SSTIC

Solution du challenge 2017

Clément Berthaux

10 avril 2017

Table des matières

1 Introduction **1**

2 Épreuve 1 : Electronic Flash **2**

3 Épreuve 2 : Don't let him escape **12**

3.1 À propos du bytecode BPF 13

3.2 Développement d'un processeur BPF pour IDA 14

3.3 Rétro-ingénierie du filtre 15

4 Épreuve 3 : Riscy Zones **18**

4.1 Première approche 18

4.2 À propos de la communication avec le TEE 20

4.3 Rétro-ingénierie de *TA.elf.signed* 22

4.4 Résolution 25

5 Épreuve 4 : Unstable Machines **36**

5.1 Première approche 36

5.2 Rétro-ingénierie de la fonction *Toaster* 39

5.3 Retro-ingénierie de la seconde machine virtuelle 42

5.4 Rétro-ingénierie des étape de post-traitement 48

5.5 Résolution 52

6 Epreuve finale : LabyQRinth **57**

7 Remerciements **60**

8 Annexes : Processeurs IDA **61**

8.1 eBPF 61

8.2 RiscV 67

8.3 VM SSTIC 76

1. Introduction

Ce document retranscrit les différents étapes, outils et autres intuitions qui m'ont permis de résoudre le challenge du SSTIC 2017. Ce challenge peut être téléchargé sous la forme d'une archive zip à l'adresse `http://static.sstic.org/challenge2017/Bittendo_email_leak_by_shadowbrokers.zip`. Le but du challenge est de récupérer une adresse e-mail.

2. Épreuve 1 : Electronic Flash

L'archive du challenge contient le dump d'un e-email envoyé par un certain John Smith de China Electronic Corp à Valerian Bedecho, travaillant pour le fameux contructeur de consoles de jeux Bittendo (il fallait oser). Le corp de l'e-mail est le suivant :

```
To: valerian.bedecho@bittendo.com
From: John Smith <john.smith@china-electronic-corp.com>
Subject: Validation du firmware
[...]

Bonjour,

La chaîne de production est prête pour produire votre nouvelle console
de jeux.

Nous avons programmé un premier modèle avec le firmware que vous nous
avez fourni.
Avant de lancer la production, merci de valider que le firmware écrit
sur ce premier modèle est bien celui que vous souhaitez
déployer sur l'ensemble de vos consoles.
Pour cela, et afin de ne pas faire d'erreur, nous avons enregistré avec
un analyseur logique la programmation de la mémoire flash NAND, vous
trouverez en pièce jointe cet enregistrement. Nous utilisons le logiciel
sigrok, vous pouvez récupérer le firmware avec le décodeur parallèle sur
le bus de données. Nous utilisons uniquement l'interface graphique
"pulseview", mais il me semble qu'il y aussi des bindings python et une
interface CLI, qui vous permettront certainement d'automatiser l'extraction.

Dans l'attente de votre validation.

Cordialement,
John
```

Deux pièces jointes sont attachées à l'e-mail : une photo de mémoire flash NAND ainsi qu'une trace générée par un analyseur logique.

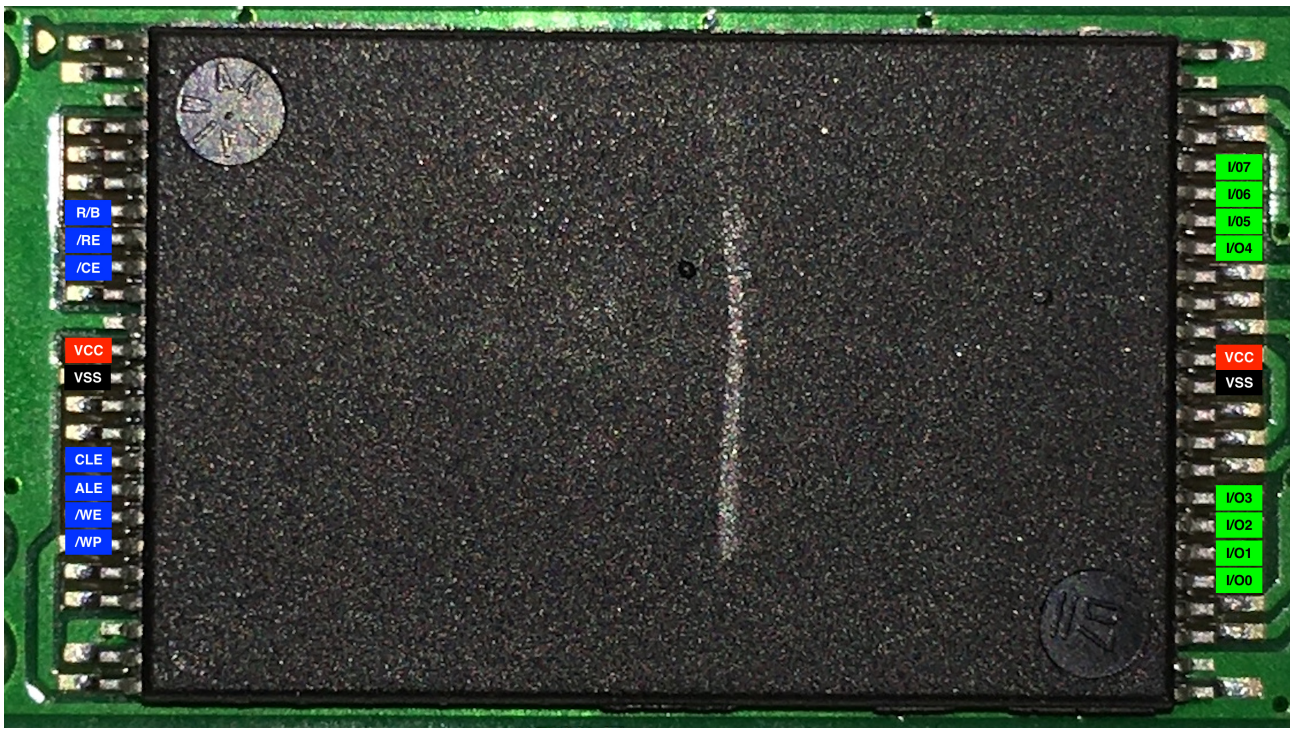


Figure 1 – photo de la mémoire NAND joite à l'e-mail

En s'appuyant sur le corp de l'e-mail, il est assez clair que le but de l'épreuve est d'extraire le *firmware* de la console de jeux depuis la trace de son écriture en mémoire. Utilisant l'outil mentionné dans l'e-mail, *pulseview*, il tout d'abord possible d'avoir une première idée de l'aspect global de la chose :

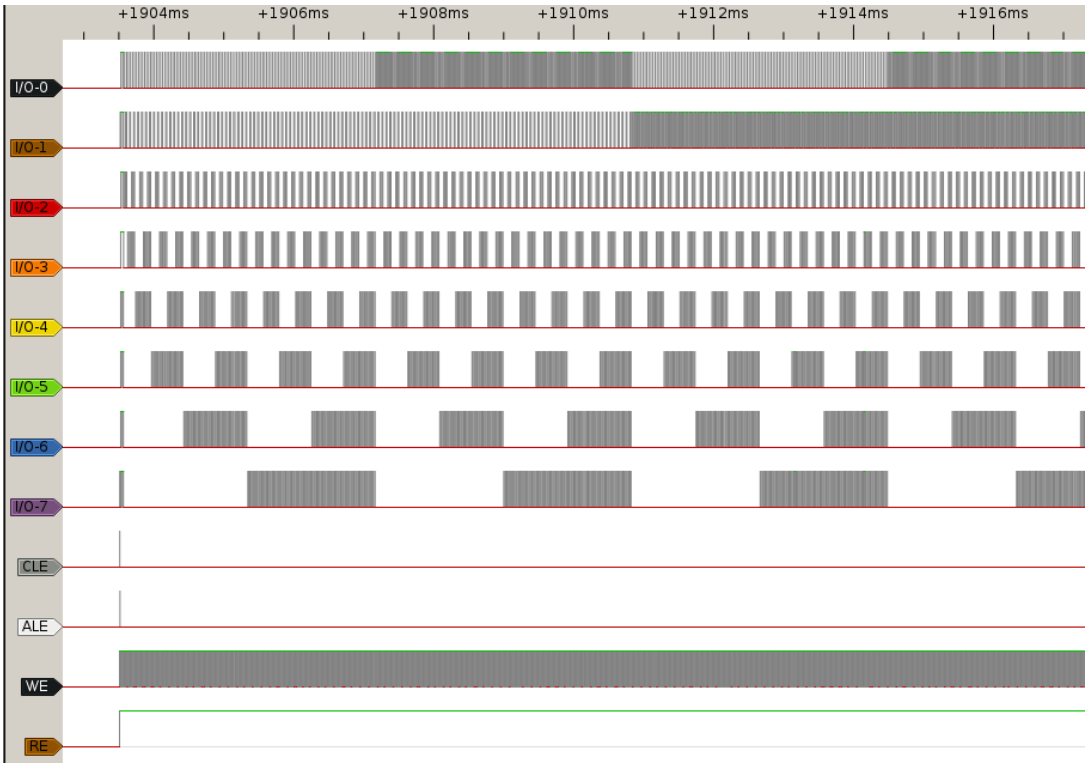


Figure 2 – Visualisation de la trace générée par l'analyseur logique

La communication avec la mémoire flash se fait par différents messages, dont on peut rapidement discerner les motifs. En googlant le nom des fiches, on tombe sur une documentation plus ou moins complète sur la mémoire utilisée.

Cette dernière nous permet de comprendre la signification des différentes fiches utilisées :

- les 7 fiches I/O sont, comme leur nom l'indique utilisées pour faire transiter les données vers et depuis le composant ;
- les fiches **RE** pour *Read Enabled* et **WE** pour *Write Enabled* font office d'horloge. Un front descendant sur WE décrit ainsi une opération d'écriture ;
- la fiche **CLE** pour permet de passer en mode commande. Lorsque sa valeur est 1, une opération d'écriture est interprétée par la puce comme le type de la prochaine commande ;
- la fiche **ALE** permet de passer en mode addressage. Toute opération d'écriture lorsque sa valeur est à 1 est interpretée comme un cycle d'addresse.

Parallèlement, le document nous renseigne sur la structure des différentes commandes permettant de communiquer avec cette mémoire flash.

Operation	Cycle 1	Cycle 2	Valid During Busy
PAGE READ	00h	30h	No
PAGE READ CACHE MODE START[1]	31h	–	No
PAGE READ CACHE MODE START LAST[1]	3Fh	–	No
READ for INTERNAL DATA MOVE[2]	00h	35h	No
RANDOM DATA READ[3]	05h	E0h	No
READ ID	90h	–	No
READ STATUS	70h	–	Yes
PROGRAM PAGE	80h	10h	No
PROGRAM PAGE CACHE[1]	80h	15h	No
PROGRAM for INTERNAL DATA MOVE[2]	85h	10h	No
RANDOM DATA INPUT for PROGRAM [4]	85h	–	No
BLOCK ERASE	60h	D0h	No
RESET	FFh	–	Yes

Figure 3 – Jeu de commandes supporté par la mémoire flash NAND

Fort de ces informations, le plan d'attaque est donc d'installer ces fameux *bindings* Python pour *libsigrock* mentionnés dans l'e-mail puis d'écrire un script Python permettant :

- de décoder les différentes commandes envoyées à la puce dans la trace ;
- de reconstruire le *firmware* à partir des commandes de type **PROGRAM PAGE** préalablement récupérées.

Hélas, on se heurte, dans la manoeuvre, au manque de documentation et de flexibilité des *bindings* Python pour *libsigrock*. En effet, il n'a pas été possible au cours de cette épreuve, de récupérer une sortie plus facilement exploitable que la suivante :

```
I/O-0:11111111 00000000 00000000 00000000 00000000 00000000 00000000 00000000
I/O-1:00000000 11111111 11111111 11000000 00000000 00000000 00000000 00000011
```

```
I/O-2:11111111 11111111 11111111 11000000 00000000 00000000 00000000 00000011
I/O-3:11111111 11111111 11111111 11000000 00000000 00000000 00000000 00000011
I/O-4:11111111 00000000 00000000 00000000 00000000 00000000 00000000 00000011
I/O-5:11111111 11111111 11111111 11000000 00000000 00000000 00000000 00000011
I/O-6:11111111 00000000 00000000 00000000 00000000 00000000 00000000 00000011
I/O-7:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
CLE:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
ALE:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
WE:11111111 11001111 11111111 11110011 11111111 11111100 11111111 11111111
RE:11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111

I/O-0:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
I/O-1:11111111 11111111 11111111 11111111 10000000 00000000 00000000 00000000
I/O-2:11111111 11111111 11111111 11111111 10000000 00000000 00000000 00000000
I/O-3:11111111 11111111 11111111 11111111 10000000 00000000 00000000 00000000
I/O-4:11111111 11111111 00000000 00000000 00000000 00000000 00000000 00000000
I/O-5:11111111 11111111 11111111 11111111 10000000 00000000 00000000 00000000
I/O-6:11111111 11111111 00000000 00000000 00000000 00000000 00000000 00000000
I/O-7:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
CLE:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
ALE:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
WE:00111111 11111111 10011111 11111111 11100111 11111111 11111001 11111111
RE:11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
[...]
```

Dans l'optique challenge/rapidité/quickn'dirty, on juge plus rapide de résoudre l'épreuve en deux temps. On *parse* tout d'abord cette sortie pour la transformer en un fichier CSV contenant l'ensemble des *ticks* d'horloge. On notera que l'approche est d'une élégance rare.

```python
from sigrok import core as sr
from collections import namedtuple
from csv import writer

context = sr.Context_create()
session = context.load_session('trace.sr')

out_file = open('ticks.csv','w')
csv_out = writer(out_file, delimiter=',')

old_re = 0
old_we = 0

ticks = []

def decode_callback(driver, packet):
    print "packet"
    global old_re, old_we, out_file
    if packet.type == sr.PacketType.HEADER:
        return

    datas = out.receive(packet).split('\n')
```

```
    i = 0
    while ':' not in datas[i]:
        i+=1

    while i+12<len(datas):
        chunk = datas[i:i+12]
        data_chunk = {}
        for c in chunk:
            channel, _, data = c.partition(':')
            data_chunk[channel] = data.replace(' ', '')

        for b in xrange(64):
            if old_re and not int(data_chunk['RE'][b]):
                old_re = 0
                csv_out.writerow([
                    'r',
                    int(data_chunk['ALE'][b]),
                    int(data_chunk['CLE'][b]),
                    int(data_chunk['WE'][b]),
                    int(data_chunk['RE'][b]),
                    sum(int(data_chunk['I/O-%d' % o][b]) << o for o in range(8))
                ])

            elif old_re != int(data_chunk['RE'][b]):
                old_re = int(data_chunk['RE'][b])

            elif old_we and not int(data_chunk['WE'][b]):
                old_we = 0
                csv_out.writerow([
                    'w',
                    int(data_chunk['ALE'][b]),
                    int(data_chunk['CLE'][b]),
                    int(data_chunk['WE'][b]),
                    int(data_chunk['RE'][b]),
                    sum(int(data_chunk['I/O-%d' % o][b]) << o for o in range(8))
                ])

            elif old_we != int(data_chunk['WE'][b]):
                old_we = int(data_chunk['WE'][b])

        i+=12

out = context.output_formats['bits'].create_output(session.devices[0])
session.add_datafeed_callback(decode_callback)
session.start()
session.run()

out_file.close()
```

On décode ensuite les différentes commandes envoyées à la Flash dans un deuxième script Python, qui prendra en entrée le fichier CSV fraichement généré.

```python
from csv import reader
from collections import namedtuple, defaultdict

Tick = namedtuple('Tick', ['mode', 'ale', 'cle', 'we', 're', 'data'])

class DecodingError(Exception):
    pass

class CommandReset:
    def __init__(self):
        pass

    @staticmethod
    def decode(csv):
        print "RESET"
        return CommandReset()

class CommandReadID:
    def __init__(self, data):
        self.data = data

    @staticmethod
    def decode(csv):
        data = ''
        t = get_next_tick(csv)

        # should be a dummy address cycle
        if not t.ale or t.data:
            raise DecodingError("wrong READ_ID command")

        t = get_next_tick(csv)

        while not t.mode == 'w':
            data += chr(int(t.data))
            t = get_next_tick(csv)


        print "READ_ID",repr(data)
        return CommandReadID(data)

class CommandEraseBlock:
    def __init__(self, addr):
        self.addr = addr

    @staticmethod
    def decode(csv):
        addr = []
        for i in range(3):
            t = get_next_tick(csv)
```

```
            if not t.ale:
                raise DecodingError("wrong BLOCK ERASE command")

            addr.append(t.data)

        t = get_next_tick(csv)

        assert t.cle and t.data == 0xd0

        t = get_next_tick(csv)

        while not t.cle:
            t = get_next_tick(csv)

        repush_tick(t)

        print "BLOCK ERASE",repr(addr)
        return CommandEraseBlock(addr)


class CommandProgramPage:
    def __init__(self, addr, data):
        self.addr = addr
        self.data = data

    @staticmethod
    def decode(csv):
        addr = []
        for i in range(5):
            t = get_next_tick(csv)
            if not t.ale:
                raise DecodingError("wrong addr for PROGRAM DATA command")
            addr.append(t.data)

        data = ""
        t = get_next_tick(csv)
        while not t.cle:
            data += chr(t.data)
            t = get_next_tick(csv)

        if not t.data == 0x10:
            raise DecodingError("wrong PROGRAM DATA command")


        print "PROGRAM DATA",addr, repr(data[:20])

        if addr[3]<8:
            DUMP[addr[3]]+=data

        t = get_next_tick(csv)

```

```
        while not t.cle:
            t = get_next_tick(csv)

        repush_tick(t)

        return CommandProgramPage(addr, data)

Commands = {
    0x90:CommandReadID,
    0xff:CommandReset,
    0x60:CommandEraseBlock,
    0x80:CommandProgramPage
}

commands = []

DUMP = defaultdict(str)

def decode_commands(csv):
    while True:
        t = get_next_tick(csv)
        if t.cle:
            if t.data not in Commands:
                raise DecodingError("undefined command 0x%x" % t.data)
            Commands[t.data].decode(csv)


repushed_tick = None

def repush_tick(t):
    global repushed_tick
    repushed_tick = t

def get_next_tick(csv):
    global repushed_tick

    # to perform lookahead kind of stuff
    if repushed_tick:
        t = repushed_tick
        repushed_tick = None
        return t

    r = csv.next()
    return Tick(*r)

with open('ticks.csv', 'r') as f:
    csv = reader(f, delimiter=',')
    try:
        decode_commands(csv)
    except StopIteration:
        pass
```

```


with open('dump', 'wb') as f:
    for i in range(8):
        f.write(DUMP[i])
```

Le *firmware* est alors reconstruit en réordonnant le contenu des blocs récupérés via les messages de type *PROGRAM DATA*.

```
$ python decode2.py
RESET
READ_ID 'SSTIC PSEUDO NAND   LUM{x.g215WiPCR}\x00\x00'
BLOCK ERASE [0, 0, 0]
PROGRAM DATA [0, 0, 0, 0, 0] '\xebX\x90mkfs.fat\x00\x02\x01 \x00\x02\x00\x00\x00'
PROGRAM DATA [0, 0, 1, 0, 0] '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
    x00\x00\x00\x00\x00\x00'
PROGRAM DATA [0, 0, 2, 0, 0] '\xf8\xff\xff\x0f\xff\xff\xff\x0f\xf8\xff\xff\x0f\x04\x00\
    x00\x00\x05\x00\x00\x00'
PROGRAM DATA [0, 0, 3, 0, 0] '\x01\x08\x00\x00\x02\x08\x00\x00\x03\x08\x00\x00\x04\x08\
    x00\x00\x05\x08\x00\x00'
PROGRAM DATA [0, 0, 4, 0, 0] '\x01\x10\x00\x00\x02\x10\x00\x00\x03\x10\x00\x00\x04\x10\
    x00\x00\x05\x10\x00\x00'
PROGRAM DATA [0, 0, 5, 0, 0] '\x01\x18\x00\x00\x02\x18\x00\x00\x03\x18\x00\x00\x04\x18\
    x00\x00\x05\x18\x00\x00'
PROGRAM DATA [0, 0, 6, 0, 0] '\x01 \x00\x00\x02 \x00\x00\x03 \x00\x00\x04 \x00\x00\x05 \
    x00\x00'
PROGRAM DATA [0, 0, 7, 0, 0] '\x01(\x00\x00\x02(\x00\x00\x03(\x00\x00\x04(\x00\x00\x05(\
    x00\x00'
PROGRAM DATA [0, 0, 8, 0, 0] '\x010\x00\x00\x020\x00\x00\x030\x00\x00\x040\x00\x00\x050\
    x00\x00'
PROGRAM DATA [0, 0, 9, 0, 0] '\x018\x00\x00\x028\x00\x00\x038\x00\x00\x048\x00\x00\x058\
    x00\x00'
 [...]
```

Nous obtenons ainsi notre premier LUM ainsi que le *firmware* écrit sur la Flash :

```
$ file firmware.dmp
firmware.dmp: 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
```

Pour continuer dans la subtilité, le second LUM attribué à l'épreuve peut rapidement être découvert en cherchant la chaîne "LUM" dans le *firmware* obtenu :

```
$ strings firmware.dmp | grep LUM
LUMQ
LUM-
X=\KLUM)
LUM{AsPBdVWz95y}
```

Après extraction du système de fichiers, nous sommes en possession d'une archive nommée challenges.zip contenant la suite du challenge.

Les épreuves suivantes se déroulent dans une machine virtuelle *OpenRisk1000* développée en JavaScript et basée sur le projet opensource jor1k. Cette machine virtuelle est exécutée dans un navigateur et lance un système GNU/Linux minimal. On notera également la présence d'un *Trusted Execution Environnment* (TEE) sous la forme d'une seconde machine virtuelle, cette fois pour l'architecture RiscV, utilisé pour les opérations de validation de clés et de LUM.

Figure 4 – Machine virtuelle sur laquelle se déroule le reste du challenge

Plusieurs challenges sont disponibles sur le système de fichiers de la machine virtuelle, dans le répertoire /challenges/ :

- **Don't let him escape**, ce qui semble être un *Berkeley Packet Filter* à *reverser* ;
- **Riscy Zones**, un binaire Openrisc1000 offrant une interface de déchiffrement utilisant le TEE ;
- **Unstable Machines**, qui est verrouillé dans l'attente de validation des deux précédents.

3. Épreuve 2 : Don't let him escape

C'est sur le challenge **Don't let him escape** que s'est tout d'abord portée notre attention. Ce dernier est composé de deux fichiers source Python ainsi qu'une bibliothèque *OpenRisc1000* implémentant une libc minimale.

Dans les grandes lignes, le challenge fonctionne de la manière suivante : le script server.py bind une *socket raw* sur l'interface de *loopback* de la machine virtuelle et y attache un *Berkeley Packet Filter* (BPF). Le but du challenge est de déterminer le bon paquet à envoyer sur l'interface pour valider le filtre et, ainsi, l'épreuve.

Pour rappel, les BPF sont compilés dans un *bytecode* interpretés par une machine virtuelle dans le noyau Linux. La résolution de l'épreuve implique donc de *reverser* le *bytecode* du BPF inclus dans le code source Python suivant.

```
#!/usr/bin/python2
import base64
import zlib
import sys
import os
from bpf import *

interface="lo"
bpf_prog = "[...]"
[...]

sock = create_raw_socket(interface)

# decompress bpf program
bpf_prog = base64.b64decode(bpf_prog)
bpf_prog = zlib.decompress(bpf_prog)


# create_bpf_table
table_fd = bpf_map_create()
print("table_fd = %d" % (table_fd))
if table_fd < 0:
    print("can't create map")
    sys.exit(1)
bpf_patched_prog=bpf_patcher(bpf_prog,table_fd)
# load bpf program
bpf_fd = load_bpf_program(bpf_patched_prog, debug=True)

[...]
    if state == 1:
        lum = payload
        print("Good job!, you find a LUM : "+lum)

```

```
        try :
            sys.path.append("/challenges/tools/")
            from tee_client import add_lum
            add_lum(lum)
        except :
            pass
    elif state == 2:
        key = payload[4:36]
        print("Good job!, key is : "+key)
        try :
            sys.path.append("/challenges/tools/")
            from tee_client import add_key
            add_key(key)
        except :
            pass
    else :
        print("Strange state !")
```

Hélas, nous nous rendons rapidement compte qu'il existe assez peu d'outils permettant d'analyser du bytecode BPF. Plusieurs approches peuvent ainsi être considérées :

- Activer le compilateur *BPF Just-In-Time* du noyau Linux et récupérer puis *reverser* le code natif équivalent ;
- Utiliser l'un des désassembleurs BPF en CLI disponibles sur Internet, ce qui peut nécessiter de plonger dans le code pour corriger quelques bugs ;
- Implémenter un processeur IDA pour désassembler le byte code BPF.

C'est cette dernière approche qui a finalement été choisie, dans la mesure où elle semblait être la plus amusante et la plus instructive.

3.1 À propos du bytecode BPF

Le *bytecode* BPF et sa variante, eBPF pour *extended BPF*, se composent d'instructions de 64 bits. La structure des instructions est la suivante :

```
msb                                                     lsb
+-------------------------------+-------------------+----+----+-----------+
|immediate                      |offset             |src |dst |opcode     |
+-------------------------------+-------------------+----+----+-----------+
```

Où :

- *immediate* est un entier de 32 bits ;
- *offset* est un entier de 16 bits ;
- *src* est l'index du registre source sur 4 bits ;
- *dst* est l'index du registre de destination sur 4 bits ;
- *opcode* est un entier de 8 bits.

Ainsi, comparé à une architecture telle que X86, le développement d'un désassembleur s'en trouve grandement facilité.

Pour de plus amples informations à ce sujet, il est possible de consulter :

- la documentation officielle du jeu d'instruction eBPF issue de Linux : `https://www.kernel.org/doc/Documentation/networking/filter.txt` ;
- une documentation non-officielle mais particulièrement claire (dont le schéma ci-dessus est issu) : `https://github.com/iovisor/bpf-docs/blob/master/eBPF.md`.

3.2 Développement d'un processeur BPF pour IDA

Les modules de désassemblage pour IDA, aussi appelés processeurs, peuvent être écrits en C/C++ ou en Python. Par praticité, c'est ce second langage qui a été choisi (on est motivé, mais quand même pas suffisamment pour le faire en C).

Le développement d'un tel processeur suit globalement le schéma suivant :

- copier/coller^W^W s'inspirer d'un code existant ;
- ajouter les mnémoniques des différents opcodes dans self.instruc ;
- ajouter les registres définis dans self.regNames ;
- modifier la fonction self.ana pour y implémenter les fonctionnalités de décodage des mnémoniques et opérandes propres à l'architecture cible ;
- modifier la fonction self.emu si nécessaire pour définir les variables de pile et autres références code/code ou code/data ;
- prendre quelques minutes/heures pour débugger …

Le processeur développé au cours du challenge est disponible en annexe chapitre 8.1.

Un fois le BPF ouvert dans IDA, le résultat est le suivant :

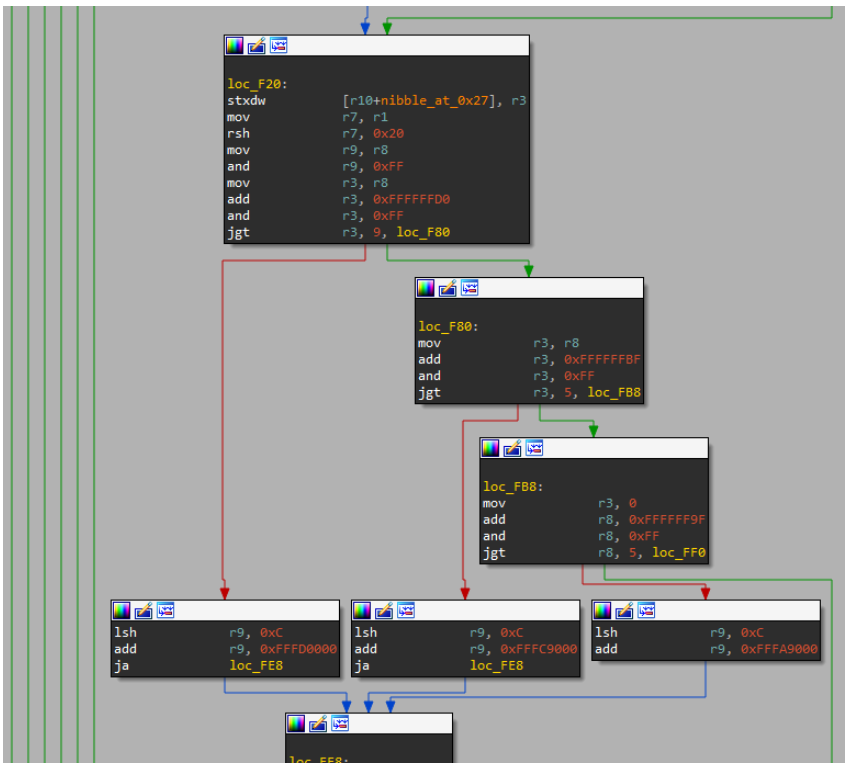


Figure 5 – Extrait du CFG du filtre dans IDA

3.3 Rétro-ingénierie du filtre

Au premier coup d'oeil sur le CFG du BPF, on en discerne rapidement plusieurs phases. La première est composés de tests sur les attributs du paquet. On apprend notamment :

- que le champ *EtherType* de la trame doit correspondre à **IPv4** (0x0800) ;
- que le champ *Protocol* du paquet IP doit être **UDP** (0x11) ;
- que le port de destination du datagramme UDP doit être **1337** (leet h4xx0r).

Le filtre vérifie si des variables de contexte sont présentes dans la map utilisée. Si du contexte est déjà existant, l'exécution continue sur la routine de vérification de la clé. Sinon, c'est la routine de vérification du LUM qui est exécutée.

Cette dernière est particulièrement simple. Après avoir vérifié que la taille du contenu du datagramme UDP était de 0x10 bytes, le différents charactères du corps du datagramme sont comparé avec des valeurs hardcodées, nous permettant de récupérer le LUM attribué à cette étape du challenge : LUM{BvWQEdCrMfA}.

```
ldxdw   r9, [r10+offset]
mov     r1, r9
add     r1, 0x1E
stxdw   [r10+offset_0x1e], r1
mov     r8, r9
add     r8, 0x1A
stxdw   [r10+offset_0x1a], r8
ldindw  r0, r8
stxdw   [r10+dword_at_0x1a], r0
mov     r8, r9
add     r8, 0x22
ldindw  r0, r8
stxdw   [r10+dword_at_0x22], r0
ldxdw   r1, [r10+offset_0x1e]
ldindw  r0, r1
mov     r8, r0
add     r9, 0x18
ldindh  r0, r9
lsh     r8, 0x20
ldxdw   r1, [r10+dword_at_0x22]
or      r8, r1
lddw    r1, 'EdCrMfA}'
jne     r8, r1, loc_26A0
```

```
ldxdw   r1, [r10+dword_at_0x1a]
lsh     r1, 0x20
rsh     r1, 0x20
jne     r1, 'BvWQ', loc_26A0
```

Figure 6 – Extrait du code de vérification du LUM

Si le LUM est correct, les indices 1 et 2 de la map utilisée sont respectivement initialisés avec les valeurs 0x42765751 ^ 0xffffffff = 0xbd89a8ae et 0x45644372 ^ 0xffffffff = 0xba9bbc8d. Ces deux constantes sont ensuite utilisées dans la procédure de vérification de la clé.

Cette dernière est plus compliquée que la précédente. Il est tout d'abord vérifié que le corps du datagramme UDP est composé de 0x25 bytes.

Le format de la clé est ensuite inspecté et doit respecter le motif KEY{XXX} où X est un mot de 128 bits en hexadécimal. La partie restante de la clé est alors décodée depuis l'hexadecimal puis scindée en 4 entiers de 32 bits.

Dans la mesure où, par design, la machine virtuelle BPF n'autorise pas les boucles, c'est cette étape de décodage qui prend la majeure partie du code.

Les 4 parties de la clé sont ensuite vérifiées indépendamment. Pour les deux premières parties, l'algo peut être trivialement inversé en lisant le code :

- le premier entier peut être retrouvé par l'opération suivante : K1 = map_key1 ^ (0x53535449 ^ (udp_size * 0x706F6C79)), où map_key1 = 0xbd89a8ae et udp_size = 0x2d ;
- le deuxième peut être retrouvé en calculant : K2 = map_key2 ^ (0x43204348 – 0x6D6F7266/udp_size) où map_key2 = 0xba9bbc8d.

Le cas des deux parties restantes est plus compliqué. Aussi, plutôt que de tenter de comprendre les aspects mathématiques de la chose et, ainsi, chercher une solution élégante, tester les 2^32 solutions

pour ces deux DWORD semblait être une approche tout à fait satisfaisante. À l'aide de copier/coller et de *sed*, il est possible de générer un code source en C reproduisant les troisième et quatrième étapes de vérification de la clé.

Après quelques minutes d'exécution, on récupère des candidats pour les dernières parties de la clé. Nous permettant ainsi de reconstruir la clé suivante : KEY{2d4ceda2fa2a0e08718fe186291de7c9}.

Toutefois, on se rend compte que la clé n'est pas reconnue par l'interface de validation mais qu'elle valide tout de même le crackme … Trois tentatives de suicides plus tard, on se rend compte qu'il existe deux solutions possibles pour la troisième partie de la clé. Arrêter le calcul après le premier candidat trouvé n'était donc peut être pas la meilleur des solutions.

La clé valide est donc : KEY{2d4ceda2fa2a0e08fc360b55291de7c9}.

Figure 7 – Validation de l'épreuve Don't Let Him Escape

4. Épreuve 3 : Riscy Zones

La troisième épreuve se compose de deux binaires, respectivement nommés TA.elf.signed et trustzone_decrypt ainsi que de 2 fichiers chiffrés : password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted et secret.lzma.encrypted.

Le principe de l'épreuve est simple : le premier binaire trustzone_decrypt permet de déchiffrer des fichiers en fournissant une clé. Le but est donc de trouver la clé permettant le déchiffrement de l'archive secret.lzma.encrypted.

```
$ ./trustzone_decrypt
usage ./trustzone_decrypt [password] [encrypted file] [destination file]
```

L'originalité de l'épeuve réside dans le fait que le déchiffrement est assuré par le TEE en y chargeant le binaire TA.elf.signed (TA pour Trusted App).

La résolution implique donc de *reverser* les deux binaires pour en comprendre les interactions et de trouver une faiblesse dans la cryptographie utilisée.

On notera également que le fait que l'extension de l'archive secret.lzma.encrypted ait été gardée suggère une possible attaque à clair connu sur l'en-tête LZMA.

4.1 Première approche

Le fichier d'exemple password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted nous fournit tout d'abords quelques informations sur le fonctionnement interne des binaires :

```
./trustzone_decrypt 00112233445566778899AABBCCDDEEFF
    password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted /tmp/out
[i] load TA.elf.signed in TrustedOS
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_GET_TA_VERSION
[+] OS return code = 0x00000000, TA return code = 0x00000000
retreived version : SSTIC Trusted APP v0.0.1
[i] check password in TEE
[+] OS return code = 0x00000000, TA return code = 0x00000000
Good password !
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
```

```
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Unload TA form TrustedOS
[+] OS return code = 0x00000000, TA return code = 0x00000000
```

À prioi, le comportement du binaire trustzone_decrypt semble être le suivant :

- le binaire TA.elf.signed est chargé dans le TEE ;
- le mot de passe fourni est vérifié dans le TEE ;
- le binaire envoie une commande CMD_DECRYPT_BLOCK pour chaque bloc du fichier d'entrée.

En utilisant *strace* et en analysant les tailles et l'enchaînement des opérations de lecture et d'écriture, on peut également récupérer quelques informations quant à la structure du chiffré :

```
$ strace -e read,write ./trustzone_decrypt 00112233445566778899AABBCCDDEEFF
    password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted bla > out
read(4, "E!O\317q/\342=O\22\30gB\370\31}\t\0\312j\216\17\367Yg\315|4%\337^\250"..., 112)
    = 112
read(4, "5\370-\3521\23\322\354\363\3555T\304\241\231\366", 16) = 16
read(4, "66LG\371\326H9\204\327\322S\333G\262!", 16) = 16
write(5, "Decrypted block\n", 16)       = 16
read(4, "\215ac\2\370\311d\27\265\324r_\354@\t\23", 16) = 16
write(5, "Decrypted block\n", 16)       = 16
read(4, "KB\6\27\364Z\343\313\253\315r=\225#\344\277", 16) = 16
write(5, "Decrypted block\n", 16)       = 16
read(4, "CmV\235\220\322\273\340\3773[\1\352\230\22}", 16) = 16
write(5, "Decrypted block\n", 16)       = 16
read(4, "\275>v\351\254\240\246-\210\270\27\354\254\\gw", 16) = 16
```

```
write(5, "Decrypted block\n", 16)        = 16
read(4, "\333t\3705G\0\3\27\241=\n\vB\212!\210", 16) = 16
write(5, "Decrypted block\n", 16)        = 16
read(4, "\320J\214\231\240,\16\331\253*\270\1\242\240\341\221", 16) = 16
write(5, "Decrypted block\n", 16)        = 16
read(4, "mX\365Z\254\7Z\341\355\236\255\25d\346b\232", 16) = 16
write(5, "Decrypted block\n", 16)        = 16
read(4, "ZF\30o\270\326\10\260\211\371\202\360k\371V'", 16) = 16
write(5, "Decrypted block\n", 16)        = 16
read(4, "\213AU\366[\325\302\326<\303\\\17\304\331[\20", 16) = 16
write(5, "Decrypted block\n", 16)        = 16
read(4, "H2\323\35\244\2320dj\303]\5a\212\325A", 16) = 16
write(5, "Decrypted block\n", 16)        = 16
read(4, "\275k\321\230\250\322\2230P\254\23\25\340y\232\302", 16) = 16
write(5, "Decrypted block\n", 16)        = 16
read(4, "\2>\204h\270@*uX:[\366\251\375%W", 16) = 16
write(5, "Decrypted block\n", 16)        = 16
read(4, "\373r\321\364]\30\31\3\371\344\310\10\303u\3\30", 16) = 16
write(5, "Decrypted block\n", 16)        = 16
read(4, "@\n\277z\243\4\33\243\277\233\221\203c\6\366Y", 16) = 16
write(5, "Decrypted block\n", 16)        = 16
read(4, "\245\230=\212.\343\311\232\227G\r\322\207\31\332>", 16) = 16
write(5, "Decrypted block\n", 16)        = 16
read(4, "\252\23\247\225\26\271\244G\206\260X\354\304`\251o", 16) = 16
write(5, "\20\20\20\20\20\20\20\20\20\20\20\20\20\20\20\20", 16) = 16
read(4, "", 16)                          = 0
+++ exited with 0 +++
```

En effet, la sortie de *strace* suggère la présence :

- d'un en-tête de taille fixe contenant une sorte de *BLOB* permettant la vérification du mot de passe ;
- d'un bloc supplémentaire, sûremement utilisé en tant que vecteur d'initialisation lors du déchiffrement ;

4.2 À propos de la communication avec le TEE

La communication avec le TEE se fait par *ioctl* sur /dev/sstic.

La structure des messages envoyés au TEE est la suivante :

```
class 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_char_p),
    ]
```

Cette dernière est extraite du module Python /challenges/tools/tee_client.py, notamment utilisé par les scripts de validation des LUM et autres clés d'épreuves.

On peut donc tenter de jouer avec la *Trusted App* en boîte noire en envoyant des commandes de type CMD_TA_MESSAGE. En testant différentes charges utiles en entrée et en s'aidant des message des erreurs émises par la *Trusted App*, on réussit finalement à comprendre que les types de messages sont codés sur 32 bits et qu'ils ont les valeurs suivantes :

- 0x00000001 = CMD_TA_INIT
- 0x00000002 = CMD_GET_TA_VERSION
- 0x00000003 = CMD_CHECK_PASSWORD
- 0x00000004 = CMD_DECRYPT_BLOCK
- 0x00000005 = CMD_GET_TA_LUM

On en profite pour récupérer le premier LUM de l'épreuve en envoyant cette dernière commande :

```
[...]
def try_cmd(msg, out_len=0x20):
    tee_msg = tee_message()
    tee_msg.cmd = tee_message.CMD_TA_MESSAGE
    tee_msg.data_in_len = len(msg)
    tee_msg.data_in = ctypes.c_char_p(msg)
    tee_msg.data_out_len = out_len
    tee_msg.data_out = ctypes.c_char_p('\x00'*out_len)

    devicehandle = open('/dev/sstic', 'r')
    data = None

    try:
        fcntl.ioctl(devicehandle, 0xc0145300, tee_msg)
        data = tee_msg.data_out
    except Exception as e:
        data = False
        print e
    finally:
        devicehandle.close()
        return data

def load_ta():
    with open('TA.elf.signed', 'rb') as f:
        data = f.read()

```

```
    tee_msg = tee_message()
    tee_msg.cmd = tee_message.CMD_LOAD_TA
    tee_msg.data_in_len = len(data)
    tee_msg.data_in = ctypes.c_char_p(data)
    tee_msg.data_out_len = tee_message.MAX_LEN
    tee_msg.data_out = ctypes.c_char_p('\x00'*tee_message.MAX_LEN)

    devicehandle = open('/dev/sstic', 'r')
    version = None

    try:
        fcntl.ioctl(devicehandle, 0xc0145300, tee_msg)

    except:
        pass
    finally:
        devicehandle.close()
        return 'Lulz'

load_ta()
print repr(try_cmd(pack("<I", 4)))
print repr(try_cmd(pack("<I", 0xdeadbeef)))
```

```
$ python ta_get_lum.py
''
'\xff\xff\xff\xffLUM{gdN8.D*@+UV}'
```

4.3 Rétro-ingénierie de *TA.elf.signed*

Dans la mesure où ce binaire est exécuté dans le TEE, on en déduit qu'il a été compilé pour l'architecture *RiscV*.

```
$ file TA.elf.signed
TA.elf.signed: ELF 32-bit LSB executable, UCB RISC-V, version 1 (SYSV), statically linked
    , not stripped
```

Comme toujours pour n'importe quelle épreuve de rétro-ingénierie du challenge SSTIC, on se heurte tout d'abord à un flagrant manque d'outils : aucun processeur IDA n'est diponible pour cette architecture et il faut compiler la toolchain *RiscV* pour obtenir un support *RiscV* avec *objdump*.

Une fois n'est pas coutume, on est donc confronté à un choix ô combien crucial :

- prendre le temps de développer un nouveau processeur pour IDA ;
- se débrouiller avec *objdump*.

La sortie d'*objdump* sur le binaire ne semblait pas particulièrement compréhensible au premier abord, si bien que l'approche IDA a été tentée.

Une nouvelle fois, l'implémentation repose sur les étapes suivantes :

- télécharger la documentation (`https://riscv.org/specifications/`) ;
- définir les *opcodes* utilisés ;
- implémenter les fonctions de décodage des différents types d'instructions ;
- déboguer, déboguer, encore déboguer.

On notera que le processeur développé pendant le challenge, disponible en annexe chapitre 8.2, ne supporte pas l'intégralité du jeu d'instructions *RiscV* mais est tout à fait suffisant pour *reverser* notre binaire.

Le résultat après ouverture du binaire dans IDA est le suivant :

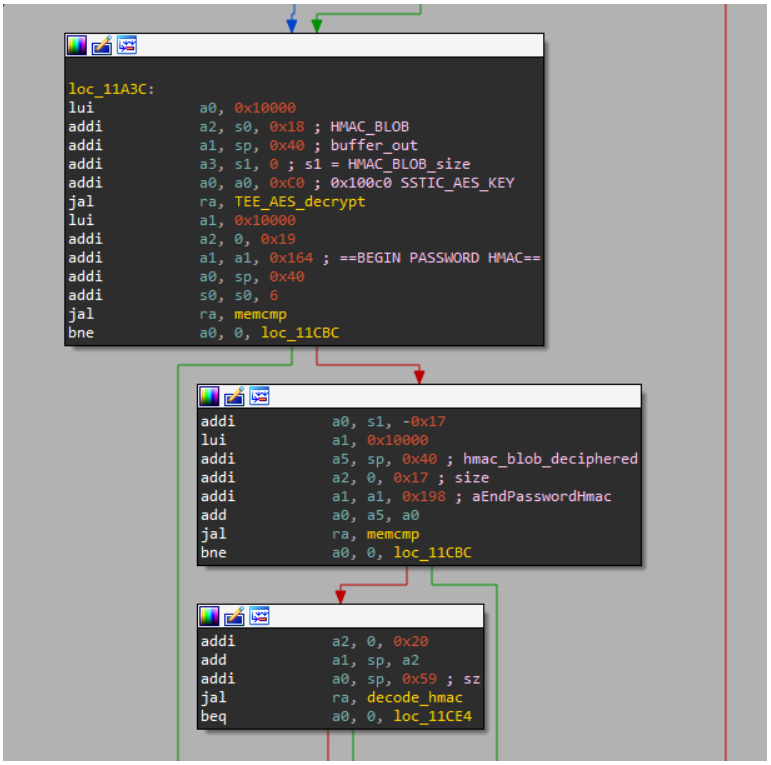


Figure 8 – Extrait du code du binaire TA.elf.signed ouvert dans IDA

En reversant plus en détail le binaire, il est possible de comprendre le comportement lié aux différents types de messages :

La commande **CMD_TA_INIT** insert deux clés hardcodées du binaire dans le *keystore* du TEE :

- SSTIC_AES_KEY dont la valeur est "___SSTIC_2017___" ;
- SSTIC_PASSWORD_HMAC_KEY qui vaut 7F DC B9 86 05 87 67 EC 47 F4 17 EF BE 85 A1 0C.

```
.rodata:00010054 ; Segment type: Pure data
.rodata:00010054 switch_cases:   dd TA_INIT
.rodata:00010058                 dd TA_VERSION
.rodata:0001005C                 dd TA_CHECK_PASSWORD    ; 0x10140
.rodata:00010060                 dd TA_DECRYPT_BLOCK
.rodata:00010064                 dd TA_GET_LUM
.rodata:00010068 aBadInputSize:  db "bad input size",0xD,0xA,0
.rodata:00010079                 db    0
.rodata:0001007A                 db    0
.rodata:0001007B                 db    0
.rodata:0001007C aBadOutputSize: db "bad output size",0xD,0xA,0
.rodata:0001008E                 db    0
.rodata:0001008F                 db    0
.rodata:00010090 aDebugCmdCmd_ta:db "[DEBUG] CMD CMD_TA_INIT",0xD,0xA,0
.rodata:000100AA                 db    0
.rodata:000100AB                 db    0
.rodata:000100AC aes_key:        db "___SSTIC_2017___",0
.rodata:000100BD                 db    0
.rodata:000100BE                 db    0
.rodata:000100BF                 db    0
.rodata:000100C0 aes_key_name:   db "SSTIC_AES_KEY",0
.rodata:000100CE                 db    0
.rodata:000100CF                 db    0
.rodata:000100D0 hmac_key:       db 0x7F, 0xDC, 0xB9, 0x86, 5, 0x87, 0x67, 0xEC, 0x47, 0xF4
.rodata:000100D0                 db 0x17, 0xEF, 0xBE, 0x85, 0xA1, 0xC
.rodata:000100E0                 db    0
.rodata:000100E1                 db    0
.rodata:000100E2                 db    0
.rodata:000100E3                 db    0
.rodata:000100E4 aSstic_password:db "SSTIC_PASSWORD_HMAC_KEY",0
.rodata:000100FC aDebugCmdCmd_ge:db "[DEBUG] CMD CMD_GET_TA_VERSION",0xD,0xA,0
.rodata:0001011D                 db    0
.rodata:0001011E                 db    0
.rodata:0001011F                 db    0
.rodata:00010120 DEBUG_GET_TA_LUM:db "[DEBUG] CMD CMD_GET_TA_LUM",0xD,0xA,0
.rodata:0001013D                 db    0
.rodata:0001013E                 db    0
.rodata:0001013F                 db    0
.rodata:00010140 aDebugCmdCmd_ch:db "[DEBUG] CMD CMD_CHECK_PASSWORD",0xD,0xA,0
```

Figure 9 – Extrait de la section .rodata du binaire

La commande **CMD_TA_CHECK_PASSWORD** prend en entrée un *HMAC BLOB* qui se trouve être l'entête de taille fixe évoquée précédemment ainsi que le mot de passe de déchiffrement fourni à *trustzone_decrypt*. Ce *BLOB* est déchiffré avec la clé SSTIC_AES_KEY pour récupérer une chaine de la forme :

```
==BEGIN PASSWORD HMAC==
HMAC
==END PASSWORD HMAC==
```

Où HMAC est un condensat de 256 bits.

Le *HMAC* du mot de passe est ensuite calculé avec la clé SSTIC_PASSWORD_HMAC_KEY puis comparé au *HMAC* extrait du *BLOB* déchiffré. Si le mot de passe est valide il est inseré en tant que clé SSTIC_CUSTOM_KEY dans le TEE.

La commande **CMD_DECRYPT_BLOCK** prend en entrée un bloc de 16 octets ainsi qu'un entier symbolisant l'index du bloc. L'opération de déchiffrement se fait de la manière suivante :

- la clé SSTIC_CUSTOM_KEY est récupérée depuis le TEE ;
- une opération de dérivation dépendant de l'index du bloc est effectuée sur la clé ;
- le bloc déchiffré est calculé par un *XOR* bit à bit entre le bloc chiffré et la clé fraîchement dérivée.

4.4 Résolution

Comme évoqué précedemment, le but de l'épreuve est de trouver la clé permettant de déchiffrer l'archive secret.lzma.encrypted. Le plan d'action pour la résolution est donc d'effectuer une attaque à clair connu sur l'en-tête LZMA pour retrouver la clé.

Toutefois, en effectuant quelques tests de compression de fichier de tailles diiférentes, on s'aperçoit qu'une en-tête LZMA ne nous permet pas de fixer tous les bits d'un bloc. En effet, la partie fixe d'une en-tête LZMA lors de la compression d'un fichier avec les options par défaut ne semble contenir que 14 octets sur les 16 qui composent un bloc :

```
$ dd if=/dev/urandom of=test& bs=10000 count=1000 && lzma test1 && hexdump -C test1.lzma
    | head -n 1
1000+0 records in
1000+0 records out
10000000 bytes (10 MB) copied, 0.729299 s, 13.7 MB/s
00000000  5d 00 00 80 00 ff ff ff  ff ff ff ff ff 00 59 be  |].............Y.|
$ rm test1.lzma
$ dd if=/dev/urandom of=test1 bs=10000 count=1000 && lzma test1 && hexdump -C test1.lzma
    | head -n 1
1000+0 records in
1000+0 records out
10000000 bytes (10 MB) copied, 0.733947 s, 13.6 MB/s
00000000  5d 00 00 80 00 ff ff ff  ff ff ff ff ff 00 65 26  |].............e&|
$ rm test1.lzma
$ dd if=/dev/urandom of=test1 bs=200 count=1000 && lzma test1 && hexdump -C test1.lzma |
    head -n 1
1000+0 records in
1000+0 records out
200000 bytes (200 kB) copied, 0.0161367 s, 12.4 MB/s
00000000  5d 00 00 80 00 ff ff ff  ff ff ff ff ff 00 4b 09  |].............K.|
```

Il conviendra donc de brute forcer ces 16 derniers bits lors de la résolution. Parallèlement, on notera que la validité de cette tentative de résolution repose sur l'hypothèse que les options de compression utilisées sont celles par défaut.

Le plan d'attaque global nécessite :

- d'inverser l'opération de dérivation de la clé ;
- de calculer la clé dérivée depuis le clair et le chiffré ;

L'opération de dérivation de la clé est composé d'un seul *basic* block et d'environ 150 instructions.

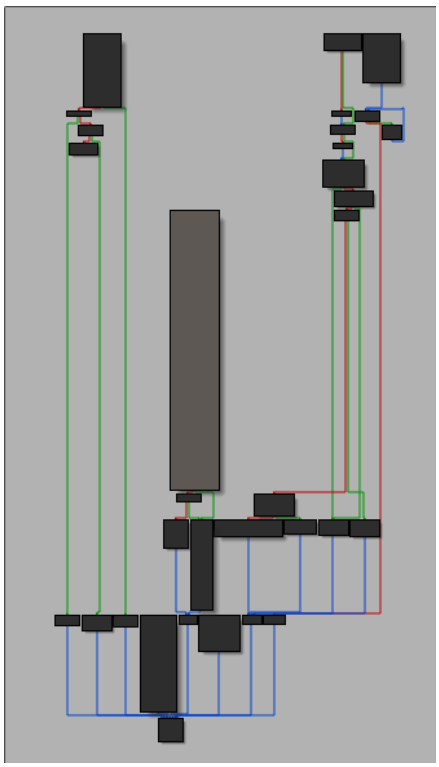


Figure 10 – Basic block responsable de la dérivation de la clé dans le CFG du binaire

```
srl     a5, a6, a5
add     a4, a0, a4
sll     a6, a6, a7
or      a5, a5, a6
addi    a0, a4, 0x56 ; add 0x56
andi    a6, a0, 0xFF
srli    a4, a5, 0x18
add     a6, a6, a4
addi    a6, a6, 0x5D ; add 0x5d
srli    a4, a5, 0x10
andi    a7, a6, 0xFF
andi    a4, a4, 0xFF
add     a7, a7, a4
addi    a7, a7, 0x64 ; add 0x64
srli    t6, a5, 8
andi    t3, a7, 0xFF
andi    t6, t6, 0xFF
addi    a4, s1, 0x11 ; add 0x11
add     t3, t3, t6
and     t5, t5, t0
andi    a4, a4, 0x1F
addi    t3, t3, 0x6B ; add 0x6b
and     a3, a3, s2
or      a3, a3, t5
sub     t6, 0, a4 ; t6 = -a4
andi    t5, t3, 0xFF
andi    a5, a5, 0xFF
add     a5, t5, a5
srl     a4, a3, a4
sll     a3, a3, t6
or      a4, a4, a3
addi    a3, a5, 0x72 ; add 0x72
andi    t5, a3, 0xFF
srli    a5, a4, 0x18
add     t5, t5, a5
addi    t5, t5, 0x79 ; add 0x79
srli    a5, a4, 0x10
andi    t6, t5, 0xFF
andi    a5, a5, 0xFF
add     t6, t6, a5
addi    t6, t6, 0x80 ; add 0x80
```

Figure 11 – Extrait du code de dérivation de la clé

Une fois n'est pas coutume, l'approche utilisée est particulièrement élégante et subtile. On récupère tout d'abord le code assembleur relatif aux opérations effectuées puis on le transforme en Python,

usant de quelques *sed* de qualité.

```python
def compute(key, block_index):
    key1, key2, key3, key4 = key & 0xffffffff, (key >> 32) & 0xffffffff, (key >> 64) & 0xffffffff, (key >> 96) & 0xffffffff
    s0 = 0
    a3 = key2
    t5 = key4
    t4 = key1
    t2 = key3

    s1 = block_index

    t0 = 0x52555655
    s2 = uint32(-0x52555656)

    s3 = 0

    a4 = uint32(s1 + 23)
    a2 = t5 & s2
    a5 = a3 & t0
    a4 = a4 & 31
    a5 = a5 | a2
    a2 = uint32(-a4 + 0)
    a4 = a5 >> (a4 & 0x1f)
    a5 = uint32(a5 << (a2 & 0x1f))
    a4 = a4 | a5
    t1 = a4 >> (0x18 & 0x1f)
    t1 = uint32(t1 + 65)
    a5 = a4 >> (0x10 & 0x1f)
    a2 = t1 & 255
    a5 = a5 & 255
    a2 = uint32(a2 + a5)
    a2 = uint32(a2 + 72)
    a0 = a4 >> (0x8 & 0x1f)
    a1 = a2 & 255
    a0 = a0 & 255
    a5 = uint32(s1 + 19)
    [...] # La fonction totalise un peu moins de 150 LOC
```

En possession d'un script permettant la dérivation de clés arbitraires, on joue ensuite avec en boîte noire pour tenter d'en comprendre le fonctionnement.

On remarque alors que certains bits de la sortie ne peuvent être modifiés que par certains bits de l'entrée. Dès lors, il est possible d'inverser le processus en fixant des bits de l'entrée et en en déduisant des contraintes.

Dans la mesure où un code vaut mieux qu'une mauvaise explication, le code Python réalisant l'inversion est le suivant :

```python
def compute(key, block_index):
```

```
    [...]

def compare_bits(key1, key2):
    changed = []
    for b in xrange(16):
        for i in range(8):
            if (key1[b] & (1<<i)) != (key2[b] & (1<<i)):
                changed.append(b*8+i)
    return changed

def analyze(n):
    ori = compute(n, 0)
    dbits = {}
    for i in xrange(128):
        dbits[i]= compare_bits(ori, compute(n ^ (1<<i), 0))

    reverse_bits = {i:[j for j in dbits if i in dbits[j]] for i in xrange(128)}
    return reverse_bits

xor = lambda a,b : ''.join(chr(ord(a[i]) ^ ord(b[i])) for i in xrange(len(a)))
str2int = lambda s:sum([ord(b) << 8*i for i,b in enumerate(s)])
int2str = lambda n:''.join(chr((n >> 8*i) & 0xff) for i in xrange(0x10))

flip = lambda n,b: n ^ (1 << b)
c = lambda n:sum(long(long(b) << (8*i)) for i,b in enumerate(compute(n, 0)))
bit = lambda n,b: int((long(n) & (1<<b)) !=0)

def inverse(g):
    n = 0
    reverse_bits = analyze(n)
    taken = []

    for j in range(0x8):
        for i in range(120+j, -1, -8):
            if bit(g, i) != bit(c(n), i):
                # print 'flip',i
                available = set(reverse_bits[i]).difference(set(taken))
                if len(available) == 1:
                    l = list(available)
                    taken.append(l[0])
                    n = flip(n, l[0])
                else:
                    print 'warning: multiple bits available :',available
                    print taken
                    t = input()
                    for b in available:
                        taken.append(b)
                    n = flip(n, t)
            else:
                # print i,taken
                taken = list(set(taken+reverse_bits[i]))
```

```
                # print taken

    return n

import sys
n = inverse(str2int(sys.argv[1].decode('hex')))
print ''.join('%02X' % ((n>>(8*i)) & 0xff) for i in xrange(0x10))
print ''.join('%02X' % ((c(n)>>(8*i)) & 0xff) for i in xrange(0x10))
```

On peut ainsi vérifier la validité de l'implémentation :

```
$ python bf_clean.py deadbeefdeadbeefdeadbeefdeadbeef
D6ABD529DA6257C10CE7B8659411C77D
DEADBEEFDEADBEEFDEADBEEFDEADBEEF
```

Nous sommes maintenant en possession d'un moyen d'inverser la dérivation de la clé, reste à déterminer la clé dérivée en fonction du clair supposé du premier bloc de l'archive secret.lzma.encrypt. Cette dernière peut être calculée de la manière suivante : key_derived = BLOC[0] ^ BLOC[1] ^ CLAIR, BLOC[0] étant utilisé comme vecteur d'initialisation par le binaire trustzone_decrypt.

On rappelle cependant qu'il faut deviner les deux derniers octets du premier bloc. Aussi, pour déterminer la validité des clés générées de manière hors-ligne, c'est-à-dire sans avoir à lancer l'attaque depuis la machine virtuelle, il est nécessaire de réimplémenter le comportement de la commande **CMD_TA_CHECK_PASSWORD**.

```
from Crypto.Cipher import AES
from hashlib import sha256
from hmac import HMAC
import sys

with open(sys.argv[1], 'rb') as f:
    hmac_blob = f.read(0x70)
    blocks = []

    chunk = f.read(0x10)
    while chunk:
        blocks.append(chunk)
        chunk = f.read(0x10)

aes = AES.new('___SSTIC_2017___')
hmac_str = aes.decrypt(hmac_blob)

print hmac_str

hmac_key = '7FDCB986058767EC47F417EFBE85A10C'.decode('hex')

print 'computed hmac = %s' % HMAC(hmac_key, sys.argv[2].decode('hex'), sha256).hexdigest
    ()
```

Toutefois, en testant ce script sur l'exemple fourni, on remarque que la clé ne semble pas valide :

```
$ python check_pass.py password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted
    00112233445566778899AABBCCDDEEFF
==BEGIN PASSWORD HMAC==
4bcea2ecf77bda18804b395a5b0a81c9c745b5add567073c3afb4ec5e7a478fa
==END PASSWORD HMAC==
computed hmac = 80d9679daf88edb5d0024d111c923ea48d9922a01da8ccdad7bf675a0be240cf
```

Il convient donc de se pencher plus en détail sur l'émulation des appels système du TEE. Cette implémentation est effectuée dans le fichier /jor1k−worker−min.js de l'archive du challenge.

Après une passe de *js-beautifier*, on est en mesure de retrouver la portion de code responsable de l'exécution des différents appels système du TEE :

```
        d.prototype.syscallHandler = function(a, b) {
            var c = b[17],
                d = b[10],
                f = b[11],
                g = b[12],
                l = b[13];
            switch (c) {
                case 1: // write
                    f = e(a, f, g);
                    f = String.fromCharCode.apply(null, f);
                    this.SendStringToTerminal(f);
                    break;
                case 2: // read key
                    d = h(a, d, 256);
                    d = this.getKey(d);
                    for (c =
                        0; c < d.length && !(c > g); c++) {
                        var k = d[c];
                        a.Write8(f + c, k)
                    }
                    break;
                case 3: // write key
                    d = h(a, d, 256);
                    f = e(a, f, g);
                    if (0 == l) this.loadKey(d, f);
                    else {
                        l = new Uint8Array([51, 137, 244, 253, 53, 246, 17, 181, 15, 206,
    70, 166, 129, 206, 151, 113]);
                        g = new Uint8Array(f.length);
                        for (c = 0; c < f.length; c++) g[c] = f[c] ^ l[c % l.length];
                        this.loadKey(d, g)
                    }
                    break;
                case 4: // aes
                    d = h(a, d, 256);
                    d = this.getKey(d);
                    c = e(a, g, l);
```

```
37                      g = (new x.ModeOfOperation.ecb(d)).decrypt(c);
38                      for (c = 0; c < g.length && !(c > l); c++) k = g[c], a.Write8(f + c,
    k);
39                      break;
40                  case 5: // hmac
41                      d = h(a, d, 256);
42                      d = this.getKey(d);
43                      l = e(a, g, l);
44                      c = new q("SHA-256",
45                                "ARRAYBUFFER");
46              console.log("hmac key",d);
47                      c.setHMACKey(String.fromCharCode.apply(null, d), "TEXT");
48                      c.update(l);
49                      l = c.getHMAC("BYTES");
50                      for (c = 0; c < l.length; c++) k = l.charCodeAt(c), a.Write8(f + c, k
    );
51                      break;
52                  default:
53                      this.PrintDebug("Syscall " + c + " not supported")
54              }
```

On s'aperçoit alors que lors de l'appel système correspondant à l'opération d'écriture d'une clé, si le quatrième argument est égal à 1, un *XOR* bit à bit de la clé avec une constante, 33 89 F4 FD 35 F6 11 B5 0F CE 46 A6 81 CE 97 71 est effectué.

C'est le cas lors de l'ajout de la clé SSTIC_PASSWORD_HMAC_KEY :

```
CMD_TA_INIT:
lui             a0, 0x10000
addi            a0, a0, 0x90 ; aDebugCmdCmd_ta
jal             ra, TEE_write_str
lui             a1, 0x10000
lui             a0, 0x10000
addi            a1, a1, 0xAC ; the_aes_key
addi            a3, 0, 0 ; add xor const ?
addi            a2, 0, 0x10 ; size
addi            a0, a0, 0xC0 ; aes_key_name
jal             ra, TEE_writekey
lui             a1, 0x10000
lui             a0, 0x10000
addi            a3, 0, 1 ; add xor const ?
addi            a2, 0, 0x10 ; size
addi            a1, a1, 0xD0 ; hmac_key
addi            a0, a0, 0xE4 ; SSTIC_PASSWORD_HMAC_KEY
jal             ra, TEE_writekey
j               loc_11608
```

Figure 12 – Activation de l'opération supplémentaire lors de l'ajout de la clé HMAC

Le calcul de la clé valide nous donne le second LUM de l'épreuve :

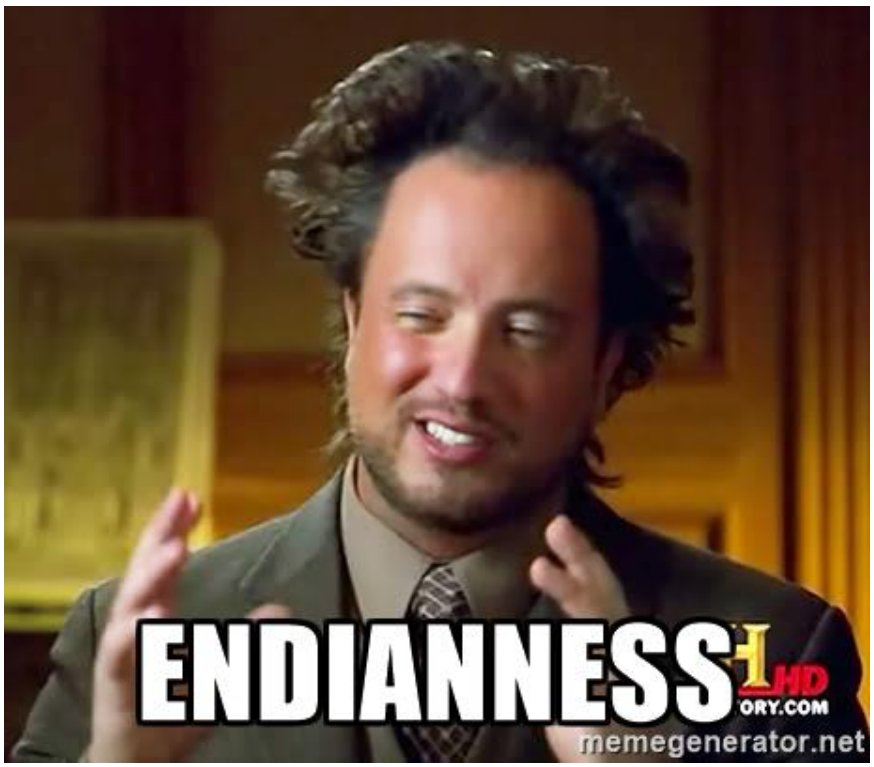
```
>>> ''.join(chr(ord(a) ^ ord(b)) for a,b in zip('3389F4FD35F611B50FCE46A681CE9771'.decode
    ('hex'), '7FDCB986058767EC47F417EFBE85A10C'.decode('hex')))
'LUM{0qvYH:QI?K6}'
```

On fixe la clé utilisée pour calculer le HMAC et on assemble les différentes briques pour obtenir notre script de résolution de l'épreuve :

```
[...]
hmac_key = 'LUM{0qvYH:QI?K6}'
secret_archive_hmac = '504861089d10a8d5900cdf3bad962004282e73c20d2d5ab95b67ae6b3356748b'.
    decode('hex')

def check_key(key):
    return HMAC(hmac_key, int2str(key), sha256).digest() == secret_archive_hmac

ciphered = str2int('88db9a9733655cbf063b590244f075db'.decode('hex'))
iv = str2int('39962d55d138efefc6e4a14d5cd7352e'.decode('hex'))
cleartext_dummy = str2int('5d00008000ffffffffffffffff00000000'.decode('hex'))

for i in xrange(0x10000):
    cleartext = cleartext_dummy + (i << 112)
    goal = ciphered ^ cleartext ^ iv
    key = inverse(goal)
    if check_key(key):
        print ''.join('%02x' % ((key>>(8*i)) & 0xff) for i in xrange(0x10))
```

Hélas, aucun candidat ne semble être valide. Quelques heures de vérification de calculs et 7 tentatives de suicides plus tard, on se souvient d'un détail quant aux deux machines virtuelles utilisées :

```
$ file trustzone_decrypt
trustzone_decrypt: ELF 32-bit MSB executable, OpenRISC, version 1 (SYSV), dynamically
    linked, interpreter /lib/ld-musl-or1k.so.1, not stripped
$ file TA.elf.signed
TA.elf.signed: ELF 32-bit LSB executable, UCB RISC-V, version 1 (SYSV), statically linked
    , not stripped
```



La machine virtuelle principale est en gros-boutiste là où le TEE est en petit-boutiste. On retente donc en inversant le boutisme du vecteur d'initialisation :

```
[...]
hmac_key = 'LUM{0qvYH:QI?K6}'
secret_archive_hmac = '504861089d10a8d5900cdf3bad962004282e73c20d2d5ab95b67ae6b3356748b'.
    decode('hex')

def check_key(key):
    return HMAC(hmac_key, int2str(key), sha256).digest() == secret_archive_hmac

ciphered = str2int('88db9a9733655cbf063b590244f075db'.decode('hex'))
iv = str2int('39962d55d138efefc6e4a14d5cd7352e'.decode('hex')[::-1])
cleartext_dummy = str2int('5d00008000ffffffffffffffff00000000'.decode('hex'))

for i in xrange(0x10000):
    cleartext = cleartext_dummy + (i << 112)
    goal = ciphered ^ cleartext ^ iv
    key = inverse(goal)
    if check_key(key):
        print ''.join('%02x' % ((key>>(8*i)) & 0xff) for i in xrange(0x10))
```

Après quelques minutes d'exécution, la clé de chiffrement de l'archive secret.lzma.encrypted est finalement retrouvée !

```
$ python solve.py
5921cd9fd3a82bd9244ece5328c6c95f
```

On déchiffre ainsi l'archive secret.lzma.encrypted.

```
./trustzone_decrypt 5921cd9fd3a82bd9244ece5328c6c95f secret.lzma.encrypted secret.lzma
[i] load TA.elf.signed in TrustedOS
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_GET_TA_VERSION
[+] OS return code = 0x00000000, TA return code = 0x00000000
retreived version : SSTIC Trusted APP v0.0.1
[i] check password in TEE
[+] OS return code = 0x00000000, TA return code = 0x00000000
Good password !
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[i] Send command to Trusted App CMD_DECRYPT_BLOCK
[+] OS return code = 0x00000000, TA return code = 0x00000000
[...]
```

Cette dernière renferme l'image suivante :

Figure 13 – Image contenue dans l'archive

```
$ file secret
secret: JPEG image data, JFIF standard 1.01, aspect ratio, density 1x1, segment length
    16, Exif Standard: [TIFF image data, little-endian, direntries=1, description=
    Congratulations : YHZ{+g%Yi.vzG8Z}], comment: " ", baseline, precision 8, 800x450,
    frames 3
```

Sa description nous donne le troisième LUM attribué à l'épreuve :

```
>>> 'YHZ{+g%Yi.vzG8Z}'.decode('rot13')
```

```
u 'LUM{+ t%Lv . imT8M} '
```

On utilise la clé de déchiffrement de l'archive pour valider la troisième épreuve.

```
/challenges/tools $ ./add_key 5921cd9fd3a82bd9244ece5328c6c95fa
OK !
```

5. Épreuve 4 : Unstable Machines

Après validation de la troisième épreuve, l'avant-dernière, **Unstable Machines** nous est enfin accessible. Cette fois, pas d'architecture exotique (du moins pour l'instant) mais un PE 32 bits avec interface graphique. Nous démarrons donc notre plus belle machine virtuelle Windows puis exécutons le binaire.

5.1 Première approche

Première surprise, le binaire semble requérir un système d'exploitation 64 bits, ce qui est pour le moins étrange de la part d'un binaire 32 bits.

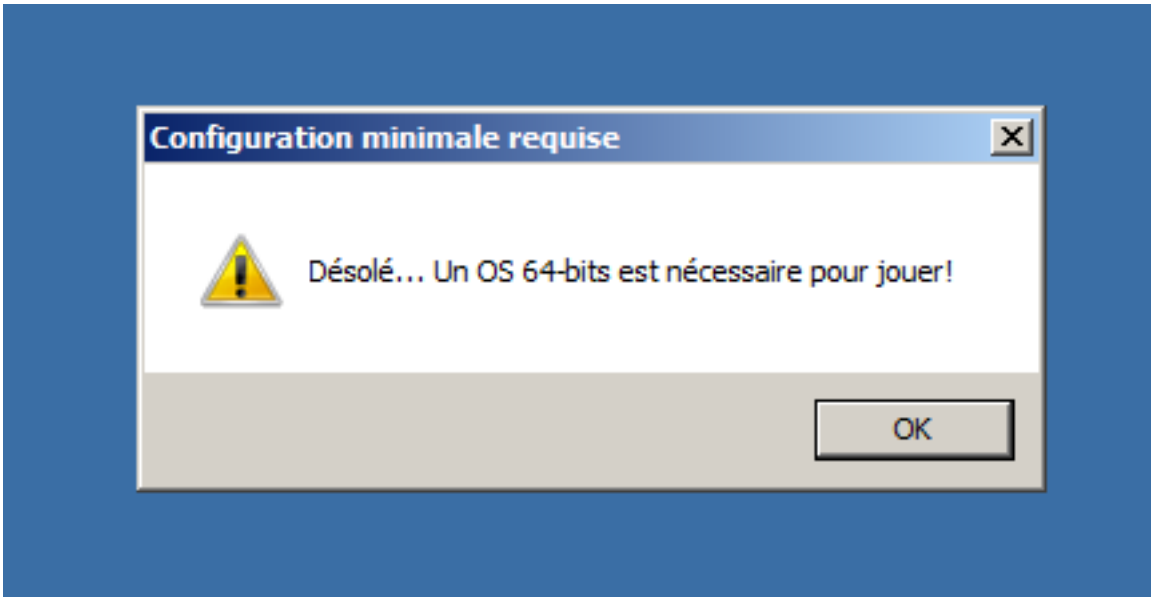


Figure 14 – Message affiché lors de l'exécution du binaire depuis une machine 32 bits

Nous démarrons donc notre deuxième plus belle machine virtuelle Windows avec un système d'exploitation 64 bits puis exécutons un nouvelle fois le binaire. Une fenêtre contenant des cases à cocher est affichée :

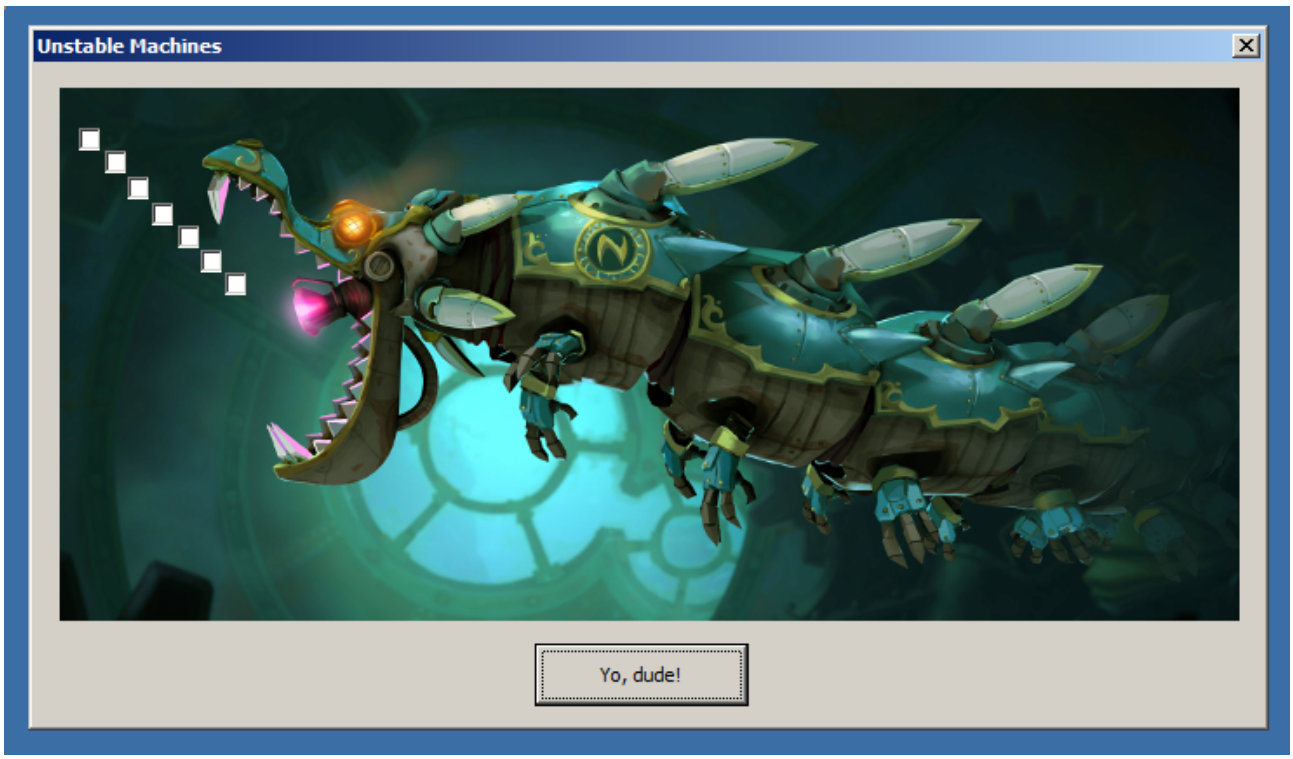


Figure 15 – Fenêtre principale du binaire

On comprend rapidement que l'on est face à une sorte de jeu où l'on doit déterminer le bon ordre dans lequel cocher les différentes cases. On ouvre donc le binaire dans IDA pour reverser le *dispatcher* des messages envoyés à la fenêtre. Cette fonction commence à l'adresse 0x401430.

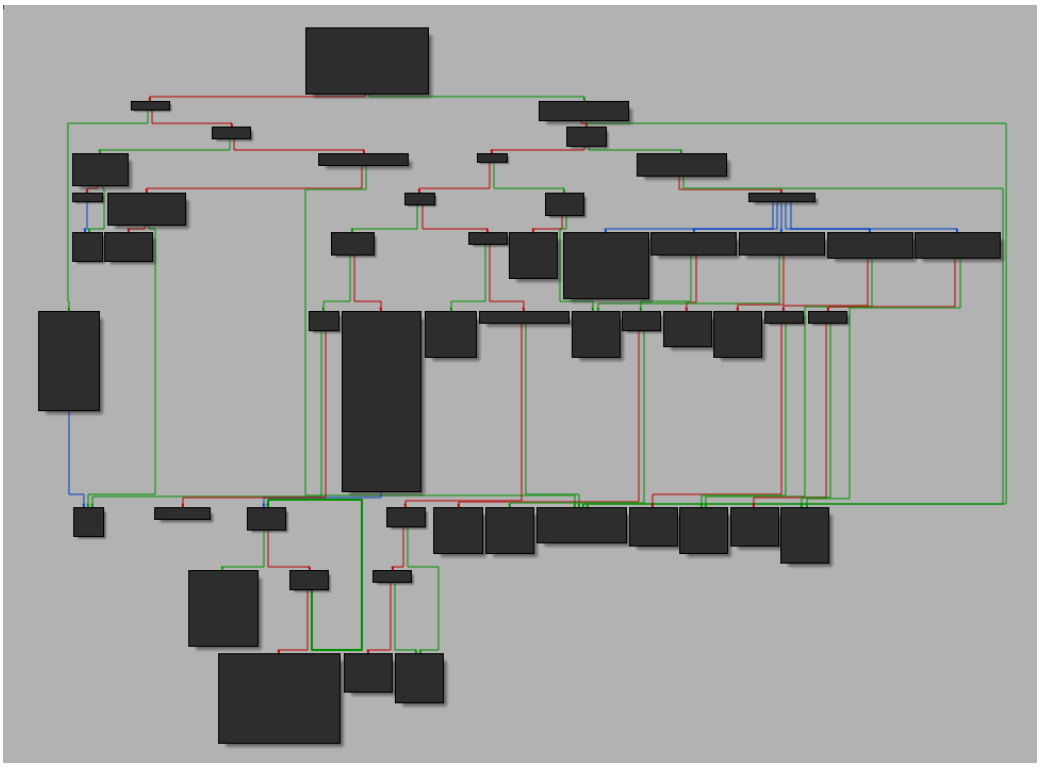


Figure 16 – CFG du dispatcher de messages

S'aidant d'une approche dynamique avec *Windbg* pour comprendre les valeurs des événements liés au clic sur les différentes cases, on détermine l'ordre valide. Il faut cliquer sur les cases (numérotées de gauche à droite) : 5, 4, 6, 2, 7, 4, 1, 3, 1, 7, et 3.

À chaque clic, plusieurs variables de context sont mises à jour.

```
switch ( (unsigned __int16)param1 )
{
  case CASE_3:
    if ( cases_state == 7 )
    {
      cases_state = 8;
      change_text(hWnd);
      state |= 0x1010101u;
      result = 1;
    }
    else if ( cases_state == 10 )
    {
      v26 = get_magical_constant_3();
      DWORD3(LUM) ^= v26 ^ state;
      state ^= v26;
      show_next_state(hWnd);                // fini
      result = 1;
    }
    else
    {
      cases_state = 0;
      reset_check_boxes(hWnd);
      change_text(hWnd);
      result = 1;
    }
    break;
  case CASE_4:
    if ( cases_state == 1 || cases_state == 5 )
    {
      state ^= 0x664A215Fu;
      ++cases_state;
      change_text(hWnd);
      result = 1;
    }
    else
    {
      cases_state = 0;
      reset_check_boxes(hWnd);
      change_text(hWnd);
      result = 1;
    }
```

Figure 17 – Extrait du code désassemblé liés au traitement des actions sur l'interface

À la fin de la séquence, l'une d'entre elles nous donne le premier LUM attribué à l'épreuve :

```
0:002> da 416614
00416614  "LUM{2KREDvn3OPf}."
```

Une fois cette première étape validée, le binaire nous demande un mot de passe. Le vrai challenge commence.

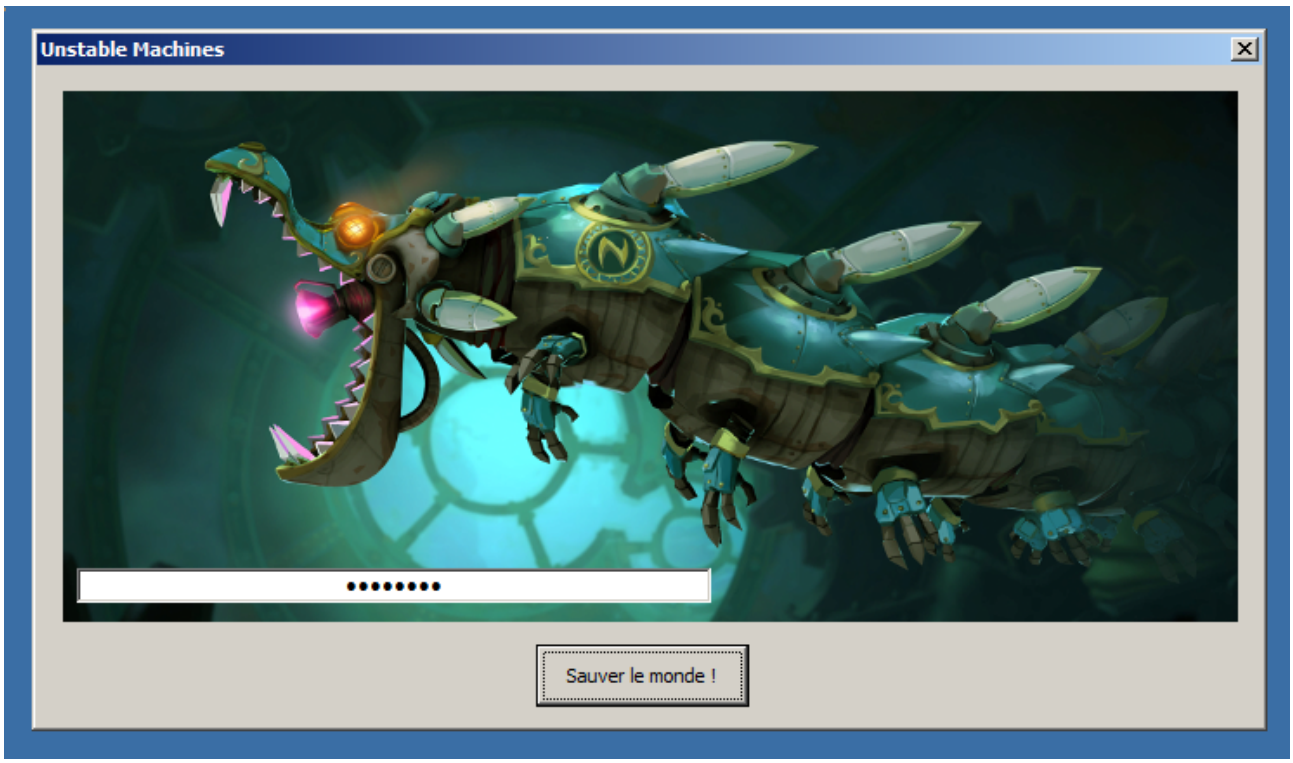


Figure 18 – Seconde fenêtre de l'épreuve

Le fonctionnement dans les très grande lignes de la vérification est classique, la clé renseignée subit différentes opérations avant d'être comparée à des valeurs écrites en dur dans le binaire. Il faut donc reverser l'ensemble de ces opérations.

La majorité de l'intelligence du binaire semble être contenue dans la fonction qui commence à l'adresse `0x402910`, baptisée **Toaster** pour l'occasion.

5.2 Rétro-ingénierie de la fonction *Toaster*

Cette fonction contient quelques faux sauts mal alignés pour freiner l'analyse, il est donc nécessaire de *patcher* quelques instructions pour y voir plus clair.

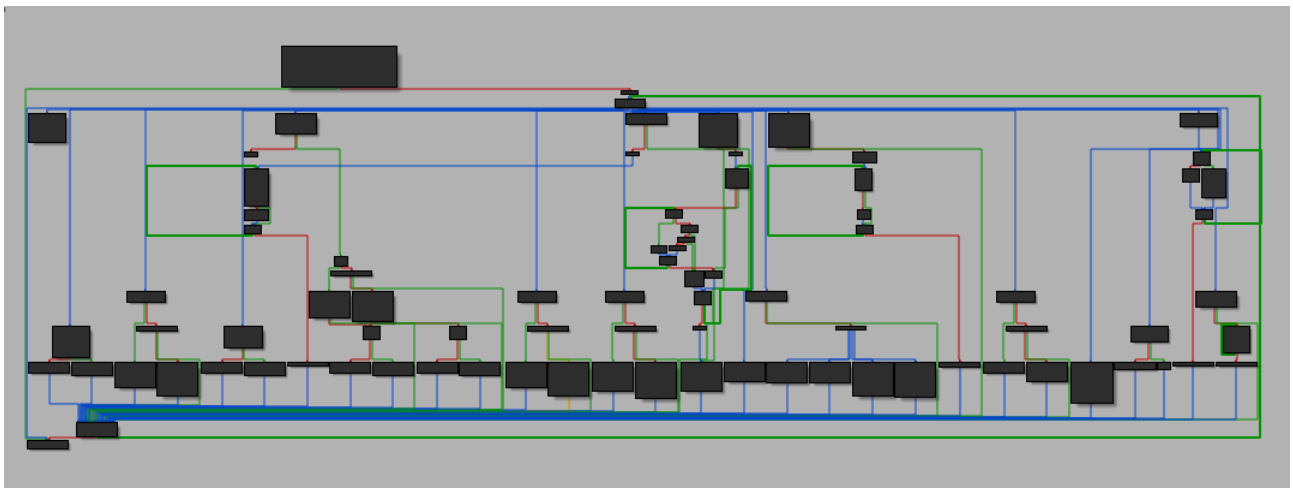


Figure 19 – CFG de la fonction Toaster

Le CFG de la fonction ressemble particulièrement à celui d'une machine virtuelle, avec une boucle principal et un commutateur vers les différents cas correspondant aux différents *opcodes*. Cette hypothèse est rapidement corroborée au fur et à mesure de l'analyse.

Le *bytecode* exécuté provient de sections virtuelles crées au début de l'exécution du programme à partir de données de la section *.data*. On *reverse* donc le prossessus de *parsing* du byte code.

Ces sections utilisées par la machine virtuelle implémentée dans la fonction *Toaster* représentent un peu plus de 5ko de code et de données. On envisage donc de développer un n-ième processeur IDA ainsi que le *loader* associé.

Le rôle d'un *loader* IDA est de *parser* un binaire, de créer les différentes sections et ainsi que la correspondance addresse virtuelle/*offset* dans le fichier. On réimplémente le processus de créaction des sections préalablement *reversées* et on obtient le résultat suivant :

```
import idaapi
from idc import *
from struct import unpack, pack

mc = [0xF4B03F4B, 0xF4B00F4B, 0xF4B01F4B, 0xF4B02F4B]

def init_data(sz):
    out = []
    sz = (sz & 0xFFFFF000) + 0x1000
    b = mc3
    for i in xrange(sz >> 2):
        b = ((b >> ((32 - 3 * i) & 0x1f)) ^ mc3 ^ (b << ((3 * i) & 0x1f))) & 0xffffffff
        out.append(b)
    return pack('I', mc3)+''.join(pack("I", i) for i in out)


from collections import namedtuple

```

```
19
20 Section = namedtuple('Section', ['type', 'vaddr', 'size', 'data'])
21 sections = []
22
23 def accept_file(f, n):
24     if n>0:
25         return 0
26
27     f.seek(0)
28     if f.read(1) == chr(0x85):
29         return 'SSTIC epreuve 4 lul'
30     return 0
31
32 def load_file(f, neflags, format):
33     f.seek(0)
34     for i in range(3):
35         t, a, size = unpack("<BII", f.read(9))
36         a = a ^ mc[i+1]
37         if t == 0xfe:
38             sections.append(Section(t, a, size, init_data(size)))
39             idaapi.add_segm(0, a, a+size, 'seg_data', 'DATA')
40
41         else:
42             sections.append(Section(t, a, size, f.read(size)+ init_data(size)[size:]))
43             idaapi.add_segm(0, a, a+size, 'text', 'CODE')
44         idaapi.mem2base(sections[-1].data, a, 0)
45
46     AddEntryPoint(sections[0].vaddr, sections[0].vaddr, 'EP', 0)
47
48     idaapi.set_processor_type("SSTICVM", idaapi.SETPROC_ALL|idaapi.SETPROC_FATAL)
49     return 1
```

La machine virtuelle implémentée se compose de 8 registres, baptisés r0, r1, r2, r3, r4, r5, r6 et sp. Le dernier registre est, comme son nom l'indique, utilisé en tant que pointeur de pile.

En *reversant* les différents opcodes définis dans la fonction *Toaster*, on s'aperçoit que plusieurs *opcodes* sont particulièrement spécifiques :

- checksum est un *wrapper* sur la fonction qui commence à l'adresse 0x401E80 du PE et qui calcule une sorte de somme de contrôle sur 32 bits à partir d'un tampon et de sa taille ;
- pixels semble récupérer des données depuis les bits de poid faible de plusieurs pixels d'une image et mettre à jour les r0 premiers registres avec les octets ainsi obtenus ;
- add y semble ajouter r0 à une variable globale du PE ;
- compute_something est un *wrapper* sur la fonction du PE disponible à l'adresse 0x402490 qui calcule un entier de 32 bits à partir d'un autre entier sur 32 bits et d'un entier sur 8 bits.

Du fait de son code plutôt imposant, la rétro-ingénierie de cette dernière fonction, baptisée très originalement compute a été laissée de côté dans un premier temps.

On écrit ainsi notre troisième processeur IDA du challenge. Il est disponible en annexe chapitre 8.3. On extrait ensuite le *bytecode* de la machine virtuelle depuis IDA :

```
with open('Z:\unstable.bin', 'wb') as f:
    f.write(GetManyBytes(0x414244, 0x151B, 0))
```

Après quelques minutes de *debugging*, on peut ouvrir le *bytecode* dans IDA. On s'aperçoit avec déception qu'il y a relativement peu de code.

5.3 Retro-ingénierie de la seconde machine virtuelle

Le programme s'articule sur une fonction centrale qui se compose d'une première phase où des constantes sont calculées ainsi qu'une boucle principale.

Figure 20 – CFG de la fonction princpipale du binaire

Une fois n'est pas coutume ce type de CFG rappelle étrangement celui d'une machine virtuelle.

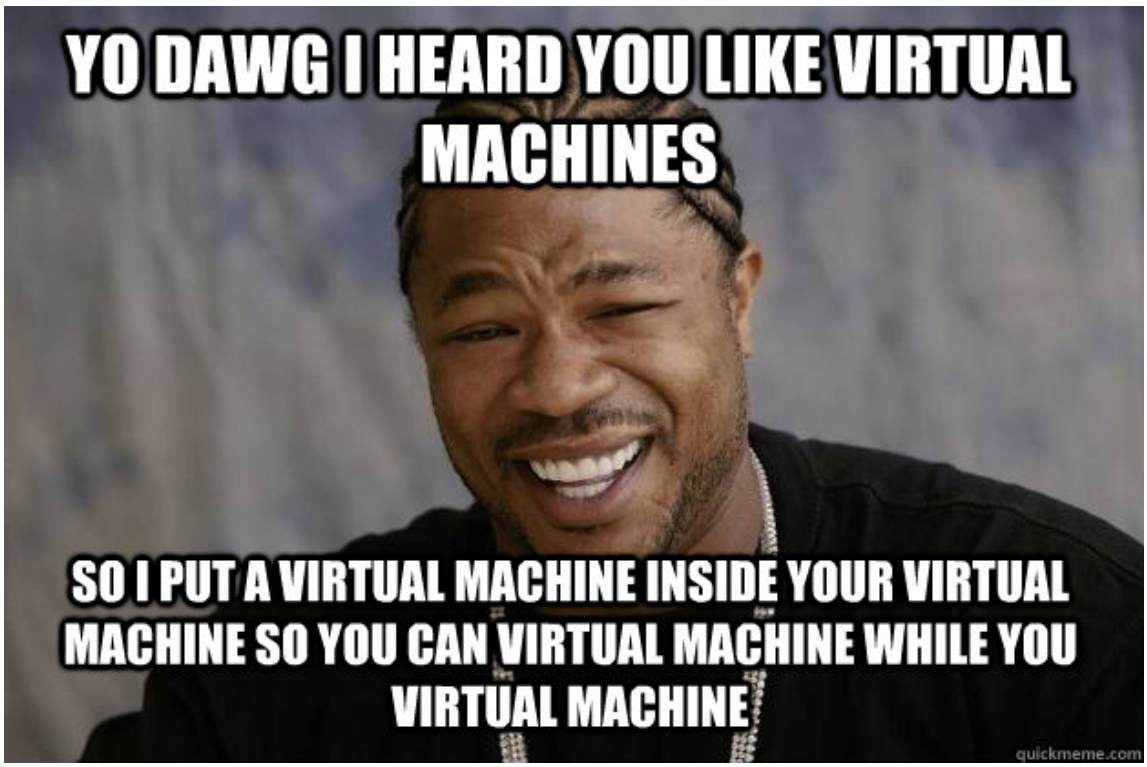


Figure 21 – Yo dawg

En analysant plus en détail cette fonction, on s'aperçoit que les instructions sont décodées depuis les pixels de l'image affichée dans la fenêtre du programme. On *reverse* donc la manière dont les instructions de cette deuxième machine virtuelle sont décodés puis on réimplémente le décodage du *bytecode* ainsi qu'un désassembleur minimaliste en texte.

```
from struct import unpack, pack
from collections import namedtuple
from termcolor import colored
from PIL import Image
from StringIO import StringIO

opcodes = {
    0x78:('mov\tr%d, 0x%x (%s)', lambda c:(c.op1/4,c.op2, repr(chr(c.op2)))),
    0xfe:('mov\tr%d, r%d', lambda c:(c.op1/4,c.op2/4)),
    0x33:('cmp\tr%d, 0x%x', lambda c:(c.op1/4,c.op2)),
    0x12:('jmp\t0x%x (0x%x, 0x%x)', lambda c:((-1 if c.op2==0x80 else 1) * c.op1 + c.ea
    +3, c.op1, c.op2)),
    0xa1:('jz \t0x%x', lambda c:((-1 if c.op2==0x80 else 1) * c.op1 + c.ea+3)),
    0xaf:('add\tr%d, r%d', lambda c:(c.op1/4, c.op2/4)),
    0x8e:('sub\tr%d, r%d', lambda c:(c.op1/4, c.op2/4)),
    0x13:('xor\tr%d, r%d', lambda c:(c.op1/4, c.op2/4)),
    0xbb:('get_key_dword\tr0, r1', lambda c:()),
    0x7c:('get_const\tr0', lambda c:()),
    0x32:('compute\tr0, r1, r2', lambda c:()),
    0x2f:('shiftmod4\tr0, r1, r2', lambda c:()),
    0xdc:('end', lambda c:()),
    0x59:('set_key_dword\tr1, r2', lambda c:()),
```

```
}

state = 0xfde7f446
Context = namedtuple('Context', ['ea', 'opcode', 'op1', 'op2'])

def disass_stream(f):
    ins = f.read(3)
    while ins:
        if ord(ins[0]) not in opcodes:
            context = Context(f.tell()-3, 0x78, ord(ins[1]), ord(ins[2]))
        else:
            context = Context(f.tell()-3, ord(ins[0]), ord(ins[1]), ord(ins[2]))

        form, values = opcodes[context.opcode]
        print colored('0x%03x' % context.ea, 'blue'), colored('%02X %02X %02X' % (ord(ins[0]), ord(ins[1]), ord(ins[2])), 'green'), colored(form % values(context), 'yellow')
        ins = f.read(3)

def get_bytecode(img):
    image = Image.open(img)
    out = ''
    sx, sy = image.size
    for y in xrange(sy):
        c = (state + (((y ^ state) + 0x42) & 0xff) ) % 3
        e = []
        b = 0
        for i in xrange(8):
            p = image.getpixel((((((y ^ state)+ 0x42) & 0xff) + (0x2a * i)), y))
            b = (b << 1) | (p[c] & 1)
            c = (c + 1) % 3
        out += chr(b)
    return out

if __name__ == '__main__':
    from sys import argv
    bytecode = get_bytecode(argv[1])
    print repr(bytecode)
    disass_stream(StringIO(bytecode))
```

Le *bytecode* décodé depuis les pixels de l'image nous donne le deuxième LUM attribué à l'épreuve :

```
$ strings vm2.dmp
LUM{C1UAidv_pzJ}M
BGRn(
```

Finalement, on obtient le code exécuté par cette seconde machine virtuelle :

```
$ python disass_vm2.py 117.bmp
0x000 78 1C 00 mov  r7, 0x0 ('\x00')
0x003 33 1C 04 cmp  r7, 0x4
0x006 A1 9C 00 jz   0xa5
0x009 FE 04 1C mov  r1, r7
```

```
0x00c AF 04 04 add  r1, r1
0x00f BB 56 47 get_key_dword    r0, r1
0x012 FE 0C 00 mov  r3, r0
0x015 78 20 01 mov  r8, 0x1 ('\x01')
0x018 AF 04 20 add  r1, r8
0x01b BB ED 11 get_key_dword    r0, r1
0x01e FE 10 00 mov  r4, r0
0x021 78 14 00 mov  r5, 0x0 ('\x00')
0x024 78 18 00 mov  r6, 0x0 ('\x00')
0x027 33 18 40 cmp  r6, 0x40
0x02a A1 57 00 jz   0x84
0x02d FE 04 10 mov  r1, r4
0x030 85 08 04 mov  r2, 0x4 ('\x04')
0x033 32 00 E5 compute  r0, r1, r2
0x036 FE 20 00 mov  r8, r0
0x039 AF 20 10 add  r8, r4
0x03c FE 04 14 mov  r1, r5
0x03f 5B 08 00 mov  r2, 0x0 ('\x00')
0x042 2F 96 48 shiftmod4    r0, r1, r2
0x045 FE 24 00 mov  r9, r0
0x048 AF 24 14 add  r9, r5
0x04b 13 20 24 xor  r8, r9
0x04e AF 0C 20 add  r3, r8
0x051 7C 88 4B get_const    r0
0x054 AF 14 00 add  r5, r0
0x057 FE 04 0C mov  r1, r3
0x05a 47 08 04 mov  r2, 0x4 ('\x04')
0x05d 32 BE 66 compute  r0, r1, r2
0x060 FE 20 00 mov  r8, r0
0x063 AF 20 0C add  r8, r3
0x066 FE 04 14 mov  r1, r5
0x069 78 08 0B mov  r2, 0xb ('\x0b')
0x06c 2F AE 58 shiftmod4    r0, r1, r2
0x06f FE 24 00 mov  r9, r0
0x072 AF 24 14 add  r9, r5
0x075 13 20 24 xor  r8, r9
0x078 AF 10 20 add  r4, r8
0x07b 78 20 01 mov  r8, 0x1 ('\x01')
0x07e AF 18 20 add  r6, r8
0x081 12 5D 80 jmp  0x27 (0x5d, 0x80)
0x084 FE 08 0C mov  r2, r3
0x087 FE 04 1C mov  r1, r7
0x08a AF 04 04 add  r1, r1
0x08d 59 D6 D3 set_key_dword    r1, r2
0x090 FE 08 10 mov  r2, r4
0x093 78 20 01 mov  r8, 0x1 ('\x01')
0x096 AF 04 20 add  r1, r8
0x099 59 D0 D6 set_key_dword    r1, r2
0x09c 78 20 01 mov  r8, 0x1 ('\x01')
0x09f AF 1C 20 add  r7, r8
0x0a2 12 A2 80 jmp  0x3 (0xa2, 0x80)
```

```
0x0a5 DC 84 D6 end
```

Concrètement, le programme scinde la clé renseignée en 8 entiers de 32 bits et dérive ces entiers deux par deux.

Le processus de dérivation peut être résummé par la fonction C suivante, où :

- k1 et k2 sont deux des entiers constituant la clé ;
- compute est la fonction du PE précédemment mentionnée ;
- boxes est un tableau d'entiers de 32 bits générés au début de l'exécution de la première machine virtuelle. Plutôt que de les calculer, on les récupère en dynamique avec *WinDbg*. Ce détail aura d'ailleurs son importance par la suite.

```c
DWORD boxes[] = {0x9aacc69b, 0x89a31d8a, 0xa5b3cd5c, 0xb0a5ce54};

void do_a_barrel_roll(DWORD k1, DWORD k2) {
  DWORD magic = 0xd2d413b3;
  DWORD r5 = 0;

  for(int i=0;i<0x40;i++) {
    k1 += (k2+compute(k2)) ^ (r5 + boxes[r5 % 4]);
    r5 += magic;
    k2 += (k1+compute(k1)) ^ (r5 + boxes[(r5 >> 0xb) % 4]);
  }
  printf("k1 = 0x%08x, k2 = 0x%08x\n", k1, k2);
}
```

Ce tableau peut être rapidement inversé de la manière suivante :

```c
DWORD resolve(DWORD k1, DWORD k2) {
  DWORD magic = 0xd2d413b3;
  DWORD r5 = magic * 0x40;

  for(int i=0;i<0x40;i++) {
    k2 -= (k1+compute(k1)) ^ (r5 + boxes[(r5 >> 0xb) % 4]);
    r5 -= magic;
    k1 -= (k2+compute(k2)) ^ (r5 + boxes[r5 % 4]);

  }
  printf("k1 = 0x%08x, k2 = 0x%08x\n", k1, k2);
}
```

On notera que dans la mesure où la fonction compute n'utilise aucun contexte lié au fonctionnement du PE, il est possible de réutiliser son code sans avoir à la *reverser*. Cette opération nécessite toutefois de corriger quelques pointeurs à cause de l'*ASLR* :

```c
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>

DWORD compute_proc = 0;
```

```
6
7  int init()
8  {
9    HMODULE unstable_machine = LoadLibraryA("unstable.machines.exe");
10   compute_proc = (DWORD)(unstable_machine) + 0x2490;
11   DWORD old_prot, i, dw;
12   DWORD r = VirtualProtect((void *)unstable_machine, 0xD000, PAGE_EXECUTE_READWRITE, &
       old_prot);
13
14   DWORD unstable_machine_dw = (DWORD) unstable_machine;
15
16   // Patch jump tables
17   for (i=0; i<5; i++) {
18     dw = *(DWORD *)(unstable_machine_dw+0x2780+4*i);
19     *(DWORD *)(unstable_machine_dw+0x2780+4*i) = dw - 0x400000 + unstable_machine_dw;
20   }
21
22   for (i=0; i<8; i++) {
23     dw = *(DWORD *)(unstable_machine_dw+0x27f8+4*i);
24     *(DWORD *)(unstable_machine_dw+0x27F8+4*i) = dw - 0x400000 + unstable_machine_dw;
25   }
26
27   // Patch ptrs
28   *(DWORD *)(unstable_machine_dw+0x24DF) = unstable_machine_dw + 0x2794;
29   *(DWORD *)(unstable_machine_dw+0x24E6) = unstable_machine_dw + 0x2780;
30   *(DWORD *)(unstable_machine_dw+0x26F1) = unstable_machine_dw + 0x2818;
31   *(DWORD *)(unstable_machine_dw+0x26F8) = unstable_machine_dw + 0x27F8;
32
33   return 0;
34 }
35
36 DWORD compute(DWORD dw) {
37   DWORD res;
38   printf("compute(0x%08x, 4) ",dw);
39   __asm {
40    mov edx, dw
41    mov ecx, 4
42    call compute_proc
43    mov res, eax
44   }
45   printf(" = 0x%08x\n", res);
46   return res;
47 }
```

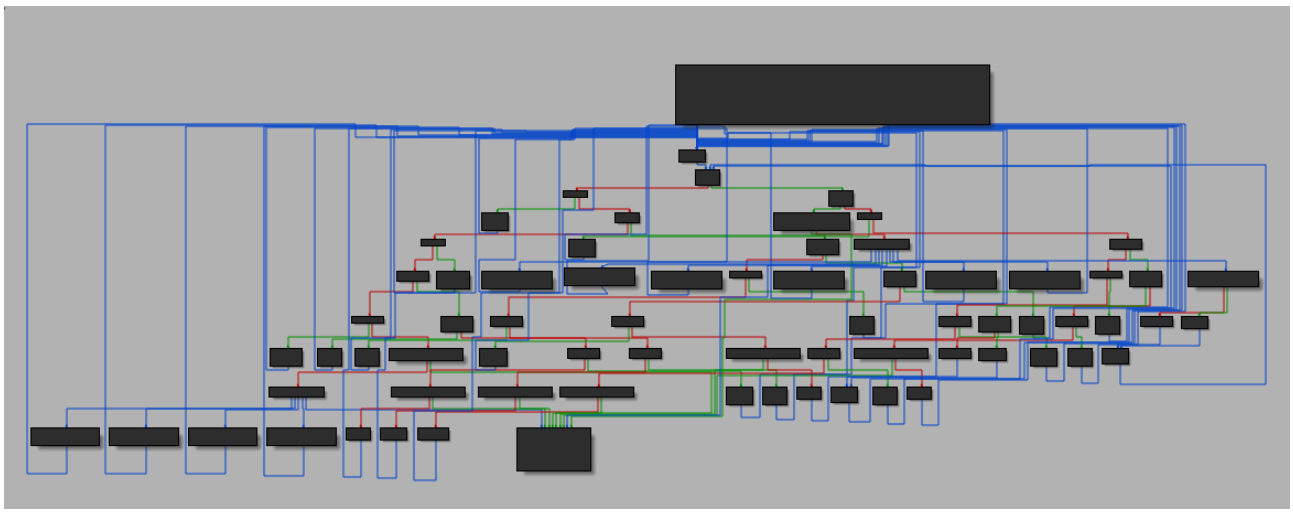


Figure 22 – CFG de la fonction compute du PE

Toutefois, en *reversant* cette fonction, on se rend compte qu'elle se résume à effectuer l'opération `(X << 4) ^ (X >> 5)` de manière obfusquée. L'utilité de réutiliser le code s'en trouve drastiquement amoindrie.

La première étape de dérivation de la clé est ainsi résolue.

5.4 Rétro-ingénierie des étape de post-traitement

Lorsque l'on continue l'exécution en dynamique après cette première étape, on atteint un appel à la fonction `Sleep`.

L'exécution est reprise par un autre *Thread*. On supposose qu'il s'agit là d'une nouvel étape de dérivation. Ce *Thread* est créé de manière légèrement cachée et obfusquée au tout début de l'exécution du programme lors de l'affichage de la fenêtre.

L'adresse de la fonction `CreateThread` est récupérée via une technique habituellement utilisée pour les *shellcodes*, c'est à dire en récupérant l'adresse de base des modules depuis le *PEB* et en calculant et comparant avec des valeurs hardcodées le condensat des noms des modules et des fonctions.

La fonctions exécutée par ce *Thread* commence à l'adresse `0x401370`.

```
void __noreturn thread_func()
{
  int *v0; // esi@2
  int *v1; // esi@4

  while ( 1 )
  {
    if ( vm2_pc == -1 )
    {
      v0 = &ROP_PAYLOAD;
      do
      {
        *v0 ^= get_magical_constant_4();
        ++v0;
      }
      while ( (signed int)v0 < (signed int)&actual_checksum );
      start_rop();
      v1 = &ROP_PAYLOAD;
      ++vm2_pc;
      do
      {
        *v1 ^= get_magical_constant_4();
        ++v1;
      }
      while ( (signed int)v1 < (signed int)&actual_checksum );
    }
    Sleep(0x64u);
  }
}
```

Figure 23 – Code décompilé de la fonction exécutée par le Thread

Globalement, toutes les 100 ms, si le flux d'exécution de la seconde machine virtuelle a atteint son terme, une charge utile en *ROP* est déchiffrée en mémoire en effectuant des opérations *XOR* bit à bit avec la constante 0xF4B02F4B, exécutée puis re-chiffrée.

Après avoir déchiffré puis écrit la charge utile dans un fichier, on écrit un désassembleur minimaliste basé sur Ropper :

```
from ropper import RopperService
from struct import unpack
from termcolor import colored

def get_gadgets_by_addr(filename, arch):
    rs = RopperService({'all':True})

    rs.addFile(filename)
    rs.setArchitectureFor(name=filename, arch=arch)
    rs.loadGadgetsFor()
    gadgets = rs.getFileFor(name=filename).gadgets
    return {g.address:g for g in gadgets}

def disass_rop(payload_filename, bin_filename):
    gadgets_64 = get_gadgets_by_addr(bin_filename, 'x86_64')
    with open(payload_filename, 'rb') as f:
        f.read(0x10) # purposely skip the first 0x10 bytes coz i know it's swtichin to 64 bits
        b = f.read(8)
```

```
        while b:
            q = unpack("Q", b)[0]
            if q in gadgets_64:
                print str(gadgets_64[q]).partition(' ')[2]
            else:
                print "0x%016x (%s)" % (q, repr(b))
            b = f.read(8)

import sys
disass_rop(sys.argv[1], sys.argv[2])
```

Ce script génère le code suivant :

```
$ python rop_disass.py rop_payload.dmp unstable.machines.exe
pop rdx; pop rcx; ret;
0x56687a2b7b4d554c ('LUM{+zhV')
0x7d713330794a7151 ('QqJy03q}')
pop rdx; pop rcx; ret;
0x756af83de1ffe263 ('c\xe2\xff\xe1=\xf8ju')
0xfa34ae1002547b89 ('\x89{T\x02\x10\xae4\xfa')
not rdx; ret;
bswap rcx; ret;
pop rax; ret;
0x00000000004140c0 ('\xc0@A\x00\x00\x00\x00\x00') <---- adresse du buffer contenant la
    clé fournie
mov eax, dword ptr [rax]; ret;
mov r8, qword ptr [rax]; ret;
xor r8, rcx; ret;
add r8, rdx; ret;
mov qword ptr [rax], r8; ret;
add rax, 8; ret;
mov r8, qword ptr [rax]; ret;
xor r8, rdx; ret;
bswap r8; ret;
sub r8, rcx; ret;
mov qword ptr [rax], r8; ret;
add rax, 8; ret;
mov r8, qword ptr [rax]; ret;
bswap r8; ret;
rol rcx, 0xd; ret;
ror r8, cl; ret;
mov qword ptr [rax], r8; ret;
add rax, 8; ret;
mov r8, qword ptr [rax]; ret;
xor r8, rcx; ret;
ror rdx, 0x15; ret;
xchg rcx, rdx; ret;
ror r8, cl; ret;
add r8, rdx; ret;
mov qword ptr [rax], r8; ret;
0x0000000000401c8d ('\x8d\x1c@\x00\x00\x00\x00\x00')
0x00000023004010cd ('\xcd\x10@\x00#\x00\x00\x00')
```

Nous obtenons au passage notre troisième et dernier LUM sur cette épreuve : LUM{+zhVQqJy03q}. En lisant rapidement le code, on comprend qu'il effectue des opérations sur les 4 entiers de 64 bits qui composent la clé. On inverse donc ces opérations.

```
rol = lambda val, r_bits, max_bits: \
    (val << r_bits%max_bits) & (2**max_bits-1) | \
    ((val & (2**max_bits-1)) >> (max_bits-(r_bits%max_bits)))

ror = lambda val, r_bits, max_bits: \
    ((val & (2**max_bits-1)) >> r_bits%max_bits) | \
    (val << (max_bits-(r_bits%max_bits)) & (2**max_bits-1))


bswap = lambda r: unpack("Q", pack("Q", r)[::-1])[0]

qw = lambda x: x & 0xffffffffffffffff

def inverse(key8):
    rdx = 0x8a9507c21e001d9c
    rcx = 0x897b540210ae34fa

    rdx = qw(ror(rdx, 0x15, 64))
    rcx = qw(rol(rcx, 0xd, 64))

    key8[3] = qw(rol(qw(key8[3] - rcx), rdx & 0xff, 64)) ^ rcx

    rdx = qw(rol(rdx, 0x15, 64))
    key8[2] = qw(bswap(rol(key8[2], rcx & 0xff, 64)))

    rcx = qw(ror(rcx, 0xd, 64))
    key8[1] = bswap(qw(key8[1] + rcx)) ^ rdx

    key8[0] = qw(key8[0] - rdx) ^ rcx

    return key8
```

Une fois cette deuxième étape de dérivation terminée, la clé obtenue est comparée à une clé de référence ayant subi une opération *XOR* bit à bit avec l'une des variables de contexte calculée lors du jeu avec les cases à cocher.

```
Toaster((int)hWnd, (__m128d)v15, a1, a2, a3, a4, a5);// HERE
real_key = some_0x20_buffer;
key = my_key;
state = ::state;
v19 = 0x1C;
_mm_storeu_si128((__m128i *)some_0x20_buffer, _mm_loadu_si128((const __m128i *)&::real_key));
_mm_storeu_si128((__m128i *)(real_key + 16), _mm_loadu_si128((const __m128i *)&real_key2));
*(_DWORD *)real_key ^= state;
*(_DWORD *)(real_key + 4) ^= state;
*(_DWORD *)(real_key + 8) ^= state;
*(_DWORD *)(real_key + 12) ^= state;
*(_DWORD *)(real_key + 16) ^= state;
*(_DWORD *)(real_key + 20) ^= state;
*(_DWORD *)(real_key + 24) ^= state;
*(_DWORD *)(real_key + 28) ^= state;
_mm_storeu_si128((__m128i *)my_key, _mm_loadu_si128((const __m128i *)key_buffer_offset));
_mm_storeu_si128((__m128i *)&my_key_2, _mm_loadu_si128((const __m128i *)(key_buffer_offset + 16)));
while ( *key == *(_DWORD *)real_key )
{
  ++key;
  real_key += 4;
  v20 = v19 < 4;
  v19 -= 4;
  if ( v20 )
  {
    MessageBoxA(hWnd, "Tu as rTussi ! Le Grand Protoon est sauvT !", "Le Magicien", 0);
    v21 = GetDlgItem(hWnd, 1);
    SetWindowTextA(v21, "Quitter");
```

Figure 24 – Extrait du code décompilé responsable de la validation de la clé

5.5 Résolution

Le plan d'action pour la résolution est le suivant :

- on récupère la clé de référence, hardcodée à l'adresse `0x4140C4` du binaire ;
- on applique un *XOR* bit à bit avec l'une des variables de context
- on applique la transformation inverse de l'étape de post-traitement en ROP ;
- on applique la transformation inverse de l'étape de dérivation effectuée dans la seconde machine virtuelle.

À la fin, on est censé récupérer une clé de 256 bits en ASCII.

On s'aperçoit toutefois que ce n'est absolument pas le cas. Après quelques heures d'arrachage de cheveux à vérifier des calculs et à chercher des *anti-debugs*, on découvre une instruction pour le moins étrange dans la générations des constantes utilisées. On rappelle que cette opération est effectuée par la première machine virtuelle.

```
compute_boxes:
mov         r1, r0
push        0x5B1C63ED
push        0x7EBB8EB3
push        0xB8AB7606
push        0x8A06FFF9
push        0xF0A21668
push        0x962E5304
push        0x36644553
push        0x40375A0
mov         r0, 8
checksum    r0
mov         [r1], r0
push        0xE9B2165A
push        0xE3A7F8B
push        0xF4C919FF
push        0xFD067499
push        0x2868793E
push        0xE2A3E528
push        0x7B1E58E8
push        0x1951D388
mov         r0, 8
checksum    r0
add         r1, 4
mov         [r1], r0
push        0x4B0B009C
push        0x32B16ACE
push        0x4B0B565E
push        0x4B0B57C1
push        0x4B0B57CE
push        0x4B0B5730
push        0xA2E81D8A
push        0x4B0B565E
push        0x4B0B5146
mov         r0, 9     ; achtuuuuuuuuuuuuuuuuung
checksum    r0
add         r1, 4
mov         [r1], r0
push        0x5B01EF00
push        0xF4BB1
push        0x5C80EB85
push        0xB602E003
push        0x508FA8D5
```

Figure 25 – Extrait du code responsable de la génération des constantes

Contrairement aux autres, le troisième entier de 32 bits est calculé en passant 9 entiers à l'*opcode checksum* qui est, lui-même, un *wrapper* sur la fonction compute_checksum à l'adresse 0x401E80. On notera d'ailleurs que les entiers poussés sur la pile ne sont pas aléatoires. En *reversant* plus en détail cette fonction, on découvre un dépassement de tampon sur la pile lorsque la taille du *BLOB* en entrée est de 0x24 octets.

```
1  unsigned int __cdecl compute_checksum(void *src, SIZE_T dwBytes)
2  {
3    HANDLE v2; // eax@3
4    char v3; // al@6
5    HANDLE v4; // eax@8
6    SIZE_T i; // [sp+0h] [bp-34h]@4
7    char *dst; // [sp+8h] [bp-2Ch]@2
8    int state; // [sp+Ch] [bp-28h]@1
9    unsigned int check_sum; // [sp+Ch] [bp-28h]@4
10   char on_the_stack; // [sp+10h] [bp-24h]@2
11
12   state = ::state;
13   if ( dwBytes > 0x24 )
14   {
15     v2 = GetProcessHeap();
16     dst = (char *)HeapAlloc(v2, 0, dwBytes);
17     memmove(dst, src, dwBytes);
18   }
19   else
20   {
21     memmove(&on_the_stack, src, dwBytes);
22     dst = &on_the_stack;
23   }
24   check_sum = state ^ get_magical_constant();
25   for ( i = 0; i < dwBytes; ++i )
26   {
27     get_magical_constant_2();
28     dst[i] ^= v3;
29     check_sum = 32 * (check_sum ^ (unsigned __int8)dst[i]) | ((check_sum ^ (unsigned __int8)dst[i]) >> 27);
30   }
31   if ( dst != &on_the_stack )
32   {
33     v4 = GetProcessHeap();
34     HeapFree(v4, 0, dst);
35   }
36   return check_sum;
37 }
```

Figure 26 – Extrait du code décompilé de la fonction vulnérable

Dans le cas de cette troisième constante, les 9 entiers utilisés sont déchiffrés et se retrouvent étrangement exécutés en tant que charge utile ROP. On les passe dans notre désassembleur :

```
1  $ python rop_disass2.py rop.dmp unstable.machines.exe
2  mov ebx, dword ptr fs:[0x30]; mov ebx, dword ptr [ebx]; not ebx; ret;
3  pop esi; add eax, esi; ret;
4  0xe9a356c1
5  xor eax, ebx; ret;
6  ror eax, 0x2a; ret;
7  sub eax, ebx; ret;
8  pop esi; add eax, esi; ret;
9  0x79fa2185
10 push ecx; ret;
```

Il s'agit d'un *anti-debug* qui modifie la constante lorsque l'attribut *BeingDebugged* du *PEB* est positionné. On détermine la vrai valeur de la troisième constante, `0xa5b2cd1c`, puis l'on corrige notre script de résolution.

```
1  dw = lambda x:x & 0xffffffff
2  qw = lambda x: x & 0xffffffffffffffff
3
4  rol = lambda val, r_bits, max_bits: \
5      (val << r_bits%max_bits) & (2**max_bits-1) | \
6      ((val & (2**max_bits-1)) >> (max_bits-(r_bits%max_bits)))
7  ror = lambda val, r_bits, max_bits: \
8      ((val & (2**max_bits-1)) >> r_bits%max_bits) | \
9      (val << (max_bits-(r_bits%max_bits)) & (2**max_bits-1))
```

```
bswap = lambda r: unpack("Q", pack("Q", r)[::-1])[0]

def inverse_step1(k1, k2):
    boxes = [0x9aacc69b, 0x89a31d8a, 0xa5b2cd1c, 0xb0a5ce54]
    magic = 0xd2d413b3

    r5 = dw(magic*0x40)

    for i in xrange(0x40):
        k2 = dw(k2 - ((k1+compute(k1)) ^ ((r5 + boxes[(r5 >> 0xb) % 4]))))
        r5 = dw(r5-magic)
        k1 = dw(k1 - ((k2+compute(k2)) ^ ((r5 + boxes[r5 % 4]))))

    return k1, k2

def do():
    key4 = [(i ^ 0xfde7f446) for i in unpack("IIIIIIII",''.join('63 93 66 D4 33 9F 03 29 A7 88 78 C9 60 1C FA CE C8 90 04 39 20 79 82 87 66 1C 84 1A F9 EE 3A 1B'.split(' ')).decode('hex'))]

    key8 = [(key4[2*i] | key4[2*i+1] << 32) for i in xrange(4)]
    rdx = 0x8a9507c21e001d9c
    rcx = 0x897b540210ae34fa

    rdx = qw(ror(rdx, 0x15, 64))
    rcx = qw(rol(rcx, 0xd, 64))

    key8[3] = qw(rol(qw(key8[3] - rcx), rdx & 0xff, 64)) ^ rcx

    rdx = qw(rol(rdx, 0x15, 64))

    key8[2] = qw(bswap(rol(key8[2], rcx & 0xff, 64)))

    rcx = qw(ror(rcx, 0xd, 64))

    key8[1] = bswap(qw(key8[1] + rcx)) ^ rdx

    key8[0] = qw(key8[0] - rdx) ^ rcx

    print ''.join(pack("II", *inverse_step1((i) & 0xffffffff, (i>>32) & 0xffffffff)) for i in key8)

do()
```

À notre plus grand bonheur, un résultat vraissemblable s'affiche enfin :

```
$ python solve.py
3f691f3d6eb60b343c931c22e0baa92f
```

Nous validons ainsi l'épreuve, débloquant la cinquième et dernière épreuve du challenge.

```
/challenges/tools $ ./add_key 3f691f3d6eb60b343c931c22e0baa92f
```

```
2 OK    !
```

6. Epreuve finale : LabyQRinth

L'épreuve finale se compose d'un seul fichier au contenu suivant :

```
Une dernière petite étape!

        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 comprend rapidement qu'il s'agit d'un QR code en ASCII. Après une tentative de scan infructueuse, on écrit un script Python pour le transformer en une image monochrome :

```python
from PIL import Image

with open('final_stripped.txt', 'r') as f:
    a = f.read()
a = a.split('\n')[:-1]

img = Image.new('1', (len(a[0]),len(a)))
```

```
for j in xrange(len(a)):
    for i in xrange(len(a[j])):
        img.putpixel((i,j), a[j][i]==' ')

img.save('qrcode.png')
```

Figure 27 – QR code obtenu

Le QR code peut maintenant être scanné. On obtient la phrase : Please use this Nibble ADD key : 5571C2017. On a maintenant une clé et une méthode de chiffrement mais on ne sait pas quoi déchiffrer.

On tente donc plusieurs d'approches sans succès :

- prendre tous les *nibbles* qui composent le QR code ;
- prendre tous les *nibbles* entre les parenthèses.

Ce n'est qu'après une bonne heure d'arrachage de cheveux que l'on se souvient du nom du challenge : **LabyQRinth**. On décide donc de récupérer les charactères utilisés dans le plus court chemin entre les parenthèses.

Ce dernier peut facilement être calculé en utilisant le module Python pathfinding. On écrit donc le script de résolution suivant :

```
from pathfinding.core.grid import Grid
from pathfinding.finder.a_star import AStarFinder
from itertools import cycle

```

```python
with open('final_stripped.txt', 'r') as f:
    a = f.read()

grid = [[int(a[i][j] == ' ') for j in xrange(33)] for i in xrange(33)]

grid = Grid(matrix=grid)

start = grid.node(0, 23)
end = grid.node(32, 26)

finder = AStarFinder()

path, runs = finder.find_path(start, end, grid)

msg = ''.join(a[j][i] for i,j in path)

key = '5571C2017'

print ''.join('%x' % (((int(c, 16) - int(k, 16)) & 0xf)) for c,k in zip(msg, cycle(key))
    ).decode('hex')
```

Et c'est ainsi que l'on récupère **enfin** l'adresse e-mail de validation, nous permettant de valider le challenge !

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

7. Remerciements

Je tiens à remercier Synacktiv pour m'avoir laissé du temps pour terminer ce challenge et toute l'équipe pour avoir supporté mes divers accès de folie et autres crises de nerf. Enfin, je remercie tout particulièrement Florian Guilbert et Eloi Vanderbeken pour leur relecture ainsi que Julien Legras pour ses conseils ô combien avisés en LaTeX, sans qui je n'aurais jamais pu rédiger cette solution.

8. Annexes : Processeurs IDA

8.1 eBPF

```
from idaapi import *
from idc import *

class DecodingError(Exception):
    pass

class EBPFProc(processor_t):
    id = 0x8000+1337
    flag = PR_ASSEMBLE | PR_SEGS | PR_DEFSEG32 | PR_USE32 | PRN_HEX | PR_RNAMESOK |
    PR_NO_SEGMOVE
    cnbits = 8
    dnbits = 8
    psnames = ['EBPF']
    plnames = ['EBPF']
    segreg_size = 0 # to patch

    instruc_start = 0
    assembler = {
        'flag':  ASH_HEXF3 | AS_UNEQU | AS_COLON | ASB_BINF4 | AS_N2CHR,
        "uflag": 0,
        "name": "Bla",
        "origin": ".org",
        "end": ".end",
        "cmnt": ";",
        "ascsep": '"',
        "accsep": "'",
        "esccodes": "\"'",
        "a_ascii": "db",
        "a_byte": "db",
        "a_word": "dw",
        'a_dword': "dd",
        'a_qword': "dq",
        "a_bss": "dfs %s",
        "a_seg": "seg",
        "a_curip": "PC",
        "a_public": "",
        "a_weak": "",
        "a_extrn": ".extern",
        "a_comdef": "",
        "a_align": ".align",
        "lbrace": "(",
        "rbrace": ")",
        "a_mod": "%",
        "a_band": "&",
```

```
44          "a_bor": "|",
45          "a_xor": "^",
46          "a_bnot": "~",
47          "a_shl": "<<",
48          "a_shr": ">>",
49          "a_sizeof_fmt": "size %s",
50
51      }
52
53      def __init__(self):
54          processor_t.__init__(self)
55
56          self.init_instructions()
57          self.init_registers()
58
59      def init_instructions(self):
60          self.OPCODES = {
61              0x07:('add', self._ana_reg_imm, CF_USE1 | CF_USE2),
62              0xbf:('mov', self._ana_2regs, CF_USE1 | CF_USE2),
63              0xb7:('mov', self._ana_reg_imm, CF_USE1 | CF_USE2),
64              0x28:('ldh', self._ana_phrase_imm, CF_USE1|CF_USE2),
65              0x55:('jne', self._ana_cond_jmp_reg_imm, CF_USE1 | CF_USE2 | CF_USE3 |
    CF_JUMP),
66              0x30:('ldb', self._ana_phrase_imm, CF_USE1|CF_USE2),
67              0x18:('lddw', self._ana_reg_imm, CF_USE1|CF_USE2),
68              0x0f:('add', self._ana_2regs, CF_USE1|CF_USE2),
69              0x67:('lsh', self._ana_reg_imm, CF_USE1|CF_USE2),
70              0x57:('and', self._ana_reg_imm, CF_USE1|CF_USE2),
71              0x1f:('sub', self._ana_2regs, CF_USE1|CF_USE2),
72              0x7b:('stxdw', self._ana_regdisp_reg, CF_USE1|CF_USE2),
73              0x85:('call', self._ana_call, CF_USE1|CF_CALL),
74              0x05:('ja', self._ana_jmp, CF_USE1|CF_JUMP),
75              0x79:('ldxdw', self._ana_reg_regdisp, CF_USE1|CF_USE2),
76              0x77:('rsh', self._ana_reg_imm, CF_USE1|CF_USE2),
77              0x50:('ldindb', self._ana_2regs, CF_USE1|CF_USE2),
78              0x40:('ldindw', self._ana_2regs, CF_USE1|CF_USE2),
79              0x48:('ldindh', self._ana_2regs, CF_USE1|CF_USE2),
80              0x4f:('or', self._ana_2regs, CF_USE1|CF_USE2),
81              0x5d:('jne', self._ana_cond_jmp_reg_reg, CF_USE1 | CF_USE2 | CF_USE3 |
    CF_JUMP),
82              0xa7:('xor', self._ana_reg_imm, CF_USE1|CF_USE2),
83              0x47:('or', self._ana_reg_imm, CF_USE1|CF_USE2),
84              0x15:('jeq', self._ana_cond_jmp_reg_imm, CF_USE1 | CF_USE2 | CF_USE3 |
    CF_JUMP),
85              0x25:('jgt', self._ana_cond_jmp_reg_imm, CF_USE1 | CF_USE2 | CF_USE3 |
    CF_JUMP),
86              0x2d:('jgt', self._ana_cond_jmp_reg_reg, CF_USE1 | CF_USE2 | CF_USE3 |
    CF_JUMP),
87              0xaf:('xor', self._ana_2regs, CF_USE1|CF_USE2),
88              0x27:('mul', self._ana_reg_imm, CF_USE1|CF_USE2),
89              0x3f:('div', self._ana_2regs, CF_USE1|CF_USE2),
```

```
            0x2f:('mul', self._ana_2regs, CF_USE1|CF_USE2),
            0x95:('ret', self._ana_nop, CF_STOP)
        }


        Instructions = [{'name':x[0], 'feature':x[2]} for x in self.OPCODES.values()]
        self.inames = {v[0]:k for k,v in self.OPCODES.items()}
        self.instruc_end = 0xff
        self.instruc = [({'name':self.OPCODES[i][0], 'feature':self.OPCODES[i][2]} if i in self.OPCODES else {'name':'unknown_opcode', 'feature':0}) for i in xrange(0xff)]

    def init_registers(self):
        self.regNames = ['r0', 'r1', 'r2', 'r3', 'r4', 'r5', 'r6', 'r7', 'r8', 'r9', 'r10', 'CS', 'DS']

        self.regFirstSreg = 0
        self.regLastSreg = 1

        self.regCodeSreg = 0
        self.regDataSreg = 1

    def ana(self):
        try:
            return self._ana()
        except DecodingError:
            return 0

    def _ana(self):
        self.opcode = ua_next_byte()
        registers = ua_next_byte()

        self.src = (registers >> 4) & 15
        self.dst = registers & 15

        self.off = ua_next_word()

        self.imm = ua_next_long()

        if self.opcode == 0x18:
            ua_next_long()
            imm2 = ua_next_long()
            self.imm += imm2 << 32


        self.cmd.itype = self.opcode

        if self.opcode not in self.OPCODES:
            raise DecodingError("wuut")

        self.OPCODES[self.opcode][1]()

```

```
139         return self.cmd.size
140
141     def _ana_nop(self):
142         pass
143
144     def _ana_reg_imm(self):
145         self.cmd[0].type = o_reg
146         self.cmd[0].dtyp = dt_dword
147         self.cmd[0].reg = self.dst
148
149         self.cmd[1].type = o_imm
150         if self.opcode == 0x18:
151             self.cmd[1].dtyp = dt_qword
152         else:
153             self.cmd[1].dtyp = dt_dword
154
155         self.cmd[1].value = self.imm
156
157
158     def _ana_2regs(self):
159         self.cmd[0].type = o_reg
160         self.cmd[0].dtyp = dt_dword
161         self.cmd[0].reg = self.dst
162
163         self.cmd[1].type = o_reg
164         self.cmd[1].dtyp = dt_dword
165         self.cmd[1].reg = self.src
166
167     def _ana_call(self):
168         self.cmd[0].type = o_imm
169         self.cmd[0].value = self.imm
170         self.cmd[0].dtyp = dt_dword
171
172     def _ana_jmp(self):
173         self.cmd[0].type = o_near
174         self.cmd[0].addr = 8*self.off + self.cmd.ea + 8
175         self.cmd[0].dtyp = dt_dword
176
177     def _ana_cond_jmp_reg_imm(self):
178         self.cmd[0].type = o_reg
179         self.cmd[0].dtyp = dt_dword
180         self.cmd[0].reg = self.dst
181
182         self.cmd[1].type = o_imm
183         self.cmd[1].value = self.imm
184         self.cmd[1].dtyp = dt_dword
185
186         self.cmd[2].type = o_near
187         self.cmd[2].addr = 8 * self.off + self.cmd.ea + 8
188         self.cmd[2].dtyp = dt_dword
189
```

```
190
191     def _ana_cond_jmp_reg_reg(self):
192         self.cmd[0].type = o_reg
193         self.cmd[0].dtyp = dt_dword
194         self.cmd[0].reg = self.dst
195
196         self.cmd[1].type = o_reg
197         self.cmd[1].dtyp = dt_dword
198         self.cmd[1].reg = self.src
199
200         self.cmd[2].type = o_near
201         self.cmd[2].addr = 8 * self.off + self.cmd.ea + 8
202         self.cmd[2].dtyp = dt_dword
203
204     def _ana_regdisp_reg(self):
205         self.cmd[0].type = o_displ
206         self.cmd[0].dtyp = dt_dword
207         self.cmd[0].value = self.off
208         self.cmd[0].phrase = self.dst
209
210         self.cmd[1].type = o_reg
211         self.cmd[1].dtyp = dt_dword
212         self.cmd[1].reg = self.src
213
214     def _ana_reg_regdisp(self):
215         self.cmd[0].type = o_reg
216         self.cmd[0].dtyp = dt_dword
217         self.cmd[0].reg = self.dst
218
219         self.cmd[1].type = o_displ
220         self.cmd[1].dtyp = dt_dword
221         self.cmd[1].value = self.off
222         self.cmd[1].phrase = self.src
223
224
225     def _ana_phrase_imm(self):
226         self.cmd[0].type = o_reg
227         self.cmd[0].dtyp = dt_dword
228         self.cmd[0].reg = self.dst
229
230         self.cmd[1].type = o_phrase
231         self.cmd[1].dtyp = dt_dword
232         self.cmd[1].value = self.imm
233
234
235     def emu(self):
236         Feature = self.cmd.get_canon_feature()
237
238         if Feature & CF_JUMP:
239             dst_op_index = 0 if self.cmd.itype == 0x5 else 2
240             ua_add_cref(self.cmd[dst_op_index].offb, self.cmd[dst_op_index].addr, fl_JN)
```

```
241             QueueSet(Q_jumps, self.cmd.ea)
242
243         if self.cmd[0].type == o_displ or self.cmd[1].type == o_displ:
244             op_ind = 0 if self.cmd[0].type == o_displ else 1
245             ua_stkvar2(self.cmd[op_ind], self.cmd[op_ind].value, 1)
246             op_stkvar(self.cmd.ea, op_ind)
247
248         # if Feature & CF_CALL:
249         #   ua_add_cref(self.cmd[0].offb, self.cmd[0].addr, fl_CN)
250
251         flow = (Feature & CF_STOP == 0) and not self.cmd.itype == 0x5
252
253         if flow:
254             ua_add_cref(0, self.cmd.ea + self.cmd.size, fl_F)
255         return True
256
257     def out(self):
258         cmd = self.cmd
259         ft = cmd.get_canon_feature()
260         buf = init_output_buffer(1024)
261         OutMnem(15)
262
263         if ft & CF_USE1:
264             out_one_operand(0)
265         if ft & CF_USE2:
266             OutChar(',')
267             OutChar(' ')
268             out_one_operand(1)
269         if ft & CF_USE3:
270             OutChar(',')
271             OutChar(' ')
272             out_one_operand(2)
273         term_output_buffer()
274         cvar.gl_comm = 1
275         MakeLine(buf)
276
277     def outop(self, op):
278         if op.type == o_reg:
279             out_register(self.regNames[op.reg])
280         elif op.type == o_imm:
281             OutValue(op, OOFW_IMM)
282         elif op.type in [o_near, o_mem]:
283             ok = out_name_expr(op, op.addr, BADADDR)
284             if not ok:
285                 out_tagon(COLOR_ERROR)
286                 OutLong(op.addr, 16)
287                 out_tagoff(COLOR_ERROR)
288                 QueueMark(Q_noName, self.cmd.ea)
289
290         elif op.type == o_phrase:
291             out_symbol('[')
```

```
            OutValue(op, OOFW_IMM)
            out_symbol(']')

        elif op.type == o_displ:
            out_symbol('[')
            out_register(self.regNames[op.phrase])
            OutValue(op, OOFS_NEEDSIGN|OOFW_IMM)
            out_symbol(']')
        else:
            return False

        return True

def PROCESSOR_ENTRY():
    return EBPFProc()
```

8.2 RiscV

```

from struct import unpack

from idaapi import *
from idc import *

class DecodingError(Exception):
    pass

def int13(v):
    if v & 0x1000:
        v -= 0x2000
    return v

def int12(v):
    if v & 0x800:
        v -= 0x1000
    return v

def int20(v):
    if v & 0x80000:
        v -= 0x100000
    return v

class RiscVProc(processor_t):
    id = 0x8000+1338
    flag = PR_ASSEMBLE | PR_SEGS | PR_DEFSEG32 | PR_USE32 | PRN_HEX | PR_RNAMESOK |
    PR_NO_SEGMOVE
    cnbits = 8
    dnbits = 8
    psnames = ['RiscV']
    plnames = ['RiscV']
```

```
32      segreg_size = 0
33
34      instruc_start = 0
35      assembler = {
36          'flag':  ASH_HEXF3 | AS_UNEQU | AS_COLON | ASB_BINF4 | AS_N2CHR,
37          "uflag": 0,
38          "name": "Bla",
39          "origin": ".org",
40          "end": ".end",
41          "cmnt": ";",
42          "ascsep": '"',
43          "accsep": "'",
44          "esccodes": "\"'",
45          "a_ascii": "db",
46          "a_byte": "db",
47          "a_word": "dw",
48          'a_dword': "dd",
49          'a_qword': "dq",
50          "a_bss": "dfs %s",
51          "a_seg": "seg",
52          "a_curip": "PC",
53          "a_public": "",
54          "a_weak": "",
55          "a_extrn": ".extern",
56          "a_comdef": "",
57          "a_align": ".align",
58          "lbrace": "(",
59          "rbrace": ")",
60          "a_mod": "%",
61          "a_band": "&",
62          "a_bor": "|",
63          "a_xor": "^",
64          "a_bnot": "~",
65          "a_shl": "<<",
66          "a_shr": ">>",
67          "a_sizeof_fmt": "size %s",
68
69      }
70
71      def __init__(self):
72          processor_t.__init__(self)
73          self.PTRSZ = 4 # Assume PTRSZ = 4 by default
74
75          self.init_instructions()
76          self.init_registers()
77
78
79      def init_instructions(self):
80          self.r_type_insts = {
81              (0x33, 0, 0) : ('add', CF_USE1|CF_USE2|CF_USE3),
82              (0x33, 0, 0x20) : ('sub', CF_USE1|CF_USE2|CF_USE3),
```

```
            (0x33, 1, 0) : ('sll', CF_USE1|CF_USE2|CF_USE3),
            (0x33, 2, 0) : ('slt', CF_USE1|CF_USE2|CF_USE3),
            (0x33, 3, 0) : ('sltu', CF_USE1|CF_USE2|CF_USE3),
            (0x33, 4, 0) : ('xor', CF_USE1|CF_USE2|CF_USE3),
            (0x33, 5, 0) : ('srl', CF_USE1|CF_USE2|CF_USE3),
            (0x33, 5, 0x20) : ('sra', CF_USE1|CF_USE2|CF_USE3),
            (0x33, 6, 0) : ('or', CF_USE1|CF_USE2|CF_USE3),
            (0x33, 7, 0) : ('and', CF_USE1|CF_USE2|CF_USE3),
        }

        self.sb_type_insts = {
            (0x63, 0) :('beq', CF_USE1|CF_USE2|CF_USE3|CF_JUMP),
            (0x63, 1) :('bne', CF_USE1|CF_USE2|CF_USE3|CF_JUMP),
            (0x63, 4) :('blt', CF_USE1|CF_USE2|CF_USE3|CF_JUMP),
            (0x63, 5) :('bge', CF_USE1|CF_USE2|CF_USE3|CF_JUMP),
            (0x63, 6) :('bltu', CF_USE1|CF_USE2|CF_USE3|CF_JUMP),
            (0x63, 7) :('bgeu', CF_USE1|CF_USE2|CF_USE3|CF_JUMP),
        }

        self.s_type_insts = {
            (0x23, 0) : ('sb', CF_USE1|CF_USE2),
            (0x23, 1) : ('sh', CF_USE1|CF_USE2),
            (0x23, 2) : ('sw', CF_USE1|CF_USE2),
        }

        self.u_type_insts = {
            (0x37,) : ('lui', CF_USE1|CF_USE2),
            (0x17,) : ('auipc', CF_USE1|CF_USE2),
        }

        self.i_type_insts = {
            (0x3, 0) : ('lb', CF_USE1|CF_USE2),
            (0x3, 1) : ('lh', CF_USE1|CF_USE2),
            (0x3, 2) : ('lw', CF_USE1|CF_USE2),
            (0x3, 4) : ('lbu', CF_USE1|CF_USE2),
            (0x3, 5) : ('lhu', CF_USE1|CF_USE2),

            (0x13, 0) : ('addi', CF_USE1|CF_USE2|CF_USE3),
            (0x13, 1) : ('slli', CF_USE1|CF_USE2|CF_USE3),
            (0x13, 2) : ('slti', CF_USE1|CF_USE2|CF_USE3),
            (0x13, 3) : ('sltiu', CF_USE1|CF_USE2|CF_USE3),
            (0x13, 4) : ('xori', CF_USE1|CF_USE2|CF_USE3),
            (0x13, 5) : ('srli', CF_USE1|CF_USE2|CF_USE3),
            # FIXME srai is not properly displayed funct3 = 5 but imm < 0
            (0x13, 6) : ('ori', CF_USE1|CF_USE2|CF_USE3),
            (0x13, 7) : ('andi', CF_USE1|CF_USE2|CF_USE3),
        }

        self.misc_type_insts = {
            (0x6f, 0):('jal', CF_USE1|CF_USE2|CF_CALL),
            (0x6f, 1):('j', CF_USE1|CF_JUMP),
```

```
            (0x67, 0):('jr', CF_USE1|CF_JUMP),
            (0x67, 1):('ret', CF_STOP),
            (0x73, 0):('ecall', 0)
        }

        self.all_insts_types = [self.r_type_insts, self.sb_type_insts, self.s_type_insts, self.u_type_insts, self.i_type_insts, self.misc_type_insts]


        self.opcode_types = {
            k[0]:t for t in self.all_insts_types for k in t
        }

        self.instruc = [{'name':t[i][0], 'feature':t[i][1]} for t in self.all_insts_types for i in t]

        self.instruc_end = len(self.instruc)

        # by name
        self.inames = self.itypes = {v['name']:k for k,v in enumerate(self.instruc)}

    def init_registers(self):
        self.regNames = ['0', 'ra', 'sp', 'x3', 'x4', 't0', 't1', 't2', 's0', 's1', 'a0', 'a1', 'a2', 'a3', 'a4', 'a5', 'a6', 'a7', 's2', 'a3', 's4', 's5', 's6', 's7', 'x24', 'x25', 'x26', 'x27', 't3', 't4', 't5', 't6'] + ['CS', 'DS']

        self.reg_ids = {v:k for k,v in enumerate(self.regNames)}

        self.regFirstSreg = self.reg_ids["CS"]
        self.regCodeSreg = self.reg_ids["CS"]
        self.regLastSreg = self.reg_ids["DS"]
        self.regDataSreg = self.reg_ids["DS"]

    def ana(self):
        try:
            return self._ana()
        except DecodingError:
            return 0

    def _ana(self):
        self.dword = ua_next_long()

        self.opcode = self.dword & 0x7f
        self.imm_4_1_11 = self.imm_4_0 = self.rd = (self.dword >> 7) & 0x1f
        self.funct3 = (self.dword >> 12) & 0x7
        self.rs1 = (self.dword >> 15) & 0x1f
        self.rs2 = (self.dword >> 20) & 0x1f
        self.imm_12_10_5 = (self.dword >> 25) & 0x7f
        self.imm_11_5 = (self.dword >> 25) & 0x7f
        self.funct7 = (self.dword >> 25) & 0x7f

```

```
        ana_funcs = [self._ana_r_type, self._ana_sb_type, self._ana_s_type, self.
    _ana_u_type, self._ana_i_type, self._ana_misc_type]

        ana_func = None

        for i, insts in enumerate(self.all_insts_types):
            for j in insts:
                if j[0]==self.opcode:
                    ana_func = ana_funcs[i]
                    break
            if ana_func:
                break

        else:
            raise DecodingError('unknown opcode 0x%x' % (self.opcode))

        print 'opcode = 0x%x, ana_func = %s' % (self.opcode, ana_func)

        ana_func()

        self.cmd.itype = self.itypes[self.mnemonic]

        return self.cmd.size


    def _ana_r_type(self):
        self.mnemonic, features = self.r_type_insts[(self.opcode, self.funct3, self.
    funct7)]

        self.cmd[0].type = o_reg
        self.cmd[0].dtyp = dt_dword
        self.cmd[0].value = self.rd

        self.cmd[1].type = o_reg
        self.cmd[1].dtyp = dt_dword
        self.cmd[1].value = self.rs1

        self.cmd[2].type = o_reg
        self.cmd[2].dtyp = dt_dword
        self.cmd[2].value = self.rs2


    def _ana_i_type(self):
        self.mnemonic, features = self.i_type_insts[(self.opcode, self.funct3)]

        if self.opcode == 0x3:
            self.cmd[0].type = o_reg
            self.cmd[0].dtyp = dt_dword
            self.cmd[0].value = self.rd

            self.cmd[1].type = o_displ
```

```
230             self.cmd[1].dtyp = dt_dword
231             self.cmd[1].value = int12((self.dword >> 20) & 0xfff)
232             self.cmd[1].phrase = self.rs1
233
234         else:
235             self.cmd[0].type = o_reg
236             self.cmd[0].dtyp = dt_dword
237             self.cmd[0].value = self.rd
238
239             self.cmd[1].type = o_reg
240             self.cmd[1].dtyp = dt_dword
241             self.cmd[1].value = self.rs1
242
243             self.cmd[2].type = o_imm
244             self.cmd[2].dtyp = dt_dword
245             self.cmd[2].value = int12((self.dword >> 20) & 0xfff)
246
247
248
249     def _ana_s_type(self):
250         self.mnemonic, features = self.s_type_insts[(self.opcode, self.funct3)]
251
252         # for store ops
253         self.cmd[0].type = o_reg
254         self.cmd[0].dtyp = dt_dword
255         self.cmd[0].value = self.rs2
256
257         self.cmd[1].type = o_displ
258         self.cmd[1].dtyp = dt_dword
259         self.cmd[1].value = self.imm_4_0 + (self.imm_11_5 << 5)
260         self.cmd[1].phrase = self.rs1
261
262
263     def _ana_sb_type(self):
264         self.mnemonic, features = self.sb_type_insts[(self.opcode, self.funct3)]
265
266         # cond branches
267         self.cmd[0].type = o_reg
268         self.cmd[0].dtyp = dt_dword
269         self.cmd[0].value = self.rs1
270
271         self.cmd[1].type = o_reg
272         self.cmd[1].dtyp = dt_dword
273         self.cmd[1].value = self.rs2
274
275         self.cmd[2].type = o_near
276         self.cmd[2].dtyp = dt_dword
277         self.cmd[2].addr = int13((((self.dword >> 8) & 0xf) << 1) + (((self.dword >> 25)
    & 0x3f) << 5) + (((self.dword >> 7) & 1) << 11) + (((self.dword >> 31) & 1) << 12)) +
     self.cmd.ea
278
```

```
    def _ana_u_type(self):
        self.mnemonic, features = self.u_type_insts[(self.opcode,)]

        # upper 12 bits
        self.cmd[0].type = o_reg
        self.cmd[0].dtyp = dt_dword
        self.cmd[0].value = self.rd

        self.cmd[1].type = o_imm
        self.cmd[1].dtyp = dt_dword
        self.cmd[1].value = ((self.dword >> 12) << 12)


    def _ana_uj_type(self):
        self.mnemonic, features = self.uj_type_insts[(self.opcode, self.funct3, self.funct_7)]

        # for jumps
        self.cmd[0].type = o_reg
        self.cmd[0].dtyp = dt_dword
        self.cmd[0].value = self.rd

        self.cmd[1].type = o_near
        self.cmd[1].dtyp = dt_dword
        self.cmd[1].addr = (-1 if ((self.dword >> 31) & 1) else 1) * (
            (((self.dword >> 21) & 0x3ff) << 1) + (((self.dword >> 20) & 1) << 11) + (((self.dword >> 12) & 0xff) << 12)
        ) + self.cmd.ea


    def _ana_misc_type(self):
        # for jal and jalr and ecall

        # jal
        if self.opcode == 0x6f:
            # case jal -> j
            if self.rd == 0:
                spec = 1

                print 'in j',hex(self.dword)
                self.cmd[0].type = o_near
                self.cmd[0].dtyp = dt_dword
                self.cmd[0].addr = self.cmd.ea + int20((((self.dword >> 21) & 0x3ff) << 1) + (((self.dword >> 20) & 1) << 11) + (((self.dword >> 12) & 0xff) << 12))
                print 'in j addr = ',hex(self.cmd[0].addr)


            # jal -> call
            else:
                spec = 0
```

```
327                 print 'in jal (call)',hex(self.dword)
328
329                 # LR
330                 self.cmd[0].type = o_reg
331                 self.cmd[0].dtyp = dt_dword
332                 self.cmd[0].value = self.rd
333
334                 self.cmd[1].type = o_near
335                 self.cmd[1].dtyp = dt_dword
336                 self.cmd[1].addr = self.cmd.ea + int20((((self.dword >> 21) & 0x3ff) <<
    1) + (((self.dword >> 20) & 1) << 11) + (((self.dword >> 12) & 0xff) << 12))
337
338
339         elif self.opcode == 0x67:
340
341             # jalr = ret
342             if self.rs1 == 1 or self.rs1 == 0:
343                 spec = 1
344
345             elif self.dword >> 20 == 0:
346                 spec = 0
347                 self.cmd[0].type = o_reg
348                 self.cmd[0].dtyp = dt_dword
349                 self.cmd[0].value = self.rs1
350
351             else:
352                 raise DecodingError("some error occured with jalr instruction")
353         elif self.opcode == 0x73:
354             spec = 0
355
356         self.mnemonic, _ = self.misc_type_insts[(self.opcode, spec)]
357
358
359     def emu(self):
360         Feature = self.cmd.get_canon_feature()
361
362         if Feature & CF_JUMP:
363             for i in xrange(3):
364                 if self.cmd[i].type == o_near:
365                     ua_add_cref(self.cmd[i].offb, self.cmd[i].addr, fl_JN)
366                     QueueSet(Q_jumps, self.cmd.ea)
367                     break
368
369         if Feature & CF_CALL:
370             ua_add_cref(self.cmd[1].offb, self.cmd[1].addr, fl_CN)
371
372         flow = (Feature & CF_STOP == 0) and not self.instruc[self.cmd.itype]['name'] in [
    'j', 'jr']
373
374         if flow:
375             ua_add_cref(0, self.cmd.ea + self.cmd.size, fl_F)
```

```
376         return True
377
378     def out(self):
379         cmd = self.cmd
380
381         ft = cmd.get_canon_feature()
382         buf = init_output_buffer(1024)
383         OutMnem(15)
384
385         if ft & CF_USE1:
386             out_one_operand(0)
387         if ft & CF_USE2:
388             OutChar(',')
389             OutChar(' ')
390             out_one_operand(1)
391         if ft & CF_USE3:
392             OutChar(',')
393             OutChar(' ')
394             out_one_operand(2)
395         term_output_buffer()
396         cvar.gl_comm = 1
397         MakeLine(buf)
398
399
400     def outop(self, op):
401         if op.type == o_reg:
402             out_register(self.regNames[op.value])
403         elif op.type == o_imm:
404             print op.value
405             OutValue(op, OOF_SIGNED)
406         elif op.type in [o_near, o_mem]:
407
408             print 'output o_near = 0x%x' % (op.addr)
409             ok = out_name_expr(op, op.addr, BADADDR)
410             if not ok:
411                 out_tagon(COLOR_ERROR)
412                 OutLong(op.addr, 16)
413                 out_tagoff(COLOR_ERROR)
414                 QueueMark(Q_noName, self.cmd.ea)
415
416         elif op.type == o_phrase:
417             out_symbol('[')
418             print "in out phrase op.value=",repr(op.value)
419             OutValue(op, OOFW_IMM)
420             out_symbol(']')
421
422         elif op.type == o_displ:
423             out_symbol('[')
424             out_register(self.regNames[op.phrase])
425             OutValue(op, OOFS_NEEDSIGN|OOFW_IMM)
426             out_symbol(']')
```

```
        else:
            return False
        return True

def PROCESSOR_ENTRY():
    return RiscVProc()
```

8.3 VM SSTIC

```
import sys
import idaapi
import json
from struct import unpack

from idaapi import *
from idc import *

class DecodingError(Exception):
    pass

def int11(x):
    if x & 0x400:
        x = -1 * (x & 0x3ff)

    return x

class SSTICVM(processor_t):
    id = 0x8000+1339
    flag = PR_ASSEMBLE | PR_SEGS | PR_DEFSEG32 | PR_USE32 | PRN_HEX | PR_RNAMESOK |
    PR_NO_SEGMOVE
    cnbits = 8
    dnbits = 8
    psnames = ['SSTICVM']
    plnames = ['SSTICVM']

    instruc_start = 0
    assembler = {
        'flag':  ASH_HEXF3 | AS_UNEQU | AS_COLON | ASB_BINF4 | AS_N2CHR,
        "uflag": 0,
        "name": "Bla",
        "origin": ".org",
        "end": ".end",
        "cmnt": ";",
        "ascsep": '"',
        "accsep": "'",
        "esccodes": "\"'",
        "a_ascii": "db",
        "a_byte": "db",
        "a_word": "dw",
        'a_dword': "dd",
```

```
41          'a_qword': "dq",
42          "a_bss": "dfs %s",
43          "a_seg": "seg",
44          "a_curip": "PC",
45          "a_public": "",
46          "a_weak": "",
47          "a_extrn": ".extern",
48          "a_comdef": "",
49          "a_align": ".align",
50          "lbrace": "(",
51          "rbrace": ")",
52          "a_mod": "%",
53          "a_band": "&",
54          "a_bor": "|",
55          "a_xor": "^",
56          "a_bnot": "~",
57          "a_shl": "<<",
58          "a_shr": ">>",
59          "a_sizeof_fmt": "size %s",
60
61      }
62
63      def __init__(self):
64          processor_t.__init__(self)
65          self.init_instructions()
66          self.init_registers()
67
68      def init_instructions(self):
69          self.OPCODES = {
70              0xe8:('mov', self._ana_mov, CF_USE1 | CF_USE2),
71              0xb0:('add', self._ana_arith_op, CF_USE1 | CF_USE2),
72              0x80:('sub', self._ana_arith_op, CF_USE1 | CF_USE2),
73              0x68:('xor', self._ana_arith_op, CF_USE1 | CF_USE2),
74              0x18:('call', self._ana_offset, CF_USE1 | CF_CALL),
75              0xA0:('jmp', self._ana_offset, CF_USE1 | CF_JUMP),
76              0x30:('beq', self._ana_cond_jmp, CF_USE1|CF_USE2|CF_USE3|CF_JUMP), # imm
77              0x58:('ret', self._ana_nop, CF_STOP),
78              0xc8:('push', self._ana_push, CF_USE1),
79              0xe0:('shl', self._ana_shift, CF_USE1|CF_USE2),
80              0xa8:('shr', self._ana_shift, CF_USE1|CF_USE2),
81              0x50:('binary_op_chelou', self._ana_misc, CF_USE1|CF_USE2),
82              0x78:('checksum', self._ana_reg0, CF_USE1),
83              0x90:('pixels', self._ana_reg0, CF_USE1),
84              0x28:('compute_something', self._ana_reg0, CF_USE1),
85              0x00:('add y', self._ana_reg0, CF_USE1),
86              0x48:('exit', self._ana_nop, CF_STOP)
87          }
88
89
90          self.inames = {v[0]:k for k,v in self.OPCODES.items()}
91          self.instruc_end = 0xff
```

```
92          self.instruc = [({'name':self.OPCODES[i][0], 'feature':self.OPCODES[i][2]} if i
    in self.OPCODES else {'name':'unknown_opcode', 'feature':0}) for i in xrange(0xff)]
93
94          self.icode_return = 0x58
95
96      def init_registers(self):
97          self.regNames = ['r0', 'r1', 'r2', 'r3', 'r4', 'r5', 'r6', 'sp', 'CS', 'DS']
98
99          self.regFirstSreg = 0
100         self.regLastSreg = 1
101
102         self.regCodeSreg = 0
103         self.regDataSreg = 1
104
105     def ana(self):
106         try:
107             return self._ana()
108         except DecodingError:
109             return 0
110
111     def _ana(self):
112         self.b = ua_next_byte()
113
114         opcode = self.b & 0xf8
115
116         print hex(opcode)
117
118         if self.b in self.OPCODES:
119             self.OPCODES[self.b][1]()
120             self.cmd.itype = self.b
121         elif opcode in self.OPCODES:
122             self.OPCODES[opcode][1]()
123             self.cmd.itype = opcode
124         else:
125             raise DecodingError('asd')
126
127         return self.cmd.size
128
129     def _ana_nop(self):
130         pass
131
132     def _ana_reg_imm(self):
133         dword = ua_next_long()
134
135         self.cmd[0].type = o_reg
136         self.cmd[0].dtyp = dt_dword
137         self.cmd[0].reg = dword >> 29
138
139         self.cmd[1].type = o_imm
140         self.cmd[1].dtyp = dt_dword
141         self.cmd[1].value = dword & 0x1FFFFFFF
```

```
142
143     def _ana_2regs(self):
144         b = ua_next_byte()
145
146         self.cmd[0].type = o_reg
147         self.cmd[0].dtyp = dt_dword
148         self.cmd[0].reg = (b >> 4) & 7
149
150         self.cmd[1].type = o_reg
151         self.cmd[1].dtyp = dt_dword
152         self.cmd[1].reg = (b >> 1) & 7
153
154
155     def _ana_mov(self):
156         if self.b & 3 == 0:
157             self._ana_reg_imm()
158         elif self.b & 3 == 1:
159             self._ana_2regs()
160         elif self.b & 3 == 2:
161             self._ana_phrase_reg()
162         elif self.b & 3 == 3:
163             self._ana_reg_phrase()
164
165     def _ana_arith_op(self):
166         if (self.b >> 1) & 1:
167             self._ana_2regs()
168         else:
169             self._ana_reg_imm()
170
171     def _ana_phrase_reg(self):
172         b = ua_next_byte()
173
174         self.cmd[0].type = o_phrase
175         self.cmd[0].dtyp = dt_dword
176         self.cmd[0].reg = (b >> 4) & 7
177
178         self.cmd[1].type = o_reg
179         self.cmd[1].dtyp = dt_dword
180         self.cmd[1].reg = (b >> 1) & 7
181
182     def _ana_reg_phrase(self):
183         b = ua_next_byte()
184
185         self.cmd[0].type = o_reg
186         self.cmd[0].dtyp = dt_dword
187         self.cmd[0].reg = (b >> 4) & 7
188
189         self.cmd[1].type = o_phrase
190         self.cmd[1].dtyp = dt_dword
191         self.cmd[1].reg = (b >> 1) & 7
192
```

```
193     def _ana_offset(self):
194         imm = int11(((self.b & 7) << 8) + ua_next_byte())
195 
196         self.cmd[0].type = o_near
197         self.cmd[0].addr = self.cmd.ea + imm + self.cmd.size
198         self.cmd[0].dtyp = dt_dword
199 
200     def _ana_cond_jmp(self):
201         dword = ua_next_long()
202 
203         self.cmd[0].type = o_reg
204         self.cmd[0].dtyp = dt_dword
205         self.cmd[0].reg = dword >> 29
206 
207         if self.b & 1 == 1:
208             # 2 regs + offset
209             self.cmd[1].type = o_reg
210             self.cmd[1].dtyp = dt_dword
211             self.cmd[1].reg = (dword >> 18) & 7
212 
213             self.cmd[2].type = o_near
214             self.cmd[2].addr = self.cmd.ea + int11((dword >> 6) & 0x7ff) + self.cmd.size
    # FIXME a voir si on a besoin de + addr len
215             self.cmd[2].dtyp = dt_dword
216 
217         else:
218             self.cmd[1].type = o_imm
219             self.cmd[1].dtyp = dt_dword
220             self.cmd[1].value = (dword >> 19) & 0xff
221 
222             self.cmd[2].type = o_near
223             self.cmd[2].addr = self.cmd.ea + int11((dword >> 6) & 0x7ff) + self.cmd.size
224             self.cmd[2].dtyp = dt_dword
225 
226     def _ana_push(self):
227         if (self.b >> 2) & 1:
228             self.cmd[0].type = o_reg
229             self.cmd[0].reg = (ua_next_byte() >> 3) & 7
230             self.cmd[0].dtyp = dt_dword
231 
232         else:
233             self.cmd[0].type = o_imm
234             self.cmd[0].value = ua_next_long()
235             self.cmd[0].dtyp = dt_dword
236 
237     def _ana_shift(self):
238         self.cmd[0].type = o_reg
239         self.cmd[0].reg = self.b & 7
240         self.cmd[0].dtyp = dt_dword
241 
242         self.cmd[1].type = o_imm
```

```
243         self.cmd[1].reg = (ua_next_byte() >> 3) & 7
244         self.cmd[1].dtyp = dt_dword
245 
246     def _ana_misc(self):
247         self.cmd[0].type = o_reg
248         self.cmd[0].reg = self.b & 7
249         self.cmd[0].dtyp = dt_dword
250 
251         self.cmd[1].type = o_imm
252         self.cmd[1].value = ua_next_byte()
253         self.cmd[1].dtyp = dt_byte
254 
255     def _ana_reg0(self):
256         self.cmd[0].type = o_reg
257         self.cmd[0].reg = 0
258         self.cmd[0].dtyp = dt_dword
259 
260     def emu(self):
261         Feature = self.cmd.get_canon_feature()
262 
263         if Feature & CF_JUMP:
264             dst_op_index = 0 if self.cmd.itype == 0xA0 else 2
265             ua_add_cref(self.cmd[dst_op_index].offb, self.cmd[dst_op_index].addr, fl_JN)
266             QueueSet(Q_jumps, self.cmd.ea)
267 
268         if Feature & CF_CALL:
269             ua_add_cref(self.cmd[0].offb, self.cmd[0].addr, fl_CN)
270 
271         flow = (Feature & CF_STOP == 0) and not self.cmd.itype == 0xA0
272 
273         if flow:
274             ua_add_cref(0, self.cmd.ea + self.cmd.size, fl_F)
275         return True
276 
277     def out(self):
278         cmd = self.cmd
279         ft = cmd.get_canon_feature()
280         buf = init_output_buffer(1024)
281         OutMnem(15)
282 
283         if ft & CF_USE1:
284             out_one_operand(0)
285         if ft & CF_USE2:
286             OutChar(',')
287             OutChar(' ')
288             out_one_operand(1)
289         if ft & CF_USE3:
290             OutChar(',')
291             OutChar(' ')
292             out_one_operand(2)
293         term_output_buffer()
```

```
294         cvar.gl_comm = 1
295         MakeLine(buf)
296
297     def outop(self, op):
298         if op.type == o_reg:
299             out_register(self.regNames[op.reg])
300         elif op.type == o_imm:
301             OutValue(op, OOFW_IMM)
302         elif op.type in [o_near, o_mem]:
303             ok = out_name_expr(op, op.addr, BADADDR)
304             if not ok:
305                 out_tagon(COLOR_ERROR)
306                 OutLong(op.addr, 16)
307                 out_tagoff(COLOR_ERROR)
308                 QueueMark(Q_noName, self.cmd.ea)
309         elif op.type == o_phrase:
310             out_symbol('[')
311             out_register(self.regNames[op.reg])
312             out_symbol(']')
313         elif op.type == o_displ:
314             out_symbol('[')
315             out_register(self.regNames[op.phrase])
316             OutValue(op, OOFS_NEEDSIGN|OOFW_IMM)
317             out_symbol(']')
318         else:
319             return False
320         return True
321
322 def PROCESSOR_ENTRY():
323     return SSTICVM()
```