbolique grâce au paquet contenant le LUM. L'exécution se termine correctement, le statut de la machine à états passe à 1 (*« Good job !, you find a LUM »*) et la valeur du LUM sert de base pour initialiser deux variables globales de 4 octets. Ces variables influenceront les vérifications de la clé.

```
@64[HASH_MAP_0] 0x1
@64[HASH_MAP_1] 0xBD89A8AE
@64[HASH_MAP_2] 0xBA9BBC8D
```

Ensuite, on relance l'exécution symbolique en utilisant les valeurs ci-dessus comme état initial. On simule ainsi le fait que le premier paquet contenant le LUM ait été correctement reçu par le serveur.

La clé à trouver est composée de 32 caractères hexa. Le code qui vérifie si la clé est correcte débute par une fonction decode_hex qui va convertir cette clé en 16 octets, afin de les manipuler. Problème, le sadisme des concepteurs du challenge fait que cette fonction decode_hex est dépliée sur plus de 1000 instructions, et que les caractères de la clé sont convertis dans un ordre chaotique. Il est donc très difficile de savoir s'il y a entourloupe ou pas, et si le décodage de la clé va se passer comme prévu.

L'exécution symbolique du code de decode_hex ne donne rien, car le flux d'exécution est beaucoup trop conditionnel. On va donc utiliser des mots qui me font peur. On va faire de l'exécution concolique [9] !.

## 3.4 Exécution concolique

L'idée est de lancer l'exécution symbolique avec une valeur concrète de la clé (ici 00112233445566778899AABBCCDDEEFF), puis de mettre un breakpoint juste avant le début du code qui vérifie la validité de chaque morceau de clé décodée. On retrouve quelle sous-partie de la clé est manipulée (par exemple 00112233), on remplace cette valeur concrète par une valeur symbolique, et on relance l'exécution symbolique.

La vérification de la clé se décompose en 4 parties, chaque partie vérifiant 4 octets. Des opérations vont être effectuées sur la clé, et le résultat final devra valoir « SSTIC CHALL 2017« . Pour la première partie (K1), l'exécution s'arrête par une exception, car le pointeur d'instruction ne peut pas être calculé :

```
Exception: Unable to compute PC : (((((K1^0x137E1FB9EB) << 0x20) >> 0x20) != 0x53535449)?(0x2648,0x2120))
```

Pour passer le premier test, il faut que K1 xor 0x7e1fb9eb fasse 0x53535449 ("SSTI"). La première partie de la clé est donc 0x2d4ceda2.

Pour la deuxième partie de la clé, l'expression symbolique est du même ordre :

9. Je ne pensais pas écrire ça un jour...

```
Exception: Unable to compute PC : ((((((K2^0xBA9BBC8D)+0x26E90C3) << 0x20) >> 0x20)
!= 0x43204348)?(0x2648,0x21D8))
```

K2 vaut donc 0xfa2a0e08. A noter que l'exécution symbolique a permis d'ignorer l'implication d'une multiplication et d'une division avec la chaîne « polymorf » [10] dans la vérification de K1 et K2.

Pour K3 et K4, les choses se corsent un peu. Le code assembleur qui vérifie ces sous-parties de la clé n'est pas très sexy. En voici un extrait.

```
2278 180200002da5468a  lddw  r2, 0x8a46a52d
2288 bf13000000000000  mov   r3, r1
2290 3f23000000000000  div   r3, r2
2298 2f23000000000000  mul   r3, r2
22A0 1f31000000000000  sub   r1, r3
22A8 2f11000000000000  mul   r1, r1
22B0 bf13000000000000  mov   r3, r1
22B8 3f23000000000000  div   r3, r2
22C0 2f23000000000000  mul   r3, r2
22C8 1f31000000000000  sub   r1, r3
22D0 bf83000000000000  mov   r3, r8
22D8 3f23000000000000  div   r3, r2
22E0 2f23000000000000  mul   r3, r2
22E8 1f38000000000000  sub   r8, r3
22F0 2f11000000000000  mul   r1, r1
22F8 bf13000000000000  mov   r3, r1
2300 3f23000000000000  div   r3, r2
2308 2f23000000000000  mul   r3, r2
2310 1f31000000000000  sub   r1, r3
2318 2f11000000000000  mul   r1, r1
2320 bf13000000000000  mov   r3, r1
2328 3f23000000000000  div   r3, r2
2330 2f23000000000000  mul   r3, r2
2338 1f31000000000000  sub   r1, r3
2340 2f11000000000000  mul   r1, r1
2348 bf13000000000000  mov   r3, r1
2350 3f23000000000000  div   r3, r2
2358 2f23000000000000  mul   r3, r2
2360 1f31000000000000  sub   r1, r3
2368 2f11000000000000  mul   r1, r1
2370 bf13000000000000  mov   r3, r1
2378 3f23000000000000  div   r3, r2
2380 2f23000000000000  mul   r3, r2
2388 1f31000000000000  sub   r1, r3
2390 2f11000000000000  mul   r1, r1
2398 bf13000000000000  mov   r3, r1
23A0 3f23000000000000  div   r3, r2
23A8 2f23000000000000  mul   r3, r2
23B0 1f31000000000000  sub   r1, r3
23B8 2f11000000000000  mul   r1, r1
23C0 bf13000000000000  mov   r3, r1
23C8 3f23000000000000  div   r3, r2
23D0 2f23000000000000  mul   r3, r2
23D8 1f31000000000000  sub   r1, r3
23E0 2f81000000000000  mul   r1, r8
23E8 bf13000000000000  mov   r3, r1
23F0 3f23000000000000  div   r3, r2
23F8 2f23000000000000  mul   r3, r2
2400 1f31000000000000  sub   r1, r3
2408 55014700414c4c20  jne   r1, 0x204c4c41, exit_bad_key
```

10. le pseudo d'un des concepteurs du challenge https://github.com/polymorf

En rajoutant plusieurs règles de simplification dans le moteur d'exécution symbolique de Miasm, on parvient à obtenir une expression concise du test effectué sur K3.

```
Exception: Unable to compute PC : (((((((K3^0xBD89A8AE) << 0x20) >> 0x20) ** 0x101) % 0x8A46A52D)
!= 0x204C4C41)?(0x2648,0x2410))
```

Il s'agit d'une exponentiation modulaire. Pour être plus précis, c'est un chiffrement RSA qui utilise comme module 0x8A46A52D et comme exposant public 257.

Pour retrouver la valeur de K3, on pourrait simplement bruteforcer 4 octets. Mais j'ai fait les choses proprement car j'avais déjà sous la main tous les outils nécessaires au cassage d'une clé RSA. Pas besoin d'algorithme avancé, une clé RSA avec un module de 32 bits se casse instantanément.

On factorise tout d'abord le module pour retrouver les deux nombres premiers P et Q qui le composent.

```
>>> n = 0x8A46A52D
>>> e = 0x101
>>> p, q = rsa_factorize(0x8A46A52D)
>>> p
43019
>>> q
53927
```

Puis, à partir de P, Q et de l'exposant public E, on peut calculer l'exposant privé D qui nous servira à déchiffrer (en utilisant l'algorithme d'Euclide étendu).

```
>>> d = rsa_private(e, p, q)
>>> d
713086789
>>> K3 = pow(0x204C4C41, d, n) ^ 0xBD89A8AE
>>> hex(K3)
'0xfc360b55'
```

Pour K4, on obtient un nouveau chiffrement RSA avec une clé différente.

```
Exception: Unable to compute PC : (((((((K4^0xBA9BBC8D) << 0x20) >> 0x20) ** 0x101) % 0xDC6C88E1)
== 0x37313032)?(0x26B0,0x2648))
```

En procédant de la même manière, on casse la clé RSA et on trouve que K4 vaut 0x291de7c9. Voici le code *en l'état* utilisé pour les différentes étapes de ce niveau.

```
1  import struct
2  import StringIO
```

```
import sys
import miasm2.expression.expression as m2
from miasm2.expression.simplifications import expr_simp
from miasm2.ir.symbexec import SymbolicExecutionEngine

Inst = struct.Struct("BBHI")

R0 = m2.ExprId("r0", 64)
R1 = m2.ExprId("r1", 64)
R2 = m2.ExprId("r2", 64)
R3 = m2.ExprId("r3", 64)
R4 = m2.ExprId("r4", 64)
R5 = m2.ExprId("r5", 64)
R6 = m2.ExprId("r6", 64)
R7 = m2.ExprId("r7", 64)
R8 = m2.ExprId("r8", 64)
R9 = m2.ExprId("r9", 64)
R10 = m2.ExprId("r10", 64)
PC = m2.ExprId("PC", 64)
PKT = m2.ExprId("PACKET", 64)
SCRATCH = m2.ExprId("SCRATCH", 64)

MEM = m2.ExprId("MEM", 64)
def HASH_MAP(i):
    return m2.ExprId("HASH_MAP_%i" % i, 64)

all_regs = [R0, R1, R2, R3, R4, R5, R6, R7, R7, R8, R9, R10]

CLASSES = {
    0: "ld",
    1: "ldx",
    2: "st",
    3: "stx",
    4: "alu",
    5: "jmp",
    7: "alu64",
}

ALU_OPCODES = {
    0: 'add',
    1: 'sub',
    2: 'mul',
    3: 'div',
    4: 'or',
    5: 'and',
    6: 'lsh',
    7: 'rsh',
    8: 'neg',
    9: 'mod',
    10: 'xor',
    11: 'mov',
    12: 'arsh',
    13: '(endian)',
}

ALU_OP = {
    "add": "+",
    "sub": "-",
    "lsh": "<<",
    "rsh": ">>",
    "and": "&",
    "or": "|",
    "xor": "^",
    "mul": "*",
    "div": "/",
}

JMP_OPCODES = {
    0: 'ja',
    1: 'jeq',
    2: 'jgt',
    3: 'jge',
    4: 'jset',
    5: 'jne',
```

```
    6: 'jsgt',
    7: 'jsge',
    8: 'call',
    9: 'exit',
}

JMP_OP = {
    "jeq": "==",
    "jne": "!=",
    "jgt": ">",
    "jge": ">=",
}

MODES = {
    0: 'imm',
    1: 'abs',
    2: 'ind',
    3: 'mem',
    6: 'xadd',
}

SIZES = {
    0: 'w',
    1: 'h',
    2: 'b',
    3: 'dw',
}

SIZE_NAMES = {
    'w': 4,
    'h': 2,
    'b': 1,
    'dw':8,
}

BPF_CLASS_LD = 0
BPF_CLASS_LDX = 1
BPF_CLASS_ST = 2
BPF_CLASS_STX = 3
BPF_CLASS_ALU = 4
BPF_CLASS_JMP = 5
BPF_CLASS_ALU64 = 7

def Rx(reg):
    return m2.ExprId("r%i" % reg, 64)

def Ix(imm, size=64):
    return m2.ExprInt(imm, size)

def Ox(off):
    if off <= 32767:
        return Ix(int("+" + str(off)))
    else:
        return Ix(int("-" + str(65536-off)))

def Jx(off):
    if off <= 32767:
        return Ix(8*(off+1))
    else:
        return Ix(8*(65536-off+1))

def Mx(reg, size, off=0):
    if off:
        return m2.ExprMem(Rx(reg) + Ix(off), size)
    return m2.ExprMem(Rx(reg), size)

def IDx(name, size=64):
    return m2.ExprId(name, size)

def Sx(s):
    return m2.ExprInt(int(s.encode("hex"), 16), len(s)*8)

def R(reg):
    return "r" + str(reg)
```

```
151
152 def I(imm):
153     return "%#x" % imm
154
155 def M(base, off):
156     if off != 0:
157         return "[%s%s]" % (base, O(off))
158     else:
159         return "[%s]" % base
160
161 def O(off):
162     if off <= 32767:
163         return "+" + str(off)
164     else:
165         return "-" + str(65536-off)
166
167
168 ######## INIT VALUES
169
170 init_reg = {r:SCRATCH for r in all_regs}
171 init_reg[R1] = PKT
172 init_reg[R10] = MEM
173 init_reg[PC] = Ix(0)
174
175 EtherType_IPv4 = "\x08\x00"
176 IP_VER_IHL = "\x45"
177 IP_TL = "\x00\x2C" # 44
178 UDP_PROTO = "\x11"
179 DST_PORT = "\x05\x39" # 1337
180
181 LUM_PACKET = "A"*12 + EtherType_IPv4 # Ethernet (14)
182 LUM_PACKET += IP_VER_IHL + "A" + IP_TL + "AAAAA" + UDP_PROTO + "A"*10  # IP (20)
183 LUM_PACKET += "AA" + DST_PORT + "AAAA" # UDP Header (8)
184 LUM_PACKET += "LUM{BvWQEdCrMfA}"# UDP Payload (16)
185
186 KEY_IP_TL = "\x00\x41"
187 UDP_LENGTH = "\x00\x2D"
188
189 KEY_PACKET = "A"*12 + EtherType_IPv4 # Ethernet (14)
190 KEY_PACKET += IP_VER_IHL + "A" + KEY_IP_TL + "AAAAA" + UDP_PROTO + "A"*10  # IP (20)
191 KEY_PACKET += "AA" + DST_PORT +  UDP_LENGTH + "AA" # UDP Header (8)
192 #KEY_PACKET += "KEY{00112233445566778899AABBCCDDEEFF}"# UDP Payload (37)
193 KEY_PACKET += "KEY{2D4CEDA2FA2A0E08FC360B55291DE7C9}"# UDP Payload (37)
194
195 LUM = False
196 if LUM:
197     PACKET = LUM_PACKET
198 else:
199     PACKET = KEY_PACKET
200     # KEY HASH MAP
201     init_reg[m2.ExprMem(HASH_MAP(0), 64)] = Ix(1)
202     init_reg[m2.ExprMem(HASH_MAP(1), 64)] = Ix(0xBD89A8AE)
203     init_reg[m2.ExprMem(HASH_MAP(2), 64)] = Ix(0xBA9BBC8D)
204
205 def symbexec_one(data, offset, symb):
206     code, regs, off, imm = Inst.unpack_from(data, offset)
207     dst_reg = regs & 0xf
208     src_reg = (regs >> 4) & 0xf
209     cls = code & 7
210     class_name = CLASSES[cls]
211     # ALU class
212     if cls in [BPF_CLASS_ALU, BPF_CLASS_ALU64]:
213         source = (code >> 3) & 1
214         opcode = (code >> 4) & 0xf
215         op = ALU_OPCODES[opcode]
216         if cls == BPF_CLASS_ALU:
217             op += "32"
218         if op == "mov":
219             if source == 0:
220                 return {Rx(dst_reg): Ix(imm)}, "%s %s, %s" % (op, R(dst_reg), I(imm))
221             return {Rx(dst_reg): Rx(src_reg)}, "%s %s, %s" % (op, R(dst_reg), R(src_reg))
222         elif op in ALU_OP:
223             if source == 0:
224                 return {Rx(dst_reg): m2.ExprOp(ALU_OP[op], Rx(dst_reg), Ix(imm & 0xFFFFFFFFFFFFFFFF))},
```

```
                        "%s %s, %s" % (op, R(dst_reg), I(imm))
            return {Rx(dst_reg): m2.ExprOp(ALU_OP[op], Rx(dst_reg), Rx(src_reg)& Ix(0xFFFFFFFFFFFFFFFF))},
                    "%s %s, %s" % (op, R(dst_reg), R(src_reg))
        else:
            raise Exception("Unknown ALU opcode %s" % op)
    # JMP class
    elif cls == BPF_CLASS_JMP:
        source = (code >> 3) & 1
        opcode = (code >> 4) & 0xf
        op = JMP_OPCODES[opcode]
        if op == "exit":
            return {}, op
        elif op == "ja":
            asm = "%s %s" % (op, O(off))
            return {PC:Ix(offset) + Jx(off)}, asm
        elif op in JMP_OP:
            if source == 0:
                asm = "%s %s, %s, %s" % (op, R(dst_reg), I(imm), O(off))
                e = {PC:m2.ExprCond(m2.ExprOp(JMP_OP[op], Rx(dst_reg), Ix(imm)), Ix(offset) + Jx(off), Ix(offset+8))}
                return e, asm
            else:
                asm = "%s %s, %s, %s" % (op, R(dst_reg), R(src_reg), O(off))
                e = {PC:m2.ExprCond(m2.ExprOp(JMP_OP[op], Rx(dst_reg), Rx(src_reg)), Ix(offset) + Jx(off), Ix(offset+8))}
                return e, asm
        elif op == "call":
            expr = {R0: m2.ExprOp("call", Ix(imm))}
            asm = "%s %s" % (op, I(imm))
            return expr, asm
        else:
            raise Exception("Unknown JMP opcode %s" % op)
    # LOAD/STORE class
    elif cls in [BPF_CLASS_LD, BPF_CLASS_LDX, BPF_CLASS_ST, BPF_CLASS_STX]:
        size_id = (code >> 3) & 3
        mode = (code >> 5) & 7
        mode_name = MODES[mode]
        size_name = SIZES[size_id]
        size = SIZE_NAMES[size_name]
        if cls == BPF_CLASS_LD and mode_name == "abs":
            pkt = symb.eval_expr(m2.ExprMem(R6+Ix(imm), size*8))
            if isinstance(pkt, m2.ExprInt):
                pkt = Ix(pkt.arg.arg)
            expr = {R0: pkt}
            scratch_registers(symb)
            asm = "%s %s, %s_packet[%s]" % (class_name + size_name, R(0), R(6), I(imm))
            return expr, asm
        elif cls == BPF_CLASS_LD and mode_name == "ind":
            p_abs =  symb.eval_expr(Rx(src_reg) + Ix(imm))
            pkt = symb.eval_expr(m2.ExprMem(R6 + p_abs, size*8))
            if isinstance(pkt, m2.ExprInt):
                pkt = Ix(pkt.arg.arg)
            expr = {R0: pkt}
            scratch_registers(symb)
            asm = "%s %s, %s_packet[%s+0x%x]" % (class_name + size_name, R(0), R(6), R(src_reg), imm)
            return expr, asm
        elif code == 0x18: # lddw
            _, _, _, imm2 = Inst.unpack_from(data, offset+8)
            imm = (imm2 << 32) | imm
            expr = {Rx(dst_reg): Ix(imm), PC: PC+Ix(16)}
            asm = "%s %s, %s" % (class_name + size_name, R(dst_reg), I(imm))
            return expr, asm
        elif cls == BPF_CLASS_LDX:
            asm = "%s %s, %s" % (class_name + size_name, R(dst_reg), M(R(src_reg), off))
            return {Rx(dst_reg): Mx(src_reg, size*8, off)}, asm
        elif cls == BPF_CLASS_STX:
            asm = "%s %s, %s" % (class_name + size_name, M(R(dst_reg), off), R(src_reg))
            return {Mx(dst_reg, size*8, off): Rx(src_reg)}, asm
        else:
            raise Exception("Unknown LOAD/STORE opcode %s" % hex(code))
    else:
        raise Exception("Unknown opcode %s" % op)


def ebpf_symbexec(data, offset = 0):
    BP0 = BP1 = BP2 = BP3 = False
```

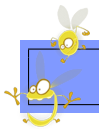
```
299     symb = SymbolicExecutionEngine(None, init_reg)
300     asm = ""
301     step = False
302     while offset < len(data) and asm != "exit":
303         print "%04X" % offset,
304         expr, asm = symbexec_one(data, offset, symb)
305         print asm
306         if PC not in expr:
307             expr[PC] = Ix(offset+8)
308         expr.update({k:v.zeroExtend(64) for k,v in expr.iteritems() if k in all_regs})
309         for k, v in expr.iteritems():
310             expr[k] = expr_simp(symb.eval_expr(v))
311         symb.eval_ir(expr)
312         if asm.startswith("call"):
313             resolve_bpf_call(symb)
314         print_id(symb)
315         print_mem(symb)
316         print "------"
317         if step:
318             raw_input()
319 
320         offset = compute_pc(symb)
321 
322         # KEY [0:32] default (00112233)
323         if BP0 and offset == 0x2000:
324             print "BP0: Will replace r5 by K1"
325             raw_input()
326             symb.symbols.symbols_id[R5] = IDx("K1", 64)
327 
328         # KEY [32:64] (default 44556677)
329         if BP1 and offset == 0x2198:
330             print "BP1: Will replace r2 by K2"
331             raw_input()
332             symb.symbols.symbols_id[R2] = IDx("K2", 64)
333 
334         # KEY [64:96] (default 8899AABB)
335         if BP2 and offset == 0x2248:
336             print "BP2: Will replace r8 by K3"
337             raw_input()
338             symb.symbols.symbols_id[R8] = IDx("K3", 64)
339 
340         # KEY [96:128] (default CCDDEEFF)
341         if BP3 and offset == 0x2480:
342             print "BP3: Will replace r4 by K4"
343             raw_input()
344             symb.symbols.symbols_id[R4] = IDx("K4", 64)
345 
346         if offset == 0x2648:
347             print "Bad key"
348             break
349 
350         if offset == 0x26B0:
351             print "Good key"
352             break
353 
354 def print_id(symb):
355     ids = symb.symbols.symbols_id.keys()
356     ids.sort()
357     for i in ids:
358         if i in init_reg and i in symb.symbols.symbols_id and symb.symbols.symbols_id[i] == init_reg[i]:
359             continue
360         print i, symb.symbols.symbols_id[i]
361 
362 def print_mem(symb):
363     mems = symb.symbols.symbols_mem.values()
364     mems.sort()
365     for m, v in mems:
366         if isinstance(v, m2.ExprInt):
367             sv = hex(v.arg.arg)[2:].strip("L")
368             if len(sv) % 2 == 1:
369                 sv = "0"+sv
370             print m, v, " # %r" % sv.decode("hex")
371         else:
372             print "oops", m, v
```

```
373
374
375
376 ### SIMPLIFICATIONS
377
378 def simp_packet(expr_simp, expr):
379     "BPF input packet value"
380     if not isinstance(expr.arg, m2.ExprOp):
381         return expr
382     if expr.arg.op != "+" or len(expr.arg.args) != 2 or expr.arg.args[0] != PKT:
383         return expr
384     i = expr.arg.args[1]
385     if not isinstance(i, m2.ExprInt):
386         return expr
387     i = i.arg.arg
388     s = expr.size / 8
389     e = m2.ExprInt(int(PACKET[i:i+s].encode("hex"), 16), expr.size)
390     return e
391
392 def simp_mem(expr_simp, expr):
393     if not isinstance(expr.arg, m2.ExprOp) or expr.arg.op != "+":
394         return expr
395     if len(expr.arg.args) != 2:
396         return expr
397     m, i = expr.arg.args
398     if m != MEM or not isinstance(i, m2.ExprInt):
399         return expr
400     i = i.arg.arg
401     print hex(i)
402     if i > 0xFFFF:
403         return expr_simp(m2.ExprOp("+", m, m2.ExprInt(i & 0xFFFF, 64)))
404     return expr
405
406 def simp_comp_op_and(expr_simp, expr):
407     if expr.op != "&":
408         return expr
409     if len(expr.args) != 2:
410         return expr
411     if not isinstance(expr.args[-1], m2.ExprInt):
412         return expr
413     i = expr.args[-1]
414     if not isinstance(expr.args[0], m2.ExprOp):
415         return expr
416     args = []
417     for a in expr.args[0].args:
418         args.append(expr_simp(a & i))
419     return expr_simp(m2.ExprOp(expr.args[0].op, *args))
420
421 # (A/B)*B ==> A - (A % B)
422 def simp_mul_to_mod(expr_simp, expr):
423     if expr.op != "*" or not isinstance(expr.args[0], m2.ExprOp):
424         return expr
425     o = expr.args[0]
426     if o.op != "/" or len(o.args) != 2:
427         return expr
428     A, B = o.args
429     return expr_simp(A - (A % B))
430
431
432 # ((A % B) * C) % B ==> (A*C) % B
433 # (C * (A % B)) % B ==> (A*C) % B
434 # ((A % B) ** C) % B ==> (A**C) % B
435 def simp_divmod(expr_simp, expr):
436     if expr.op != "%" or len(expr.args) != 2:
437         return expr
438     e, B = expr.args
439     if not isinstance(e, m2.ExprOp) or e.op not in ["*", "**"] or len(e.args) != 2:
440         return expr
441     o = e.op
442     for e, C in [e.args, e.args[::-1]]:
443         if isinstance(e, m2.ExprOp) and e.op == "%" and len(e.args) == 2 and e.args[1] == B:
444             A = e.args[0]
445             return expr_simp(m2.ExprOp(o, A, C) % B)
446     return expr
```

```
447
448 # A*A ==> A**2
449 def simp_mul_pow(expr_simp, expr):
450     if expr.op != "*" or len(expr.args) != 2 or expr.args[0] != expr.args[1]:
451         return expr
452     e = expr.args[0]
453     return expr_simp(m2.ExprOp("**", expr.args[0], Ix(2, expr.args[0].size)))
454
455 # (A**B) * A ==> A**(B+1)
456 def simp_mul_pow2(expr_simp, expr):
457     if expr.op != "*" or len(expr.args) != 2:
458         return expr
459     e, A = expr.args
460     if not isinstance(e, m2.ExprOp) or e.op != "**" or len(e.args) != 2:
461         return expr
462     e, B = e.args
463     if e != A:
464         return expr
465     return expr_simp(m2.ExprOp("**", A, B+Ix(1, B.size)))
466
467
468
469 # (A**B)**C ==> A**(B*C)
470 def simp_pow_pow(expr_simp, expr):
471     if expr.op != "**" or len(expr.args) != 2:
472         return expr
473     e, C = expr.args
474     if not isinstance(e, m2.ExprOp) or e.op != "**" or len(e.args) != 2:
475         return expr
476     A, B = e.args
477     return expr_simp(m2.ExprOp("**", A, B*C))
478
479
480 def resolve_bpf_call(symb):
481     MAP_LOOKUP = 1
482     MAP_UPDATE = 2
483     call = symb.symbols.symbols_id[R0]
484     if not isinstance(call, m2.ExprOp) or call.op != "call":
485         return
486     if not len(call.args) == 1 or not isinstance(call.args[0], m2.ExprInt):
487         return
488     func = call.args[0].arg.arg
489     if func == MAP_LOOKUP:
490         r2 = dereference_mem_int(symb, R2)
491         if r2 is not None:
492             print "Lookup of HASH_MAP[%i]" % r2
493             if isinstance(expr_simp(symb.eval_expr(m2.ExprMem(HASH_MAP(r2), 64))), m2.ExprInt):
494                 r0 = HASH_MAP(r2)
495             else:
496                 r0 = Ix(0)
497             symb.symbols.symbols_id[R0] = r0
498             scratch_registers(symb)
499     elif func == MAP_UPDATE:
500         r2 = dereference_mem_int(symb, R2)
501         r3 = dereference_mem_int(symb, R3)
502         if r2 is not None and r3 is not None:
503             symb.eval_ir({m2.ExprMem(HASH_MAP(r2), 64): Ix(r3), R0: SCRATCH})
504             print "Update HASH_MAP[%i] = %i" % (r2, r3)
505             scratch_registers(symb)
506
507
508 def dereference_mem_int(symb, reg):
509     r = expr_simp(m2.ExprMem(symb.symbols.symbols_id[reg], 64))
510     r = symb.eval_expr(m2.ExprMem(r, 64))
511     if isinstance(r, m2.ExprInt):
512         return r.arg.arg
513
514 def compute_pc(symb):
515     r = None
516     pc = expr_simp(symb.symbols.symbols_id[PC])
517     if isinstance(pc, m2.ExprInt):
518         return pc.arg.arg
519     elif isinstance(pc, m2.ExprCond) and isinstance(pc.cond, m2.ExprOp) and len(pc.cond.args) == 2:
520         o = pc.cond
```

```
521             if o.op in ["==", "!=", ">"]:
522                 if all(isinstance(o.args[x], m2.ExprInt) for x in [0,1]):
523                     a, b = o.args[0].arg.arg, o.args[1].arg.arg
524                     if ((o.op == "==" and a == b) or
525                         (o.op == "!=" and a != b) or
526                         (o.op == ">" and a > b)):
527                         r = pc.src1
528                     else:
529                         r = pc.src2
530             if isinstance(o.args[0], m2.ExprId) and o.args[0].name.startswith("HASH_MAP"):
531                 if isinstance(o.args[1], m2.ExprInt) and o.args[1].arg.arg == 0:
532                     if o.op == "==":
533                         r = pc.src2
534                     elif o.op == "!=":
535                         r = pc.src1
536         if r:
537             symb.symbols.symbols_id[PC] = r
538             return compute_pc(symb)
539         raise Exception("Unable to compute PC : %s" % pc)
540 
541 def scratch_registers(symb):
542     symb.eval_ir({r:SCRATCH for r in [R1, R2, R3, R4, R5]})
543 
544 # Enable pass
545 expr_simp.enable_passes({m2.ExprMem: [simp_packet]})
546 expr_simp.enable_passes({m2.ExprMem: [simp_mem]})
547 expr_simp.enable_passes({m2.ExprOp: [simp_comp_op_and]})
548 expr_simp.enable_passes({m2.ExprOp: [simp_mul_to_mod]})
549 expr_simp.enable_passes({m2.ExprOp: [simp_mul_pow]})
550 expr_simp.enable_passes({m2.ExprOp: [simp_mul_pow2]})
551 expr_simp.enable_passes({m2.ExprOp: [simp_divmod]})
552 expr_simp.enable_passes({m2.ExprOp: [simp_pow_pow]})
553 
554 if __name__ == '__main__':
555     import sys
556     if len(sys.argv) != 2:
557         print "Usage: miasm_ebpf.py [file]"
558         sys.exit(2)
559     with open(sys.argv[1]) as f:
560         ebpf_symbexec(f.read())
```

## 3.5 Résumé des secrets



LUM{BvWQEdCrMfA}

2d4ceda2fa2a0e08fc360b55291de7c9

# 4 RISCy zones

## 4.1 Présentation

```
$ ls /challenges/riscy_zones

TA.elf.signed
password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted
secret.lzma.encrypted
trustzone_decrypt
```

trustzone_decrypt est un binaire pour l'architecture Openrisc capable de déchiffrer des fichiers ayant un format propriétaire. Un fichier de test, dont le mot de passe est fourni, nous permet de jouer un petit peu. Le but de l'épreuve va être de réussir à déchiffrer le fichier secret.lzma.encrypted, fichier dont on ne connaît pas la clé de chiffrement.

Pour procéder au déchiffrement, le binaire trustzone_decrypt va utiliser un ioctl sur le device /dev/sstic du kernel Openrisc qui va conduire au chargement dans une trustzone du binaire TA.elf.signed, qui lui est un ELF utilisant l'architecture Risc-V. Ce qui veut dire, j'en ai bien peur, qu'il doit y avoir un VM Risc-V implémentée dans le Kernel linux Openrisc. Hum. La dernière chose dont j'ai envie est d'aller regarder dans le Kernel vmlinuz.bin avec Objdump...

Certaines étapes du déchiffrement sont gérées directement par trustzone_decrypt, d'autres sont déléguées à TA.elf.signed, et d'autres sont re-délégées par cette trusted app au mistérieux TrustedOs, qui réside probablement dans le kernel linux. Quand on déchiffre le fichier de test ou qu'on regarde les chaines de caractères des binaires, on s'aperçoit qu'il y a beaucoup de clés différentes qui sont utilisées :

- TA_HMAC_KEY
- SSTIC_AES_KEY
- SSTIC_CUSTOM_KEY
- SSTIC_PASSWORD_HMAC_KEY

Voilà qui n'est pas fort motivant.

## 4.2 Analyse statique du pauvre

J'ai commencé par analyser les binaires trustzone_decrypt et TA.elf.signed de manière statique. Évidemment, les outils qui permettent d'étudier ces architectures exotiques ne courent pas les rues. Pour le binaire Openrisc, on peut directement utiliser la VM Linux javascript du challenge, puisqu'Objdump est installé. Pour TA.elf.signed, par contre, il a fallu builder les riscv-tools [11], ce qui fut particulièrement pénible mais qui a fini par fonctionner. J'ai ainsi pu récupérer une toolchain très fournie, bien qu'au final je n'ai utilisé que le Objdump riscv.

Reverser des binaires avec Objdump, ce n'est pas la panacée. Pour retrouver un semblant d'ergonomie, j'ai fait un petit script qui améliore la sortie texte d'Objdump afin de rajouter en commentaire le contenu des données quand une adresse pointe vers la section .data.

11. https://github.com/riscv/riscv-tools

```python
import re

def get_string(off, data, data_base):
    o = int(off,16) - data_base
    if not (0 <= o < len(data)):
        return
    end = data.index("\x00", o)
    return repr(data[o:end])

def openrisc_clochard():
    data_base = 0x3408
    with open("tz_decrypt_dump.txt") as f:
        text = f.read()
    with open("tz_rodata.bin") as f:
        data = f.read()
    for s, off in re.findall(r"(l\.ori r.+,0x(3[4-7][0-9a-f]{2}))\n", text):
        string = get_string(off, data, data_base)
        print off, string
        text = text.replace(s, s + "  #%s" % string)
    return text

def riscv_clochard():
    data_base = 0x10054
    with open("ta_dump.txt") as f:
        text = f.read()
    with open("ta_rodata.bin") as f:
        data = f.read()
    for s, off in re.findall(r"(#\s+(10[0-3][0-9a-f]{2}) <[\w_]+-\w+>)", text):
        string = get_string(off, data, data_base)
        print off, string
        text = text.replace(s, "# %s: %s" % (off, string))
    return text

if __name__ == '__main__':
    with open("/tmp/tz_text.txt", "wb") as f:
        f.write(openrisc_clochard())
    with open("/tmp/ta_text.txt", "wb") as f:
        f.write(riscv_clochard())
```

```
[...]
   11b9c:    09050513        addi    a0,a0,144 # 10090: '[DEBUG] CMD CMD_TA_INIT\r\n'
   11ba0:    9bdff0ef        jal     ra,1155c <TEE_HMAC+0x164>
   11ba4:    000105b7        lui     a1,0x10
   11ba8:    00010537        lui     a0,0x10
   11bac:    0ac58593        addi    a1,a1,172 # 100ac: '___SSTIC_2017___'
   11bb0:    00000693        li      a3,0
   11bb4:    01000613        li      a2,16
   11bb8:    0c050513        addi    a0,a0,192 # 100c0: 'SSTIC_AES_KEY'
   11bbc:    825ff0ef        jal     ra,113e0 <TEE_writekey>
[...]
```

Quel luxe. On égale presque l'expérience utilisateur de Miasm2... :)

## 4.3 IOCTL et Python

Par acquis de conscience, je constate que si on change un octet du binaire TA.elf.signed, l'application refuse de se charger à cause d'une mauvaise signature. On n'a donc pas la main sur le code Riscv qui tournera dans la trustzone, mais on peut explorer les services non-privilégiés proposés par la Trusted App, en s'inspirant des ioctl de trustzone_decrypt.

Python est installé sur la VM Openrisc, et on dispose justement d'un exemple d'utilisation d'ioctl vers /dev/sstic et le TrustedOs. En effet, le script tee_client.py (dans /challenges/tools) est une bibliothèque nous permettant de communiquer en Python avec le TrustedOs. Ce script est utilisé par les outils add_key et add_lum qui permettent de valider un LUM ou une clé.

L'idée est donc de recoder l'équivalent de trustzone_decrypt en Python, d'une part pour bien comprendre ce que fait ce binaire, d'autre part car il sera forcement nécessaire de faire pas mal de bricolage avec les ioctl vers /dev/sstic, et que je préfère mille fois le faire en Python plutôt que de devoir cross-compiler du C pour Openrisc...

La bibliothèque tee_client.py utilise ctypes pour définir les structures des différents messages supportés par /dev/sstic. On s'en inspire donc fortement pour ajouter le support des commandes implémentées par TA.elf.signed. Attention cependant à ne pas s'écharper avec les c_char_p ! En effet, le champ `data_out` était initialement de type c_char_p, mais cela ne permet pas de récupérer des réponses contenant des octets nuls. Voici mon contournement permettant d'envoyer des messages à la Trusted App.

```python
class custom_tee_message(ctypes.Structure):
    CMD_GET_VERSION = 0x0001
    CMD_LOAD_TA     = 0x0002
    CMD_TA_MESSAGE  = 0x0003
    CMD_UNLOAD_TA   = 0x0004
    CMD_CHECK_LUM   = 0x0005
    CMD_CHECK_KEY   = 0x0006
    MAX_LEN         = 8192
    _fields_ = [
        ('cmd', ctypes.c_int),
        ('data_in_len', ctypes.c_int),
        ('data_in', ctypes.c_char_p),
        ('data_out_len', ctypes.c_int),
        ('data_out', ctypes.c_int),
    ]

def ta_message(cmd):
    msg = custom_tee_message()
    msg.cmd = tee.tee_message.CMD_TA_MESSAGE
    msg.data_in_len = 280
    msg.data_in = ctypes.c_char_p(cmd + "\x00"*(280 - len(cmd)))
    msg.data_out_len = 260
    data_out = ctypes.create_string_buffer(260)
    msg.data_out = ctypes.addressof(data_out)

    devicehandle = open("/dev/sstic", "r")
    try:
        fcntl.ioctl(devicehandle, 0xc0145300, msg)
    except Exception as e:
        print e
    finally:
        devicehandle.close()
        r = data_out.raw
        return r[:2], r[2:4], r[4:].strip("\x00")
```

La Trusted App supporte 5 commandes :

**0 : TA_INIT** Enregistrement de plusieurs clés dans l'environnement de confiance.

**1 : GET_TA_VERSION** Récupération de la version de la Trusted App.

**2 : CHECK_PASSWORD** Vérification de la validité du mot de passe de déchiffrement, en calculant un HMAC qu'on compare avec celui stocké au début du fichier chiffré. Le header du fichier étant lui-même chiffré en AES avec une clé connue.

**3 : DECRYPT_BLOCK** Prend en entrée un compteur de 4 octets et un bloc de 16 octets à chiffrer ou déchiffrer. Le compteur sert à dériver le mot de passe, à l'aide d'un algorithme maison. La clé dérivée est ensuite simplement xorée avec le bloc de 16 octets.

**4 : GET_TA_LUM** En envoyant la commande ta_message("\x04"), on récupère un LUM gratuit : LUM{gdN8.D*@+UV}.

En jouant avec le fichier chiffré de test et notre implémentation python de trustzone_decrypt, on constate que les problèmes cryptographiques ne manquent pas.

– Un premier regard laisse penser que le déchiffrement est implémenté en mode CBC. C'est du bluff, car chaque bloc est xoré avec le bloc chiffré précédent (et non le bloc déchiffré). Le résultat obtenu est donc plus proche de l'ECB.

– Le compteur cycle tous les 32 tours. Pour un mot de passe, il n'y aura donc que 32 clés dérivées possibles. Si on parvient à retrouver un clair connu sur un des blocs de 16 octets, il sera donc possible de déchiffrer tous les autres blocs qui utilisent la même clé dérivée (donc 1/32eme du fichier).

– On dispose d'un clair connu, puisque le fichier secret.lzma.encrypted aura un header lzma. La spécification lzma que j'ai trouvé n'était pas extraordinairement claire, mais de manière empirique en compressant plusieurs types de fichiers de plusieurs tailles, on peut espérer un header connu de 14 octets, il ne reste donc que 2 octets à bruteforcer pour casser le premier bloc.

Seulement, tout ça n'est pas encore suffisant pour déchiffrer le fichier secret.lzma.encrypted. Le but du jeu semble être d'inverser l'algorithme de dérivation de clé, car puisque nous disposons d'un clair connu de 14 octets, si l'algorithme est inversible, alors on pourra directement en déduire le mot de passe.

Cet algorithme de dérivation se présente sous la forme d'un bloc séquentiel d'environ 150 instructions riscv. Comme on ne change pas une équipe qui gagne, je m'empresse d'aller récupérer mon sceau de cailloux avant que ça cicatrise.

## 4.4 Le retour de l'exécution symbolique

Me voilà donc parti pour faire de l'exécution symbolique d'instructions Riscv dans Miasm. Comme pour eBPF, le support de l'architecture Riscv dans Miasm a été fait au minimum, (d'autant plus qu'il n'y a ici ni boucle ni code conditionnel). Cette fois, j'ai parsé la sortie texte d'objdump pour générer à la volée des expressions pour le langage intermédiaire de Miasm.

On extrait d'abord l'algorithme de dérivation dans un fichier ta_code.txt

```
lw s1,4(s0)
lw a3,68(sp)
lw t5,76(sp)
lui a5,0xadaab
lui t0,0x52555
addi s2,a5,-1622
addi t0,t0,1621
addi a4,s1,23
and a2,t5,s2
and a5,a3,t0
andi a4,a4,31
or a5,a5,a2
neg a2,a4
[...]
```

Après exécution symbolique, on obtient des expressions pas très sympa, que je ne parviens pas spécialement à simplifier ni à inverser.

C'est pour moi l'occasion d'expérimenter pour la première fois une des fonctionnalités de Miasm : les bindings vers le SMT solver z3. Et je dois reconnaître que ça a étonnement bien fonctionné. En m'inspirant d'un exemple présent sur le blog de Miasm [12], j'ai pu convertir mon expression Miasm en contraintes z3.

Voici le script qui génère les 64k mots de passes possibles permettant d'obtenir un header lzma valide. Il a nécessité environ 3 heures d'exécution sur un laptop.

```python
import sys
import re
import z3
import miasm2.expression.expression as m2
from miasm2.expression.simplifications import expr_simp
from miasm2.ir.symbexec import SymbolicExecutionEngine
from miasm2.ir.translators import Translator

def iter_instr():
    with open("ta_code.txt") as f:
        data = f.read()
    for line in data.split("\n"):
        l = line.strip().split(" ")
        op = l[0]
        args = []
        if len(l) > 1:
            args = l[1].split(",")
        yield op, args, line

def Rx(reg):
    return m2.ExprId(reg, 32)

def Ix(i, size=32):
    if isinstance(i, str):
        i = eval(i)
    return m2.ExprInt(i, size)

def Dx(s, size=32):
    return m2.ExprId(s, size)

def Mx(reg, offset=0, size=32):
    if offset:
        return m2.ExprMem(Rx(reg) + Ix(offset), size)
    return m2.ExprMem(Rx(reg), size)

```

12. http://www.miasm.re/blog/2016/03/24/re150_rebuild.html

```
36  def Str2Mx(s, size=32):
37      """ 4(s0) ==> Mx(s0 + 4) """
38      s = s.strip()
39      r = re.findall(r"^(\d+)\(((\w+)\)$", s)
40      if len(r) == 1:
41          return Mx(r[0][1], int(r[0][0]), size)
42      raise Exception("Unable to parse mem access %s" % s)
43
44  def symbexec_one(op, args, symb):
45      if op == "andi":
46          return {Rx(args[0]): Rx(args[1]) & Ix(args[2])}
47      elif op == "and":
48          return {Rx(args[0]): Rx(args[1]) & Rx(args[2])}
49      elif op == "addi":
50          return {Rx(args[0]): Rx(args[1]) + Ix(args[2])}
51      elif op == "add":
52          return {Rx(args[0]): Rx(args[1]) + Rx(args[2])}
53      elif op == "or":
54          return {Rx(args[0]): Rx(args[1]) | Rx(args[2])}
55      elif op == "xori":
56          return {Rx(args[0]): Rx(args[1]) ^ Ix(args[2])}
57      elif op == "neg":
58          c = (Ix(0) - Rx(args[1])) & Ix(0xFF)
59          return {Rx(args[0]): c}
60      elif op == "srli":
61          return {Rx(args[0]): Rx(args[1]) >> Ix(args[2])}
62      elif op == "srl":
63          return {Rx(args[0]): Rx(args[1]) >> (Rx(args[2])&Ix(31))}
64      elif op == "sll":
65          return {Rx(args[0]): Rx(args[1]) << (Rx(args[2])&Ix(31))}
66      elif op == "sltu":
67          cond = m2.ExprCond(m2.ExprOp("<", Rx(args[1]), Rx(args[2])), Ix(0), Ix(1))
68          return {Rx(args[0]): cond}
69      elif op == "lw":
70          return {Rx(args[0]): Str2Mx(args[1])}
71      elif op == "lui":
72          return {Rx(args[0]): Ix(args[1]) << Ix(12) }
73      elif op == "sb":
74          return {Str2Mx(args[1], 8): Rx(args[0])[0:8]}
75      else:
76          print "Unknown opcode", op
77
78  def dump_id(symb):
79      ids = symb.symbols.symbols_id.keys()
80      ids.sort()
81      for expr in ids:
82          if (expr in regs_init and expr in symb.symbols.symbols_id and symb.symbols.symbols_id[expr] == regs_init[expr]):
83              continue
84          print expr, "=", symb.symbols.symbols_id[expr]
85      print "-"*15 + "\n"
86
87  def xor(a, b):
88      r = []
89      for i in range(len(a)):
90          r.append(chr(ord(a[i]) ^ ord(b[i])))
91      return "".join(r)
92
93  def simp_compose_op_int(expr_simp, expr):
94      if expr.op not in "&|^" or len(expr.args) != 2:
95          return expr
96      A, i = expr.args
97      if not isinstance(A, m2.ExprCompose) or not isinstance(i, m2.ExprInt):
98          return expr
99      i = i.arg.arg
100     new_args = []
101     start = 0
102     for e in A.args:
103         int_part = (i >> start) & (2**e.size-1)
104         new_args.append(m2.ExprOp(expr.op, e, Ix(int_part, e.size)))
105         start += e.size
106     return expr_simp(m2.ExprCompose(*new_args))
107
108 def simp_compose_op_compose(expr_simp, expr):
109     if expr.op not in "&|^" or len(expr.args) != 2:
```

```
110         return expr
111     A, B = expr.args
112     if not all(isinstance(e, m2.ExprCompose) for e in expr.args):
113         return expr
114     new_args = []
115     for i in range(expr.size):
116         new_args.append(m2.ExprOp(expr.op, A[i:i+1], B[i:i+1]))
117     return expr_simp(m2.ExprCompose(*new_args))
118 
119 def model2dword(m, prefix):
120     dword = [None]*32
121     for k in m:
122         if str(k).startswith(prefix):
123             i = int(str(k).strip(prefix))
124             dword[i] = str(m[k])
125     return "%08X" % int("".join(dword)[::-1], 2)
126 
127 def iter_models(z, nb=None):
128     count = 0
129     while z.check() == z3.sat:
130         m = z.model()
131         yield m
132         block = []
133         for d in m:
134             c = d()
135             block.append(c != m[d])
136         z.add(z3.Or(block))
137         count += 1
138         if nb is not None and count >= nb:
139             break
140 
141 regs_init= {
142    Mx("sp", 64): m2.ExprCompose(*[Dx("a%i" % i, 1) for i in range(32)]),
143    Mx("sp", 68): m2.ExprCompose(*[Dx("b%i" % i, 1) for i in range(32)]),
144    Mx("sp", 72): m2.ExprCompose(*[Dx("c%i" % i, 1) for i in range(32)]),
145    Mx("sp", 76): m2.ExprCompose(*[Dx("d%i" % i, 1) for i in range(32)]),
146    Mx("s0", 4): Ix(0),
147 }
148 
149 expr_simp.enable_passes({m2.ExprOp: [simp_compose_op_int]})
150 expr_simp.enable_passes({m2.ExprOp: [simp_compose_op_compose]})
151 
152 trans = Translator.to_language("z3")
153 
154 def main():
155     symb = SymbolicExecutionEngine(None, regs_init)
156     for op, args, raw in iter_instr():
157         print raw
158         expr = symbexec_one(op, args, symb)
159         if expr is None:
160             break
161         for k, v in expr.iteritems():
162             expr[k] = expr_simp(v)
163         symb.eval_ir(expr)
164         #symb.dump_mem()
165         #dump_id(symb)
166 
167     iv = "39962d55d138efefc6e4a14d5cd7352e".decode("hex")
168     block0 = "88db9a9733655cbf063b590244f075db".decode("hex")
169     lzma_header = "5d00008000ffffffffffffffff000000".decode("hex")
170     # Two last bytes of lzma header will be bruteforced
171 
172     key = xor(xor(iv[::-1], lzma_header), block0)
173     print "Target derivated key", key.encode("hex")
174 
175     # Init z3 with the 14 known key bytes
176     z = z3.Solver()
177     for count, i in enumerate(range(64, 80)):
178         a = symb.eval_expr(Mx("sp", i, 8))
179         e = m2.ExprAff(m2.ExprInt(ord(key[count]), 8), a)
180         if not isinstance(a, m2.ExprInt) and count < 14:
181             z.add(trans.from_expr(e))
182 
183     # Compute the 64k possible passwords
```

```
184     with open("passwords.txt", "wb") as f:
185         c = 0
186         for m in iter_models(z):
187             r = ""
188             print c
189             for prefix in "abcd":
190                 dword = model2dword(m, prefix)
191                 r += dword.decode("hex")[::-1].encode("hex").upper()
192             c += 1
193             f.write(r+"\n")
194     print "Win?!"
195 
196 if __name__ == '__main__':
197     main()
```

Pour savoir lequel de nos 64k mots de passe est correct, on déplace notre fichier sur la machine virtuelle Openrisc, et on va utiliser la commande 0x2 CHECK_PASSWORD de la Trusted App sur chaque candidat. Le verdict tombe en moins d'une minute : KEY = 5921cd9fd3a82bd9244ece5328c6c95f.

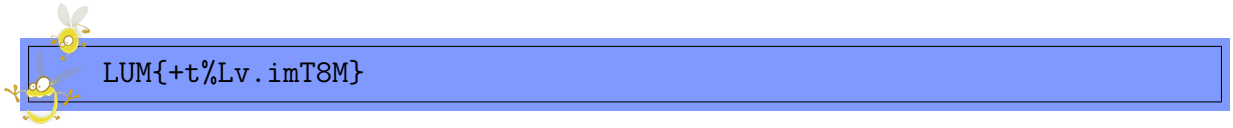
Ce mot de passe (qui est en fait la clé de validation du niveau) permet de déchiffrer le fichier secret.lzma.encrypted. On obtient une image qui nous indique que le niveau est terminé.

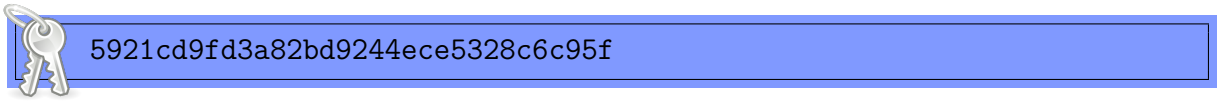

```
$ exif secret

Marqueurs EXIF dans'secret' (ordre des octets 'Intel') :
--------------------+------------------------------------------------------------
Marqueur            |Valeur
--------------------+------------------------------------------------------------
Description de l'ima|Congratulations : YHZ{+g%Yi.vzG8Z}
X-Resolution        |72
Y-Resolution        |72
Unite de la resoluti|pouces
Version d'exif      |Version d'exif 2.1
FlashPixVersion     |FlashPix Version 1.0
Espace des couleurs |Internal error (unknown value 65535)
--------------------+------------------------------------------------------------
```

Les métadonnées exif de l'image contiennent un LUM encodé en rot13, qui vaut donc LUM+t%Lv.imT8M.

Il me manque un LUM ! Je l'ai cherché un petit moment, mais il est resté bien caché ce coquinou. Tant pis !

## 4.5 Résumé des secrets


# 5 Unstable machines

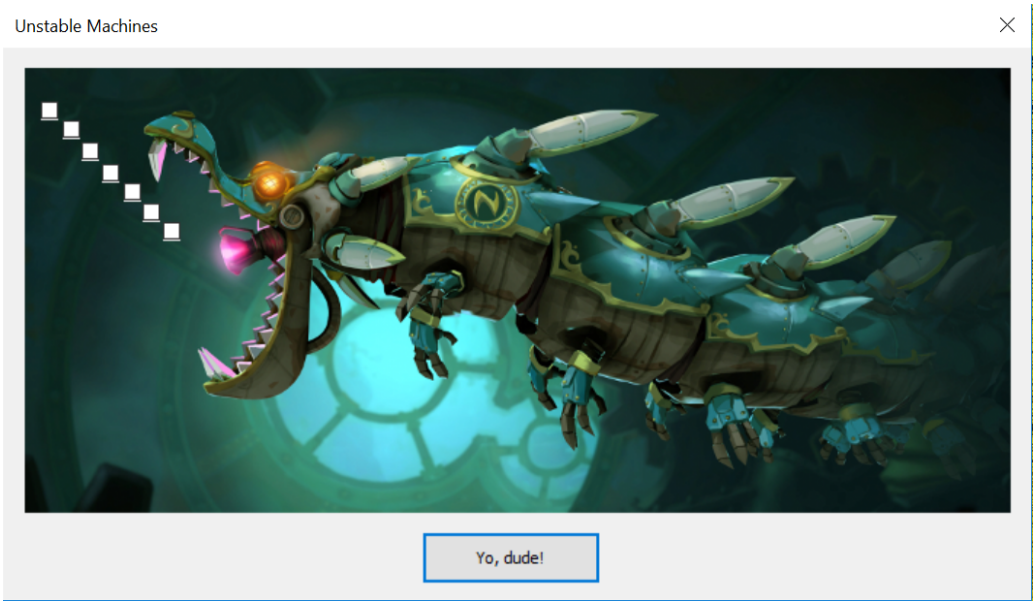
L'épreuve est un PE x86. Fini les architectures exotiques [13] ! Il s'agit d'une application graphique qui contient un bouton, 7 cases à cocher et une image de fond sur le thème de Rayman.

Petite précision, au moment de la rédaction [14] de cette solution, je n'ai plus accès à la VM Windows sur laquelle j'ai effectué cette épreuve, ni à mes notes. J'ai seulement récupéré mes scripts. Donc désolé :)

## 5.1 Monkey

Rapidement, je comprends qu'il faut cocher et décocher les cases selon une séquence bien précise. En cas d'erreur, toutes les cases se décochent et on recommence. A chaque fois, on a une chance sur 7 de trouver la case suivante. Plutôt que de commencer la rétroconception, je me suis transformé en singe, cliquant frénétiquement partout pour bruteforcer la séquence à la main, sans savoir quelle longueur elle fera. Qui de ma patience, de ma dextérité, de ma mémoire ou de ma souris cédera en premier ? Finalement, la séquence n'était pas trop longue. En considérant que la case en bas à droite est la numéro 1, la séquence de cochage/décochage est 3 4 2 6 1 4 7 5 7 1 5.

13. Je ne crois pas si bien dire...
14. un peu tardive, je dois l'avouer




Une fois cette séquence réalisée, un champ texte permettant d'entrer un mot de passe apparaît, et un bouton permet de le soumettre. Ça y est, il va vraiment falloir commencer à reverser pour savoir comment est vérifié ce mot de passe.

Je précise que je n'ai jamais cherché à patcher le PE pour supprimer la séquence des cases à cocher, ou pour l'automatiser d'une manière ou d'une autre. En effet, je crois que j'ai fini par prendre goût à la nécessité de devoir ré-entrer cette séquence à chaque fois que je relance le débugger.

## 5.2 VM

Par analyse statique, on retrouve l'endroit qui vérifie la séquence des cases à cocher. On constate que chaque clic va modifier la valeur de plusieurs variables globales. Une fois la séquence correcte rentrée, l'une de ces variables forme un LUM qui est écrit sur la stack : LUM{2KREDvn3OPf}.

L'autre variable va servir de clé permettant de déchiffrer tout un tas de choses. Sans entrer dans les détails, on découvre que le code qui vérifie le mot de passe est en fait une machine virtuelle maison, ayant la particularité sadique d'utiliser des registres chiffrés.

Après un travail relativement fastidieux de rétroconception, on récupère le code qui sera exécuté par cette VM, et on écrit un désassembleur qui fasse abstraction de la complexité induite par l'utilisation de registres chiffrés.

```python
import json
import struct

with open("segments.json") as f:
        segments = json.load(f)

class Disass:
        def __init__(self, data):
```

```
9                 self.data = bytearray(data)
10                self.i = 0
11                self.locations = set()
12                self.functions = set()
13
14        def get_opcode(self):
15                op = self.data[self.i]
16                self.i += 1
17                return op
18
19        def get_imm(self):
20                imm = struct.unpack("<I", self.data[self.i:self.i+4])[0]
21                self.i += 4
22                return imm
23
24        def get_reg_imm(self):
25                arg = self.get_imm()
26                reg = arg >> 0x1D
27                imm = arg & 0x1FFFFFFF
28                return reg, imm
29
30        def get_reg_reg(self):
31                arg = self.get_opcode()
32                dst = (arg >> 4) & 7
33                src = (arg >> 1) & 7
34                return dst, src
35
36        def disass_one(self):
37                opcode = self.get_opcode()
38                op = opcode & 0xF8
39                # MOV
40                if op == 0xE8:
41                        mov_type = opcode & 3
42                        # Mov Reg, Imm
43                        if mov_type == 0:
44                                dst, imm = self.get_reg_imm()
45                                return "mov  r%i, %s" % (dst, hex(imm))
46                        # Mov Reg, Reg
47                        elif mov_type == 1:
48                                dst, src = self.get_reg_reg()
49                                return "mov  r%i, r%i" % (dst, src)
50                        # Mov Mem, Reg
51                        elif mov_type == 2:
52                                dst, src = self.get_reg_reg()
53                                return "mov  [r%i], r%i" % (dst, src)
54                        # Mov Reg, Mem
55                        elif mov_type == 3:
56                                dst, src = self.get_reg_reg()
57                                return "mov  r%i, [r%i]" % (dst, src)
58
59                # ADD
60                elif op == 0xB0:
61                        # Add Reg, Imm
62                        if (opcode >> 1) & 1 == 0:
63                                reg, imm = self.get_reg_imm()
64                                return "add  r%i, %s" % (reg, hex(imm))
65                        # Add Reg, Reg
66                        else:
67                                dst, src = self.get_reg_reg()
68                                return "add  r%i, r%i" % (dst, src)
69
70                # SUB
71                elif op == 0x80:
72                        # Sub Reg, Imm
73                        if (opcode >> 1) & 1 == 0:
74                                reg, imm = self.get_reg_imm()
75                                return "sub  r%i, %s" % (reg, hex(imm))
76                        # Sub Reg, Reg
77                        else:
78                                dst, src = self.get_reg_reg()
79                                return "sub  r%i, r%i" % (dst, src)
80
81                # XOR
82                elif op == 0x68:
```

```
                # Xor Reg, Imm
                if (opcode >> 1) & 1 == 0:
                        reg, imm = self.get_reg_imm()
                        return "xor  r%i, %s" % (reg, hex(imm))
                # Xor Reg, Reg
                else:
                        dst, src = self.get_reg_reg()
                        return "xor  r%i, r%i" % (dst, src)

        # AND
        elif op == 0x50:
                imm = self.get_opcode()
                reg = opcode & 7
                return "and  r%i, %s" % (reg, hex(imm))

        # LSHIFT
        elif op == 0xE0:
                count = self.get_opcode()
                reg = opcode & 7
                return "lsh  r%i, %s" % (reg, hex(count))

        # PUSH
        elif op == 0xC8:
                # Push Imm
                if (opcode >> 2) & 1 == 0:
                        imm = self.get_imm()
                        return "push %s" % hex(imm)
                # Push Reg
                else:
                        data = self.get_opcode()
                        reg = (data >> 3) & 7
                        return "push r%i" % reg

        # CALL
        elif op == 0x18:
                offset = self.get_opcode()
                addr = offset + ((opcode & 7) << 8)
                if addr & 0x400 == 0:
                        loc = self.i + addr
                else:
                        loc = self.i - (addr & 0x3FF)
                self.functions.add(loc)
                return "call %04X" % loc

        # JUMP
        elif op == 0xA0:
                offset = self.get_opcode()
                addr = offset + ((opcode & 7) << 8)
                if addr & 0x400 == 0:
                        loc = self.i + addr
                else:
                        loc = self.i - (addr & 0x3FF)
                self.locations.add(loc)
                return "jmp  %04X" % loc

        # JEQ
        elif op == 0x30:
                # Jeq Reg, Imm, Offset
                if opcode & 1 == 0:
                        reg, data = self.get_reg_imm()
                        imm = (data >> 0x13) & 0xFF
                        msg = "r%i, %s" % (reg, hex(imm))
                # Jeq Reg, Reg, Offset
                else:
                        data = self.get_imm()
                        dst = data >> 0x1D
                        src = (data >> 0x12) & 7
                        msg = "r%i, r%i" % (dst, src)
                addr = (data >> 6) & 0x7FF
                if addr & 0x400 == 0:
                        loc = self.i + addr
                else:
                        loc = self.i - (addr & 0x3FF)
                self.locations.add(loc)
```

```
157                         return "jeq  %s, %04X" % (msg, loc)
158
159
160                 # RET
161                 elif op == 0x58:
162                         return "ret"
163
164                 # Special Opcodes
165                 elif op == 0x78:
166                         return "hash r0"
167
168                 elif opcode == 0x97:
169                         return "pix  r0"
170
171                 elif op == 0x00:
172                         return "add GLOBAL_CTR, r0"
173
174                 elif op == 0x28:
175                         return "spag r0"
176
177                 elif op == 0x48:
178                         return "Exit"
179
180                 else:
181                         offset = self.i - 1
182                         if offset in self.locations or offset in self.functions:
183                                 raise Exception("Unknown opcode %s (family %s)" % (hex(opcode), hex(op)))
184                         else:
185                                 return "Warning: Unable to disassemble byte 0x%02X (%r)" % (opcode, chr(opcode))
186
187         def disass_all(self):
188                 while self.i < len(self.data):
189                         offset = self.i
190                         if offset in self.functions:
191                                 print "\nsub_%04X:" % offset
192                         elif offset in self.locations:
193                                 print "\nloc_%04X:" % offset
194                         d = self.disass_one()
195                         hexdump = str(self.data[offset:self.i]).encode("hex").upper().ljust(10, " ")
196                         print "%04X  %s  %s" % (offset, hexdump,  d)
197
198 if __name__ == "__main__":
199         d = Disass(segments[0]["data"].decode("hex"))
200         d.disass_all()
```

```
0000  E800215705  mov  r0, 0x5572100
0005  B000100000  add  r0, 0x1000
000A  1802        call 000E
000C  A0E1        jmp  00EF

sub_000E:
000E  E911        mov  r1, r0
0010  CAED631C5B  push 0x5b1c63ed
0015  C8B38EBB7E  push 0x7ebb8eb3
001A  C80676ABB8  push 0xb8ab7606L
001F  C8F9FF068A  push 0x8a06fff9L
0024  C86816A2F0  push 0xf0a21668L
0029  C904532E96  push 0x962e5304L
002E  C953456436  push 0x36644553
0033  CAA0750304  push 0x40375a0
0038  E808000000  mov  r0, 0x8
003D  78          hash r0
003E  EE11        mov  [r1], r0
0040  CB5A16B2E9  push 0xe9b2165aL
0045  CB8B7F3A0E  push 0xe3a7f8b
004A  CAFF19C9F4  push 0xf4c919ffL
004F  CB997406FD  push 0xfd067499L
0054  C83E796828  push 0x2868793e
0059  CB28E5A3E2  push 0xe2a3e528L
005E  C8E8581E7B  push 0x7b1e58e8
0063  CA88D35119  push 0x1951d388
0068  E808000000  mov  r0, 0x8
006D  7C          hash r0
006E  B104000020  add  r1, 0x4
0073  EA10        mov  [r1], r0
0075  C89C000B4B  push 0x4b0b009c
007A  CACE6AB132  push 0x32b16ace
007F  CA5E560B4B  push 0x4b0b565e
0084  C9C1570B4B  push 0x4b0b57c1
0089  CBCE570B4B  push 0x4b0b57ce
008E  CB30570B4B  push 0x4b0b5730
0093  C88A1DE8A2  push 0xa2e81d8aL
0098  CB5E560B4B  push 0x4b0b565e
009D  C946510B4B  push 0x4b0b5146
00A2  EC09000000  mov  r0, 0x9
00A7  7D          hash r0
00A8  B004000020  add  r1, 0x4
00AD  EA91        mov  [r1], r0
00AF  C800EF015B  push 0x5b01ef00
00B4  CBB14B0F00  push 0xf4bb1
00B9  CB85EB805C  push 0x5c80eb85
00BE  CB03E002B6  push 0xb602e003L
00C3  C9D5A88F50  push 0x508fa8d5
00C8  C93781FCC5  push 0xc5fc8137L
00CD  C9C319F943  push 0x43f919c3
00D2  CAE882AAD8  push 0xd8aa82e8L
00D7  E808000000  mov  r0, 0x8
00DC  79          hash r0
00DD  B004000020  add  r1, 0x4
00E2  EE90        mov  [r1], r0
00E4  B0800000E0  add  r7, 0x80L
00E9  B5040000E0  add  r7, 0x4L
00EE  58          ret
```

```
loc_00EF:
00EF  E803000000  mov  r0, 0x3
00F4  97          pix  r0
00F5  325010C023  jeq  r1, 0x78, 013B
00FA  321613F027  jeq  r1, 0xfe, 014B
00FF  3445199821  jeq  r1, 0x33, 0169
0104  3682249020  jeq  r1, 0x12, 019B
0109  302B260825  jeq  r1, 0xa1, 01A6
010E  341C2D7825  jeq  r1, 0xaf, 01C7
0113  324C347024  jeq  r1, 0x8e, 01E9
0118  32BC3B9820  jeq  r1, 0x13, 020B
011D  32F142D825  jeq  r1, 0xbb, 022D
0122  30B66CE023  jeq  r1, 0x7c, 02D9
0127  36C7529021  jeq  r1, 0x32, 0277
012C  30865D7821  jeq  r1, 0x2f, 02A7
0131  30B670E026  jeq  r1, 0xdc, 02F8
0136  30D645C822  jeq  r1, 0x59, 0252

loc_013B:
013B  EC00215785  mov  r4, 0x5572100L
0140  B440100080  add  r4, 0x1040L
0145  B2C4        add  r4, r2
0147  EEC6        mov  [r4], r3
0149  A45C        jmp  00EF

loc_014B:
014B  EC00215785  mov  r4, 0x5572100L
0150  B040100080  add  r4, 0x1040L
0155  B6C5        add  r4, r2
0157  E8002157A5  mov  r5, 0x5572100L
015C  B5401000A0  add  r5, 0x1040L
0161  B357        add  r5, r3
0163  EF6A        mov  r6, [r5]
0165  EACC        mov  [r4], r6
0167  A47A        jmp  00EF

loc_0169:
0169  EC00215785  mov  r4, 0x5572100L
016E  B140100080  add  r4, 0x1040L
0173  B7C5        add  r4, r2
0175  EBC8        mov  r4, [r4]
0177  E8002157A5  mov  r5, 0x5572100L
017C  B4401000A0  add  r5, 0x1040L
0181  B0300000A0  add  r5, 0x30L
0186  E8010000C0  mov  r6, 0x1L
018B  EEDD        mov  [r5], r6
018D  35C6010C80  jeq  r4, r3, 0199
0192  E8000000C0  mov  r6, 0x0L
0197  EADC        mov  [r5], r6

loc_0199:
0199  A4AC        jmp  00EF

loc_019B:
019B  EDC7        mov  r4, r3
019D  E408        lsh  r4, 0x8
019F  B6C4        add  r4, r2
01A1  E989        mov  r0, r4
01A3  07          add  GLOBAL_CTR, r0
01A4  A4B7        jmp  00EF

loc_01A6:
01A6  EC00215785  mov  r4, 0x5572100L
01AB  B440100080  add  r4, 0x1040L
01B0  B030000080  add  r4, 0x30L
01B5  EFD9        mov  r5, [r4]
01B7  326C0200A0  jeq  r5, 0x0L, 01C5
01BC  ED47        mov  r4, r3
01BE  E408        lsh  r4, 0x8
01C0  B7C5        add  r4, r2
01C2  E909        mov  r0, r4
01C4  00          add  GLOBAL_CTR, r0
```

```
loc_01C5:
01C5  A4D8        jmp  00EF

loc_01C7:
01C7  E800215785  mov  r4, 0x5572100L
01CC  B440100080  add  r4, 0x1040L
01D1  B2C6        add  r4, r3
01D3  EBC9        mov  r4, [r4]
01D5  EC002157A5  mov  r5, 0x5572100L
01DA  B4401000A0  add  r5, 0x1040L
01DF  B6D5        add  r5, r2
01E1  EF6A        mov  r6, [r5]
01E3  B369        add  r6, r4
01E5  EE5D        mov  [r5], r6
01E7  A4FA        jmp  00EF

loc_01E9:
01E9  E800215785  mov  r4, 0x5572100L
01EE  B540100080  add  r4, 0x1040L
01F3  B646        add  r4, r3
01F5  EFC8        mov  r4, [r4]
01F7  E8002157A5  mov  r5, 0x5572100L
01FC  B1401000A0  add  r5, 0x1040L
0201  B7D5        add  r5, r2
0203  EBEA        mov  r6, [r5]
0205  8769        sub  r6, r4
0207  EA5C        mov  [r5], r6
0209  A51C        jmp  00EF

loc_020B:
020B  E800215785  mov  r4, 0x5572100L
0210  B540100080  add  r4, 0x1040L
0215  B6C7        add  r4, r3
0217  EF49        mov  r4, [r4]
0219  EC002157A5  mov  r5, 0x5572100L
021E  B5401000A0  add  r5, 0x1040L
0223  B754        add  r5, r2
0225  EBEB        mov  r6, [r5]
0227  6A68        xor  r6, r4
0229  EE5C        mov  [r5], r6
022B  A53E        jmp  00EF

loc_022D:
022D  EC00215785  mov  r4, 0x5572100L
0232  B044100080  add  r4, 0x1044L
0237  EB28        mov  r2, [r4]
0239  E202        lsh  r2, 0x2
023B  8524000080  sub  r4, 0x24L
0240  B745        add  r4, r2
0242  EB28        mov  r2, [r4]
0244  EC00215785  mov  r4, 0x5572100L
0249  B540100080  add  r4, 0x1040L
024E  EAC4        mov  [r4], r2
0250  A563        jmp  00EF

loc_0252:
0252  E800215785  mov  r4, 0x5572100L
0257  B544100080  add  r4, 0x1044L
025C  EB28        mov  r2, [r4]
025E  E202        lsh  r2, 0x2
0260  8124000080  sub  r4, 0x24L
0265  B6A8        add  r2, r4
0267  EC00215785  mov  r4, 0x5572100L
026C  B148100080  add  r4, 0x1048L
0271  EFB8        mov  r3, [r4]
0273  EAA6        mov  [r2], r3
0275  A588        jmp  00EF
```

```
loc_0277:
0277  EC00215785  mov  r4, 0x5572100L
027C  B144100080  add  r4, 0x1044L
0281  EBA9        mov  r2, [r4]
0283  E8002157A5  mov  r5, 0x5572100L
0288  B0481000A0  add  r5, 0x1048L
028D  EB3B        mov  r3, [r5]
028F  CD10        push r2
0291  CF98        push r3
0293  2E          spag r0
0294  B0080000E0  add  r7, 0x8L
0299  EC002157A5  mov  r5, 0x5572100L
029E  B1401000A0  add  r5, 0x1040L
02A3  EED1        mov  [r5], r0
02A5  A5B8        jmp  00EF

loc_02A7:
02A7  EC00215745  mov  r2, 0x5572100
02AC  B044100040  add  r2, 0x1044
02B1  EB04        mov  r0, [r2]
02B3  EC00215745  mov  r2, 0x5572100
02B8  B448100040  add  r2, 0x1048
02BD  EF15        mov  r1, [r2]
02BF  EC00215745  mov  r2, 0x5572100
02C4  B400100040  add  r2, 0x1000
02C9  1922        call 03ED
02CB  EC002157A5  mov  r5, 0x5572100L
02D0  B5401000A0  add  r5, 0x1040L
02D5  EAD0        mov  [r5], r0
02D7  A5EA        jmp  00EF

loc_02D9:
02D9  E800215705  mov  r0, 0x5572100
02DE  B100100000  add  r0, 0x1000
02E3  1913        call 03F8
02E5  B102000000  add  r0, 0x2
02EA  E8002157A5  mov  r5, 0x5572100L
02EF  B0401000A0  add  r5, 0x1040L
02F4  EE50        mov  [r5], r0
02F6  A609        jmp  00EF

loc_02F8:
02F8  4C          Exit

[...]

sub_03ED:
03ED  6E82        xor  r0, r1
03EF  5007        and  r0, 0x7
03F1  E001        lsh  r0, 0x1
03F3  B685        add  r0, r2
03F5  EB80        mov  r0, [r0]
03F7  5C          ret

sub_03F8:
03F8  B110000000  add  r0, 0x10
03FD  EF01        mov  r0, [r0]
03FF  5B          ret
```

Trois opcodes ne sont pas du tout standards. Celui que j'ai baptisé `hash` appelle des fonctions bizarres et prend beaucoup de valeurs en entrées, pour au final générer un entier de 4 octets.

Le programme de la VM commence par plusieurs utilisations de cet opcode `hash`. Il est utilisé pour calculer la valeur qui sera comparée avec le mot de passe de l'épreuve (après avoir subit *quelques* opérations).

Celui que j'ai baptisé `pix` va lire la valeur de plusieurs pixels provenant de l'image de fond Rayman.

L'opcode `spag` correspond à un code spaghetti assez pénible utilisant des instructions SSE. C'est une très grosse machine à état, qui va changer d'état (de manière fixe) à chaque instruction SSE exécutée. En générant un trace d'exécution de cette fonction, et en ne conservant que le code utile, on parvient à comprendre que cet opcode va réaliser plusieurs shifts et un xor sur ses registres d'entrée. On comprendra plus tard qu'il s'agit d'une des opérations de l'algorithme XTEA.

Il s'avère que le code exécuté par cette VM maison implémente une seconde VM, qui elle va chercher ses instructions dans certains bits de poids faible de l'image de fond. J'ai baptisé cette VM imbriquée dans l'autre « stegavm ».

## 5.3 StegaVM

Cacher des opcodes dans les bits de poids faible d'une image, il fallait oser. On touche au sublime. Connaissant l'amour du concepteur de cette épreuve pour la stéganographie (cf challenge SSTIC 2015), je ne sais pas si je dois considérer cette stegavm comme une revanche. Ou alors comme un hommage :p

Voici un script qui extrait les instructions contenues dans certains bits de poids faible de l'image.

```python
from PIL import Image

img = Image.open("unstable.bmp")

def dump_opcode(y):
    x = ((y ^ 0x46) + 0x42) & 0xFF
    start_idx = (x + 0xFDE7F446) % 3
    idx = start_idx
    b = ""
    for i in range(8):
        xx = x + 42*i
        pix = img.getpixel((xx, y))
        b += str(pix[idx] & 1)
        idx = (idx + 1) % 3
    return chr(int(b, 2))

with open("stegavm.bin", "wb") as f:
    for i in range(img.size[1]):
        f.write(dump_opcode(i))
```

On trouve ainsi un LUM caché à la suite des instructions de la stegavm

```
$ hd stegavm.bin
00000000  78 1c 00 33 1c 04 a1 9c  00 fe 04 1c af 04 04 bb  |x..3............|
00000010  56 47 fe 0c 00 78 20 01  af 04 20 bb ed 11 fe 10  |VG...x ... .....|
00000020  00 78 14 00 78 18 00 33  18 40 a1 57 00 fe 04 10  |.x..x..3.@.W....|
00000030  85 08 04 32 00 e5 fe 20  00 af 20 10 fe 04 14 5b  |...2... .. ....[|
00000040  08 00 2f 96 48 fe 24 00  af 24 14 13 20 24 af 0c  |../.H.$..$.. $..|
00000050  20 7c 88 4b af 14 00 fe  04 0c 47 08 04 32 be 66  | |.K......G..2.f|
00000060  fe 20 00 af 20 0c fe 04  14 78 08 0b 2f ae 58 fe  |. .. ....x../.X.|
00000070  24 00 af 24 14 13 20 24  af 10 20 78 20 01 af 18  |$..$.. $.. x ...|
00000080  20 12 5d 80 fe 08 0c fe  04 1c af 04 04 59 d6 d3  | .]..........Y..|
00000090  fe 08 10 78 20 01 af 04  20 59 d0 d6 78 20 01 af  |...x ... Y..x ..|
000000a0  1c 20 12 a2 80 dc 84 d6  4c 55 4d 7b 43 31 55 41  |. ......LUM{C1UA|
000000b0  69 64 76 5f 70 7a 4a 7d  4d f6 b0 08 8e 8a af d5  |idv_pzJ}M.......|
000000c0  71 f5 35 78 99 3e fa e2  de 77 9a cc b5 89 81 4e  |q.5x.>...w.....N|
000000d0  79 b9 94 8a 8a 67 5a 34  d1 7d d0 21 7d a5 bd 9c  |y....gZ4.}.!}...|
000000e0  35 37 76 fd cf 63 c2 b5  55 2f 0f f4 9c 16 f4 3e  |57v..c..U/.....>|
000000f0  e0 2c 41 bd ad 6f 6f 87  57 ba 79 46 2c 99 c5 c2  |.,A..oo.W.yF,...|
00000100  53 64 83 58 e8 26 fa 8c  a2 40 6a ba f2 c5 29 64  |Sd.X.&...@j...)d|
00000110  c6 86 7e c7 21 8e 75 e3  e0 fe 20 c5 38 90 42 47  |..~.!.u... .8.BG|
00000120  52 6e 28 13 93 50 03 e0  96 43 7d e7 9a db 4c c3  |Rn(..P...C}...L.|
00000130  ec 17                                            |..|
00000132
```

Comme précédemment, on va écrire un désassembleur pour ce nouveau jeu d'instruction.

```python
with open("stegavm.bin") as f:
    data = f.read()

OPCODES = [
    0x78,
    0xFE,
    0x33,
    0x12,
    0xA1,
    0xAF,
    0x8E,
    0x13,
    0xBB,
    0x7C,
    0x32,
    0x2F,
    0xDC,
    0x59
]

def R(reg):
    if reg % 4 != 0:
        raise Exception("Bad register (%s)" % hex(reg))
    return "r%i" % (reg/4)

def J(pc, arg1, arg2):
    offset = arg1 + (arg2 << 8)
    if offset & 0x8000:
        pc -= (offset & 0x7FFF)
    else:
        pc += offset
    return "loc_%02X" % pc


class Disass:
    def __init__(self):
        self.pc = 0
        self.locations = set()

    def disass_one(self, op, arg1, arg2):
        if op not in OPCODES or op == 0x78:
            return "mov  %s, %s" % (R(arg1), hex(arg2))
```

```
43
44          elif op == 0xFE: # Mov Reg, Reg
45              return "mov  %s, %s" % (R(arg1), R(arg2))
46
47          elif op == 0x33: # Cmp Reg, Imm
48              return "cmp  %s, %s" % (R(arg1), hex(arg2)) # Put 1 or 0 in the reg r12
49
50          elif op == 0xA1: # Jeq imm (Jump if Z flag is set)
51              return "jeq  %s" % J(self.pc, arg1, arg2)
52
53          elif op == 0x12: # Jump
54              return "jmp  %s" % J(self.pc, arg1, arg2)
55
56          elif op == 0xAF: # Add Reg, Reg
57              return "add  %s, %s" % (R(arg1), R(arg2))
58
59          elif op == 0x13: # Xor Reg, Reg
60              return "xor  %s, %s" % (R(arg1), R(arg2))
61
62          elif op == 0xBB: # Load password dword
63              return "mov  r0, KEY[r1*4]"
64
65          elif op == 0x59: # Store password dword
66              return "mov  KEY[r1*4], r2"
67
68          elif op == 0x2F: # Load hash dword
69              return "mov  r0, HASH[(r1^r2)&7 << 1]"
70
71          elif op == 0x7C: #
72              return "mov  r0, TODO+2    (TODO is the Value at Address 5572100 + 0x1010)"
73
74          elif op == 0x32: # Spaghetti code
75              return "spag r0, r1, r2"
76
77          elif op == 0xDC: # Exit
78              return "exit"
79
80          else:
81              raise Exception("TODO %s %s %s" %  (hex(op), hex(arg1), hex(arg2)))
82
83      def get_locations(self):
84         for i in range(0, len(data), 3):
85              self.pc = i+3
86              op, arg1, arg2 = [ord(x) for x in data[i:i+3]]
87              d = self.disass_one(op, arg1, arg2)
88              if d.startswith(("jmp", "jeq")):
89                  self.locations.add(d.split()[1].strip())
90              if d.startswith("exit"):
91                  break
92
93      def disass_all(self):
94          for i in range(0, len(data), 3):
95              loc = "loc_%02X" % i
96              if loc in self.locations:
97                  print "\n%s:" % loc
98              self.pc = i+3
99              op, arg1, arg2 = [ord(x) for x in data[i:i+3]]
100             hd = data[i:i+3].encode("hex").upper()
101             d = self.disass_one(op, arg1, arg2)
102             print "%02X %s  %s" % (i, hd, d)
103
104
105 if __name__ == '__main__':
106     d = Disass()
107     d.get_locations()
108     d.disass_all()
```

```
00 781C00  mov  r7, 0x0

loc_03:
03 331C04  cmp  r7, 0x4
06 A19C00  jeq  loc_A5
09 FE041C  mov  r1, r7
0C AF0404  add  r1, r1
0F BB5647  mov  r0, KEY[r1*4]
12 FE0C00  mov  r3, r0
15 782001  mov  r8, 0x1
18 AF0420  add  r1, r8
1B BBED11  mov  r0, KEY[r1*4]
1E FE1000  mov  r4, r0
21 781400  mov  r5, 0x0
24 781800  mov  r6, 0x0

loc_27:
27 331840  cmp  r6, 0x40
2A A15700  jeq  loc_84
2D FE0410  mov  r1, r4
30 850804  mov  r2, 0x4
33 3200E5  spag r0, r1, r2
36 FE2000  mov  r8, r0
39 AF2010  add  r8, r4      # r8 == ((v1<<4 ^ v1>>5) + v1)
3C FE0414  mov  r1, r5
3F 5B0800  mov  r2, 0x0
42 2F9648  mov  r0, HASH[(r1^r2)&7 << 1]
45 FE2400  mov  r9, r0
48 AF2414  add  r9, r5
4B 132024  xor  r8, r9
4E AF0C20  add  r3, r8
51 7C884B  mov  r0, TODO+2  # TODO is the Value at Address 5572100 + 0x1010
54 AF1400  add  r5, r0      # (B979379E + 2 ?)
57 FE040C  mov  r1, r3
5A 470804  mov  r2, 0x4
5D 32BE66  spag r0, r1, r2  # r0 = ((r1 << r2) & 0xFFFFFFFF) ^ (r1 >> (r2+1))
60 FE2000  mov  r8, r0
63 AF200C  add  r8, r3
66 FE0414  mov  r1, r5
69 78080B  mov  r2, 0xb
6C 2FAE58  mov  r0, HASH[(r1^r2)&7 << 1]
6F FE2400  mov  r9, r0
72 AF2414  add  r9, r5
75 132024  xor  r8, r9
78 AF1020  add  r4, r8
7B 782001  mov  r8, 0x1
7E AF1820  add  r6, r8
81 125D80  jmp  loc_27

loc_84:
84 FE080C  mov  r2, r3
87 FE041C  mov  r1, r7
8A AF0404  add  r1, r1
8D 59D6D3  mov  KEY[r1*4], r2
90 FE0810  mov  r2, r4
93 782001  mov  r8, 0x1
96 AF0420  add  r1, r8
99 59D0D6  mov  KEY[r1*4], r2
9C 782001  mov  r8, 0x1
9F AF1C20  add  r7, r8
A2 12A280  jmp  loc_03

loc_A5:
A5 DC84D6  exit
```

On comprend, aidé notamment par de lointains souvenirs du challenge SSTIC 2011, qu'il s'agit d'une implémentation de l'algorithme de chiffrement XTEA. La seule différence est que l'algorithme fait un "+ 2" sur la constante magique 0x9E3779B9 de XTEA.

Haha ! Pour retrouver le mot de passe de l'épreuve, il suffit maintenant de récupérer la valeur générée par les opcodes `hash`, et de la déchiffrer avec l'algorithme XTEA modifié ! Si seulement ....

## 5.4 Entourloupe n° 1

En effet, Monsieur 0xf4b nous réserve encore quelques surprises. Tout d'abord, pendant l'exécution du code de la 1ère VM, la stack va sournoisement déborder sur la zone de code exécutable, à l'occasion d'un des `push` servant à passer des arguments à l'un des opcodes `hash`. Ainsi, le code qui sert à charger la constante XTEA va être modifié (un 0x10 est remplacé par un 0xF4B). Bilan, au lieu de 0x9E3779B9+2, c'est la constante 0xd2d413b1+2 qui sera utilisée pour le déchiffrement.

## 5.5 Entourloupe n° 2

Un thread est lancé de manière discrète par le programme, en calculant dynamiquement l'adresse de CreateThread à partir du PEB, au lieu d'utiliser un import du PE. Ce thread se met en attente active sur la valeur du pointeur d'instruction de la stegavm. Une fois que la stegavm a terminé son travail, le thread va déchiffrer une zone de données contenant une ropchain, et va sauter dessus.

Un LUM est caché dans la zone déchiffrée : LUM{+zhVQqJy03q}

Petite astuce que je ne connaissais pas, un des premiers gadgets de la ropchain est l'instruction `retf`, qui est un équivalent de `pop eip; pop cs`. `retf` est ainsi utilisé pour placer 0x33 dans le registre `cs` et passer le processeur en mode 64 bits. Il y a donc une ropchain 64 bits qui va s'exécuter au sein de notre programme 32 bits. Normal !

Windbg x64 permet de *survivre* à ce changement d'architecture à la volée, et de générer une trace d'exécution. La ropchain va accéder au secret, et le modifier à l'aide d'un algorithme basique avant la vérification.

## 5.6 Entourloupe n° 3

A présent, le programme me dit que j'ai trouvé la bonne clé.... mais uniquement lorsque Windbg est attaché. Il y a donc de l'antidebug caché quelque part. J'ai cherché longtemps. J'ai cherché encore. Mais rien. Sorcellerie !

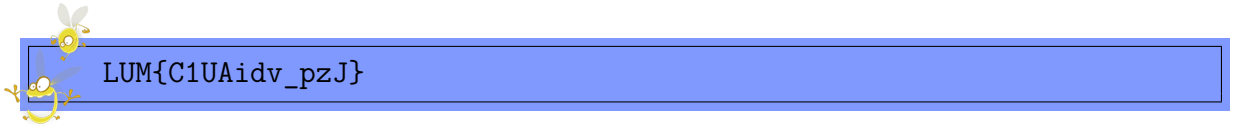
J'ai fini par faire de la dichotomie, en attachant et en détachant Windbg, afin de trouver petit à petit à quel endroit j'étais détecté. Le coupable se cache dans la fonction anodine *check_cookie*, qui vérifie à la fin de certaines fonctions si le biscuit de pile [15] est valide. Un moment, la VM s'arrange pour que le cookie soit invalide. L'analyse statique rapide de la fonction qui gère le cas d'erreur semble nous montrer que tout va se dérouler normalement. Mais une nouvelle fourberie nous attend avec l'inversion, l'air de rien, d'un `pop eax` et d'un `pop ebp` [16], qui va conduire le programme à sauter sur une nouvelle ropchain, cette fois plus petite que la précédente et en 32 bits.

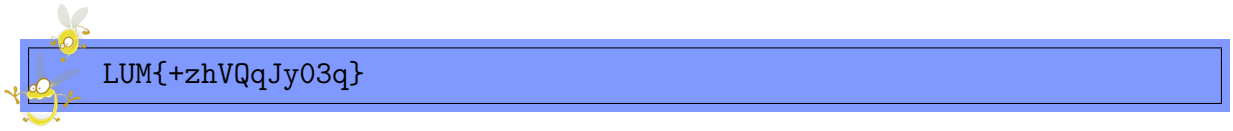
15. j'ai pas pu résister...
16. de mémoire, car je rappelle que je n'ai plus accès à ma VM...Hum...

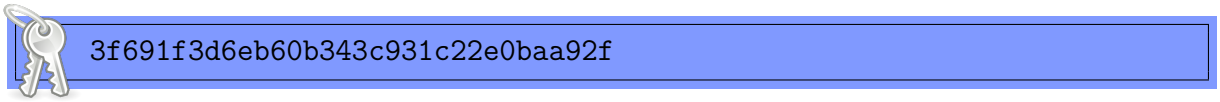
Cette seconde ropchain va lire le début de la structure PEB à la recherche du flag IsDebuggerPresent, et va utiliser la valeur de ce flag pour modifier à nouveau le secret.

On peut alors enfin calculer la vraie clé de l'épreuve !

## 5.7 Résumé des secrets


# 6 LabyQRinth

Le niveau bonus est un fichier texte *final.txt*

```
Une derniere petite etape!

        cefec06    b   a  e   0   cb519c4
        6     9 434 a4d 12   660e 4     4
        4 d94 f 9 bf f   02 b   a f 287 4
        f 01d 1 2 65   0   0   65 7 183 6
        a fd6 b   7 e7 5  a     5 d d4b c
        f     6   7  a  6   e   6 3     3
        c56b0b4 0 8 3 9 9 0 4 5 c d186a57
                60 3  8c      a 9
        2     6 214 9c b 20   d e80  65f
          422       f e95 b3   6 1 16e  8
        a    49eb 157  2  d  a  96d5  f
        d    9   2d   f4   5  a   d  6d2
        a   b55  cf36d6b657658a8abeca0 fc
          8  a  73   1  5   3 c 2c    7 5
         0 3 a4 0  2c58   86   1 f 2  56
         6 041 a   e   8  3f   3791 7   4
             6558bab a   e13   d 4 16 0
         f1 27   2 c  90 8829bb1 f8  431
         4b3 7e c    9 26  5e5 70e  c 9
        35 d61   e 9 3 a c2f c     f 7dc
        26   ad 0 4d b f4   a938ac9a7   3
        7   cc e  92    431 6 6   d   4c5
          8 f a  0da9  a0f  23  6 a3  8d
deQRypt(a  e6  8  8b 66   24   6 92 c   e
        ce80 5ba a c   d 2 5bd 81e52f c 3
                c   8f  f  3d   6   d 2
        228caa6   e2 90   6 8b3ab 5 4c973);
        a     8   d  96f9   b   e   2b7e
        0 294 e  7   0    44 0  2061d2  9
        0 40c 9   fb 44 10  91bcc4  2 44
        3 829 5  1 c0   4   8   6e  f 08d
        e     b     07 2b a6b   0  a8c7
        fd0ca91 26ad 6     9  3    a   9
```

On reconnait la forme d'un QRcode, mais il n'est pas flashable en l'état. Il convient donc de le transformer en une image avec PIL.

```python
from PIL import Image

with open("final_clean.txt") as f:
    data = f.read()

im = Image.new("RGB", (42, 35))

y = 0
for line in data.splitlines():
    for x, c in enumerate(line):
        if c in " \n":
            pix = (0xff, 0xff, 0xff)
        else:
            pix = (0, 0, 0)
        im.putpixel((x, y), pix)
    y += 1

im.save("qr.png")
```

Si on flash ce QRcode avec un téléphone, on obtient la chaîne de caractères « Please use this Nibble ADD key : 5571C2017 ».

Étant donné le nom de l'épreuve, je suis d'abord parti sur une piste qui sentait très, très mauvais, car j'ai trouvé l'existence d'un algorithme de chiffrement par flux appelé

LABYRINTH. J'ai lu sa cryptanalyse [17] en me demandant s'il utilisait quelque part une « Nibble ADD Key ». Mais après avoir constaté que j'étais incapable de mettre la main sur la moindre implémentation de LABYRINTH, je me suis dit qu'il devait y avoir plus simple.

« Nibble ADD Key » fait référence à une sorte de chiffre de Vigenère appliqué à des caractères hexadécimaux. Mais quand on déchiffre les 4 lignes comprises entre les parenthèses de *deQRypt()*, cela ne donne rien.

Après quelques efforts, la solution a fini par sauter aux yeux : si on suit les caractères, il existe un chemin continu qui permet de relier les deux parenthèses. Le voilà notre labyrinth !

On récupère le chemin le plus court. Pour cela, j'ai codé une implémentation de l'algorithme de Trémaux et de l'algorithme de Pledge [18], que j'ai fait converger grâce à une convolution discrète non-euclidienne. Bon, bon, ok... j'ai fait comme tout le monde. J'ai calculé le chemin à la main.

Le déchiffrement du chemin le plus court avec la Nibble ADD Key nous donne l'adresse email finale. Victoire !

```python
import itertools

path = "ace80e6fcca1776558babe2c51d6b657658a8abcf973d1bb5c938ac9da252fd4c973"

key = itertools.cycle("5571c2017")

email = ""
for c in path:
    k = next(key)
    i = (int(c, 16) - int(k, 16)) % 16
    email += "%x" % i

print email.decode("hex")
```

17. http://www.isg.rhul.ac.uk/~sean/simon_sh.pdf
18. https://en.wikipedia.org/wiki/Maze_solving_algorithm

```
$ python deQRypt.py
WwLnWZkEAeMjPaoKEs5K7rld@sstic.org
```

# 7 Conclusion

Comme toujours, cette édition 2017 du challenge SSTIC fut très instructive. J'ai particulièrement apprécié la partie BPF qui fut l'occasion de remettre un peu les mains dans Miasm, ainsi que tous les tricks utilisés dans Unstable Machines.

Merci pour tout et à l'année prochaine !

PS : Chers amis courageux, pour toute question ou remarque, merci de m'envoyer un message à l'adresse email ci-dessous.

```
email.ebpf.zlib

eJzdWl1sHcUVnr13l7UjExtIiGuhxjS5rWUpxSBonagSTpFJlBbZULmGVJV4qDAPIMJPMZsVUhBF
SiMVuU2qOrwkNoXYfQntiyNehkdblSo/9oGHVKrUPPKI3FS3e+Z8Z3dmdvdeXze8cCV77tmZM3O+
OWfONzN79f3KfNZjLseyv4Hsb05thapPqYns+wEj/z28B/VD2Z/u4/ZUv5fk/kL/sGm/ER6NlBoO
lPqy3W4rfAaHuVzI2jeycj4rf5CVBz/g5+nlW6ZtnDOn09YDPB/5OjwfzoxoquKjV7iMGzzOe9n3
wIzPipEZRakE/cwFP3PwLmQ4RrNyEeVc31/CIdhnnqOcXcJ4wJ2sFXYesOrns/q26Wc1/DHJCrL6
OJyz9PRvYXdm1zcNvpv8HPMQZ/2MOPMLG+b51DnMw56tUrtvOfX/MPXJMvc3NeTaTfaSCO4uufiT
ZdabgX/Ib28ff+Ep+v7aE7+Y/vnw+yH3y+MvBJgflHPBhfDp+Td/aPC2Ce/5MH2K55mmPj3L/iv8
+UVXf36R+VN/CLnJ/lOPWSY/N4w9Gw7Oa2b8dntBwT6U6R4ePz17p8a92dO4hLtxB3EPxzye6Efq
tGlRH9c/DVtUBnMhuaAuvot4nQ5/RHb9gZ+fgB9nz8BejGPHIdk1ew7PrXhrVvZ/NEyVvV6+Hz5H
cgw5fiycpnm+hnWWPT9mjV/OMxG+jfk1cdZjnliHnUVc8jrY7+UNeW7bM6Jsv/Y27n7kpwTjpxd4
XeX5oXZ93yjVjzv1n5XqH3HqEbfiR4wvfiZc37bqOz2ftP367zr1VOv133PqN/I8SXluSvLor1Fi
fucD9qfE5Qz4RP+Oy8Vs/shUjcmldbGVzed8k/VazYYiFcqn9DwyLOa21177CO03Tfu7uFOmr2Uy
hfcVaq+4/XqmRy2ONNlgiRd9EfY1wYuHgSdkPd3C+BHsjTB+1DT2LmTtaOHG9O+/ZMfd3K+lpz29
yNNbul3Yvx5BrwH7G9BrVOhtZ3lmGn46jbi7BDwh84R+DHgwvn6k6H/L69/gidhPZDc1yPFYerrK
Lltv08LTKPx3xfFfhV7mOP1M2S+jJE97OKbQrwIO+DkdXzLzOFIFngHpP3NQjsfS15Y+6UW+Xuag
Ep4QeBAnrbBKb7vAg/VM+Zz2WPpR4JE4g5/8OEtHPmDeGz/PuKy4G4IdpCH22f24cXcgj59Cb7Mt
85CsuHliMea9oX4XdsJv+lcF/iOLf7LMeY3mwdiXtd8Hv9PER4o3qra+duZvgO1z9JZuR2qQ511B
z4sjsbvVvNUu41vL8UXqX8zj4GHKq6Yf8Ferr1k1/rb4Kd8vZeAmCcfrhT12HktPrcIezEPM6zBm
vt2KzG7b1ddV68LV24wUfxH7bf3KdeXqr+k3uP3Mg9C/WGe/zCfnY8rnD5j5GyjpuXYHZtxIcQKT
PGa3r7QT/cfMF0v699Dz8zD2tbR+DI8cqrH/9A3HfonT3H5Lr5JPQo9PQvCJl49zPjkUuH45ADtj
8Al4ks5JBse71Xbn8dKP/E6CzSeWXmW8WHoOn0i+qot3W4/4BHxtz7fhkcTDURP/No4+wWHzSLe4
t/VsHpF9QAwc2Ge24go94hHEXXrsTNv3y+gu/TIg/ds8sgO/5Ho2jwieDn4p9LYtPC+2ff+M7dI/
QzmerZ78U+ht9uSfQm/NwvN8yT8Tu/TPvrx/4pud+6fQI77ZuX8sve0Cz7N8DljlfXx+jgs688ZX
xheyD++VLxBPyTU+H8y8hX7uq7MffBEIXwyV2lfajfYxp9GlSLHBej/r5fv0n3SxP0I/zDtL+mn4
Y2SW4+tVluWcInFB+0vDJ79E/zFwxcKH57D/eovLy+zXVlxzXpkv+tHOOuhyXomAC/s1ugcw/DLv
nVfeQ4l91CL2O/oo8Ml6mYQdfcDTJ/tJ9ifFcUe+sfR1N96w+UbuDTrwvcM32Bfr54AH91v6CeCR
feeJav9QfpH9Zh/GdfjG0tNVecnWqzq31O3zbT3iG5yPxP85z1z0cFwq+t3y+hUcA9KvzTOWnq6x
J9ezeQZxQ/NtcCA+Wv1VehnPPFz2h+GX1z0cb3THMZTjsPjFOqvDUehZ/CL79jp+cfQyfsG+PTO1
2fb9MrFLv7jnmZ37xT3PeHg6+KXqHJKemmB++ZDvgRZwPk6PsbwYdeaZ5E8bX+355P/km3RkysE3
gzwg+GwecPb9l8eAK8zP/Q9Yfk+W+Ryd47P6sfFJu9J5Ru5Le+UhjJ+eHQUuvqcZvE9w4d4GeUri
OlmDP7Gv1ccxLvhGeCwdP8n2Boxb8qPG+Su5eh64+Rxt96OtfqR9K2Ac1A/xVKHPEODPDV95eZHy
zF3WeEeOB868JSt8L5MsA/+D4u9beA68AfgM5xA5b9WdV/M4boBXKG/bPFZ7XoW/LD2Hx2T/V7c+
bT3isUPiT15fyQpwgkfSceC8ijjGc7oXNXyHeMjzOT6MX8pHUSXfCf/L+53c31Y/dn5K1rby+5qO
PNhfxMsVK14kTko8mNsB/NiHyX5O7ovlvZPMW477cB3uLvxo6fXEj5KH6+5bfX7M8zD8ucL3PYO4
77X9OtaTX7vwZo0fd8ybHfxXzZurWJd8HzRY4b+JXfqvkkd34L+OPNrBf1U8mqzccPxG7/sMb95b
9Fd5Px6At3geCr609HSVHa5ewZfIM7Z+JQ5Xf03iSt77Xvf8U+KJy7hHxv2e8CnFAeGS/R69/+Tz
295Sf9qJHz7PiF5+/wxZ9s+2fmX8YTzQxJLO8yX7h9YQvc+S+3np7ztZmbz0ypnr34Dd3vvQuf6P
1B8/f3NvsnwO65TPb4P3yryd8eQXPfl5T37Wkz/z5FlP5vx6HTLx4wuvvfLy47Bf+K/8Hv599dff
vPy3ZJnPafJ7i2T5pCdPefKkJ0948pgnj3oy4kh+3+G9f9bv8POH7uFyOPPTkY+evKDwkXh8HPng
IZQHH3b1urWTc3mpfrK3fr427d7ZWbu54IR69dM/3y/7+0Gcs2Q/O7hP5FuevOrJn3jyTU++4cmI
c8j0/mVU8bOElbLftOPmn+df+lypOztPcl9Qqn+0t36+Nu2O76zd/uBu9e/bTx6Ufbr8zudO/c5G
43cEeK1T+r2F3E9E6tJ/qPwfLtQzwg==
```