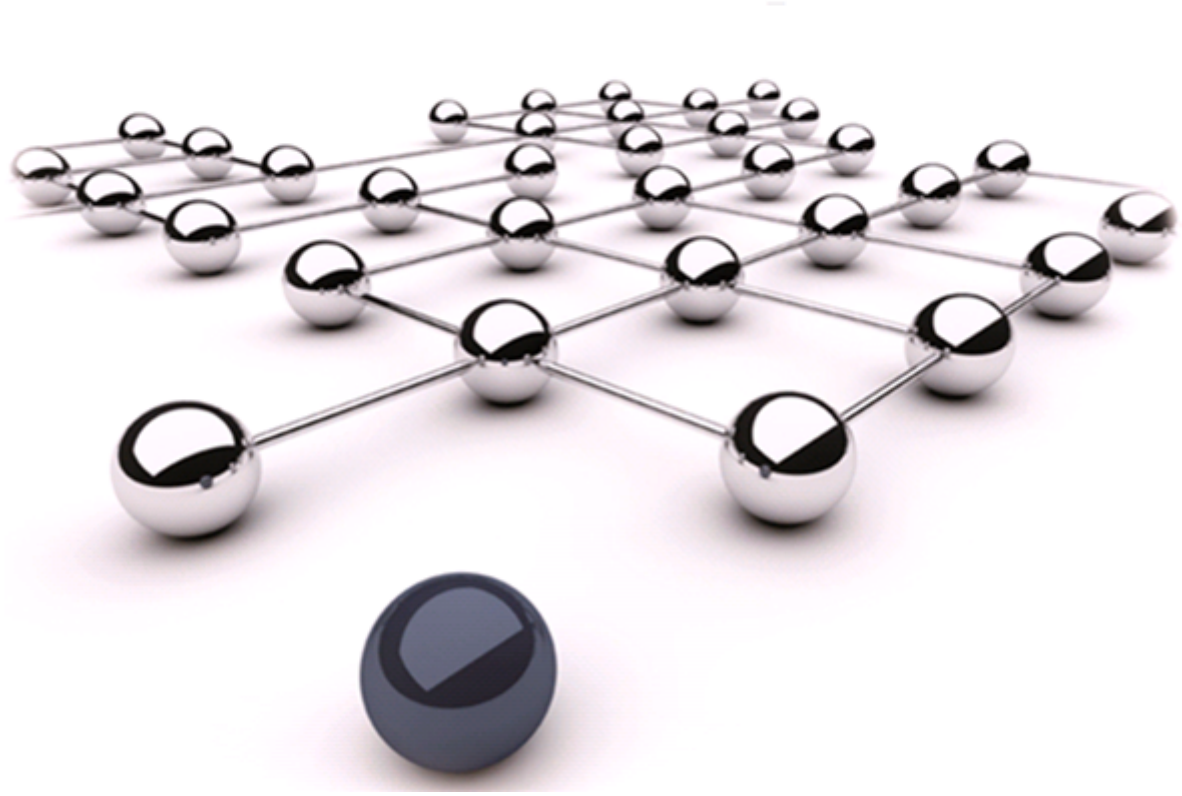
OPPIDA

EXPERT EN SÉCURITÉ
DES SYSTÈMES D'INFORMATION



Challenge SSTIC 2017

Auteur :
Damien Millescamps

Référence : OPPIDA/DOC/2017/INT/699/1.0

11/04/2017

Table des matières

1 Introduction . . . 2
2 Electronic Flash . . . 3
3 Don't Let Him Escape ! . . . 6
3.1 Clé LUM . . . 6
3.2 Clé de l'épreuve . . . 8
3.3 Résolution rapide . . . 11
4 Riscy Zones . . . 12
4.1 Commandes de la Trusted Application . . . 13
4.2 Première clé LUM . . . 14
4.3 Format du séquestre de clé . . . 15
4.4 Seconde clé LUM . . . 18
4.5 Signature de la Trusted Application . . . 18
4.6 Fonction de déchiffrement . . . 18
4.7 Inversion du keystream . . . 20
4.8 Clé de l'épreuve . . . 21
4.9 Troisième clé LUM . . . 22
5 Unstable Machines . . . 23
5.1 Premier LUM . . . 26
5.2 Première machine virtuelle . . . 26
5.3 Jeu d'instruction de la première VM . . . 29
5.4 Code assembleur de la première VM . . . 32
5.5 Instruction de hachage et cookie de sécurité . . . 34
5.6 Première ROP-chain . . . 35
5.7 Stéganographie . . . 36
5.8 Second LUM . . . 36
5.9 Seconde machine virtuelle . . . 37
5.10 Automate fini et SSE4.2 . . . 37
5.11 Inversion du code de la deuxième VM . . . 41
5.12 Second thread d'exécution . . . 41
5.13 Troisième LUM . . . 42
5.14 Deuxième ROP-chain . . . 42
5.15 Clé de l'épreuve . . . 44
6 LabyQRinth . . . 45

1 Introduction

Comme chaque année, le principe du challenge reste le même :

« Le défi consiste à extraire les données d'une trace d'analyseur logique, puis à résoudre les épreuves suivantes. L'objectif est d'y retrouver une adresse e-mail (...@sstic.org). »

Cette année, des clés optionnelles sont présentes dans les épreuves et permettent de valider la direction prise pour la recherche de la solution :

« Le challenge contient des clés optionnelles (LUM), il n'est pas nécessaire de les trouver toutes pour obtenir l'adresse email. »

Le défi comporte 4 épreuves, la résolution de la première permettant d'obtenir le code d'un émulateur OpenRISC (OR1k) en JavaScript faisant tourner un Linux sur lequel se trouvent les épreuves suivantes. La dernière épreuve étant chiffrée, il faut résoudre les épreuves intermédiaires afin de la débloquer.

Le fichier du challenge se trouve à l'adresse : `http://static.sstic.org/challenge2017/Bittendo_email_leak_by_shadowbrokers.zip`.

Le fichier ZIP contient un email. Après l'avoir ouvert avec son client mail favori, on peut récupérer les deux pièces jointes :
- NAND_FLASH_writes_no_OOB_5MHz.sr
- NAND_pinout.jpg

Le fichier `NAND_FLASH_writes_no_OOB_5MHz.sr` est une trace d'analyseur logique au format `sigrok` et peut se visualiser à l'aide de `PulseView`.

L'ensemble du code présenté ici pourra être mis à disposition. Probablement par le biais de `github`.

2 Electronic Flash

La trace correspond à la programmation d'une flash NAND classique parallèle. De la documentation sur ce type de flash peut se trouver à l'adresse : `https://www.micron.com/~/media/documents/products/technical-note/nand-flash/tn2919_nand_101.pdf`.

La trace comporte 12 signaux, le rôle de ces signaux est le suivant :

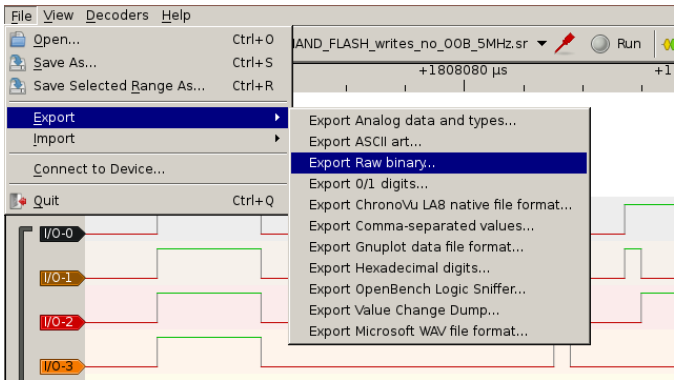
Nom	Nom complet	Rôle
I/O-{0..7}	Input/Output	Bus d'entrée/sortie 8 bits
WE	Write Enable	Horloge sur front montant pour l'écriture de données, de commande ou d'adresse
RE	Read Enable	"Chip enable don't care" / Horloge sur front montant pour la lecture de données
ALE	Adddress Latch Enable	I/O-{0..7} représente un octet d'adresse
CLE	Command Latch Enable	I/O-{0..7} représente un octet de commande

Seules certaines commandes sont utilisées lors de la programmation :

Commande	Commande 1	Octets d'adresse	Données	Commande 2
READ ID	90h	1	No	-
PROGRAM PAGE	80h	5	Yes	10h
ERASE BLOCK	60h	3	No	D0h
RESET	FFh	-	No	-

Il est possible d'écrire un décodeur `sigrok` pour ce type de flash, cependant la stratégie de décodage utilisée pose deux soucis : tous les échantillons sont conservés en mémoire lors du décodage, nécessitant bien plus de RAM que ce qui est normalement disponible sur une machine récente, et le fichier d'entrée est lu octet par octet rendant le décodage extrêmement lent.

Une solution plus efficace consiste à convertir le fichier au format brut ("raw binary"), puis à le décoder dans un language adapté pour ce type de tâche :



Le format brut contient chaque échantillon codé selon la structure suivante, sans données supplémentaires :

```
struct pinout {
    unsigned char io;
    unsigned char cle:1;
    unsigned char ale:1;
    unsigned char we:1;
    unsigned char re:1;
};
```

Le décodage peut se faire en lisant le fichier par blocs de 4096 octets, représentant 2048 échantillons, limitant ainsi le nombre d'opérations d'entrée/sortie nécessaires au décodage. Il est important de correctement décoder l'adresse car la flash est programmée dans le désordre.
La flash est découpée en blocs de 2MB divisés en 256 pages de 8kB. Une fois les 17 blocs reconstitués, on obtient une image au format VFAT. L'auteur de cet épreuve a visiblement choisi de ne pas trop compliquer la tâche avec une image au format UBI, généralement utilisé pour les flash NAND.

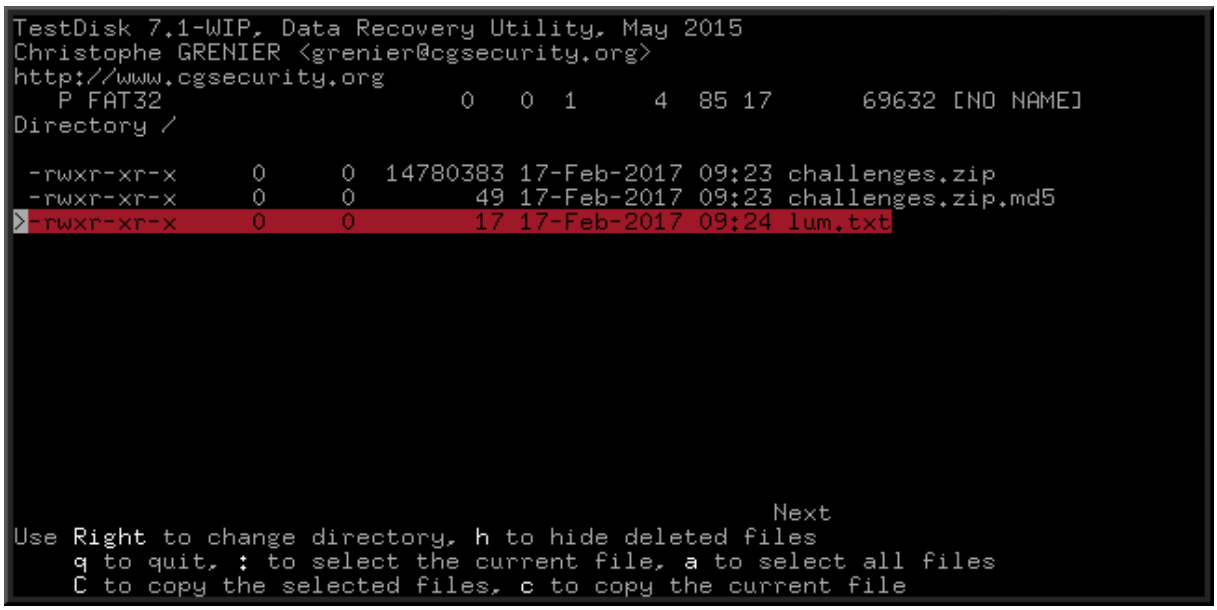
```
$ ./decode nand_flash.raw electronic_flash
> Reset (C=FF)
> Read Id (C=90 A=00)
< SSTIC PSEUDO NAND   LUM{x.g215WiPCR}
> Block Erase (C=60 A=000000 C=D0)
.............................................................
> Block Erase (C=60 A=000800 C=D0)
.............................................................

[...]

> Block Erase (C=60 A=000300 C=D0)
.............................................................
$ file electronic_flash
electronic_flash: DOS/MBR boot sector, code offset 0x58+2, OEM-ID "mkfs.fat", Media descriptor 0
    xf8, sectors/track 63, heads 255, sectors 67584 (volumes > 32 MB) , FAT (32 bit), sectors/
    FAT 520, serial number 0xf0fa166b, unlabeled
```

On trouve notre premier LUM dans l'identifiant de la flash : `LUM{x.g215WiPCR}`. On peut alors récupérer les fichiers (effacés ou non) présents sur l'image VFAT à l'aide de la commande `testdisk` :

```
$ testdisk electronic_flash
```



Ce qui permet de finaliser l'épreuve et d'obtenir un second LUM : `LUM{AsPBdVWz95y}` :

```
$ cat lum.txt
LUM{AsPBdVWz95y}
$ cat challenges.zip.md5
3977e2084331bb1c52abb115a332c6cc  challenges.zip
$ md5sum challenges.zip
3977e2084331bb1c52abb115a332c6cc  challenges.zip
```

Après avoir suivi les instructions pour la mise en place du jeu, on obtient cette interface :

3 Don't Let Him Escape !

L'épreuve contient deux scripts python ainsi qu'une `libc`. Le script `server.py` crée une table eBPF puis ouvre une raw socket et lui applique un filtre. Le filtre est compilé en eBPF qui diffère fondamentalement du BPF classique. La table eBPF créée permet au serveur de connaître un « état » lié à la socket.

Pour que le `read()` du serveur retourne, il faut que le paquet reçu soit accepté par le filtre. Il n'existe pas de désassembleur eBPF correct et la documentation du bytecode est particulièrement peu fournie. On peut cependant se baser sur ebpf-disasm, sachant qu'il faudra recalculer tous les sauts, appels de fonction et corriger les instructions de `ldxdw` (qui sont mal interprétées) pour obtenir une sortie exploitable.

Le filtre commence par une règle « classique », qui se traduirait en PCAP par : `ip proto udp and dst port 1337` :

```
000:    mov64 r6, r1
001:    mov64 r7, 0x0               ; ret = 0 (rejected)
002:    ldabsh 0xc
003:    jne r0, 0x800, exit         ; ipv4
004:    ldabsb 0x17
005:    jne r0, 0x11, exit          ; proto udp
006:    ldabsh 0x10
007:    mov64 r8, r0                ; r8 = len
008:    ldabsb 0xe
009:    mov64 r9, r0                ; r9 = version | len
00a:    ldabsh 0x24
00b:    jne r0, 0x539, exit         ; dst port 1337
00c:    lddw r1, 0xfffffff8
00e:    add64 r8, r1                ; len -= 8 (udp header)
00f:    lsh64 r9, 0x2
010:    and64 r9, 0x3c              ; r9 = offset
011:    sub64 r8, r9                ; len -= 20 (ip header)
012:    stxdw [r10+0xffe8], r9      ; offset -> sp-0x18
013:    add64 r9, 0x16              ; offset += 22 (&payload[0])
```

Une fois ces vérifications faites, le filtre fait une requête pour récupérer l'état de la variable `state` du serveur python. Si la clé n'existe pas ou si sa valeur est 0, alors le filtre vérifie que le paquet contient un LUM valide, sinon la vérification porte sur la clé permettant de valider l'épreuve :

```
016:    lddw r1, 0x4
018:    mov64 r2, r10
019:    add64 r2, 0xfffffff8        ; r2 = sp-8
01a:    call bpf_map_lookup_elem    ; fd=4, key=0
01b:    jne r0, 0x0, .L1            ; value != NULL
01c:    ja .L2
.L1:
01d:    ldxdw r1, [r0+0x0]
01e:    jne r1, 0x0, .L3            ; *value != 0 (state)
.L2:                                ; check_lum
```

3.1 Clé LUM

Compte tenu du code python de `server.py` on peut déduire que cette partie du filtre vérifie un LUM. Le code commence par comparer la taille du payload calculée précédemment :

```
022:    jne r8, 0x10, exit          ; len != 16 (payload)
```

Les deux premiers octets du payload sont vérifiés par rapport à "LU" :

```
026:    mov64 r8, r0                ; r8 = payload[0]
027:    ldxdw r9, [r10+0xffe8]
028:    add64 r9, 0x17              ; offset += 23 (&payload[1])
029:    ldindb r9, 0x0              ; r0 = payload[1]
02a:    and64 r8, 0xff
02b:    jne r8, 0x4c, exit          ; L...............
02c:    and64 r0, 0xff
02d:    jne r0, 0x55, exit          ; LU.............
```

Le payload est ensuite découpé en 1 mot de 16 bits et 3 mots de 32 bits aux offsets 0x18, 0x1a, 0x1e et 0x22. Les deux mots de 32 bits de poids forts sont d'abord combinés pour former un mot de 64 bits :

```
03b:    ldxdw r1, [r10+0xffe0]      ; offset2 = &payload[8]
03c:    ldindw r1, 0x0
03d:    mov64 r8, r0
03e:    add64 r9, 0x18              ; offset = &payload[2]
03f:    ldindh r9, 0x0
040:    lsh64 r8, 0x20
041:    ldxdw r1, [r10+0xffd8]      ; ((u32 *)payload)[3]
042:    or64 r8, r1
043:    lddw r1, 0x456443724d66417d ; LU......EdCrMfA}
045:    jne r8, r1, exit
046:    ldxdw r1, [r10+0xffd0]      ; ((u32 *)payload)[1]
047:    lsh64 r1, 0x20
048:    rsh64 r1, 0x20
049:    jne r1, 0x42765751, exit    ; LU..BvWQEdCrMfA}
04a:    and64 r0, 0xffff
04b:    jne r0, 0x4d7b, exit        ; LUM{BvWQEdCrMfA}
```

La table eBPF est ensuite mise à jour pour passer `state` à 1 :

```
050:    lddw r1, 0x4
052:    mov64 r2, r10
053:    add64 r2, 0xfffffff0
054:    mov64 r3, r10
055:    add64 r3, 0xfffffff8
056:    mov64 r4, 0x0
057:    call bpf_map_update_elem    ; fd=4, key=0, value=1, flags=0
```

Finalement, les mots de 32 bits aux offsets 0x1a et 0x1e sont logiquement inversés et insérés dans la table eBPF pour les clés 1 et 2 :

```
058:    ldxdw r1, [r10+0xffc8]      ; offset = &payload[8]
059:    ldindw r1, 0x0
05a:    xor64 r0, 0xffffffff
05b:    lsh64 r0, 0x20
05c:    rsh64 r0, 0x20
05d:    stxdw [r10+0xfff8], r0
05e:    stxdw [r10+0xfff0], r7
05f:    lddw r1, 0x4
061:    mov64 r2, r10
062:    add64 r2, 0xfffffff0
063:    mov64 r3, r10
064:    add64 r3, 0xfffffff8
065:    mov64 r4, 0x0
066:    call bpf_map_update_elem    ; fd=4, key=1, value=0xBD89A8AE, flags=0
```

```
067:    ldxdw r1, [r10+0xffe0]      ; offset = &payload[12]
068:    ldindw r1, 0x0
069:    xor64 r0, 0xffffffff
06a:    lsh64 r0, 0x20
06b:    rsh64 r0, 0x20
06c:    stxdw [r10+0xfff8], r0
06d:    mov64 r1, 0x2
pkt_accept:
06e:    stxdw [r10+0xfff0], r1
06f:    lddw r1, 0x4
071:    mov64 r2, r10
072:    add64 r2, 0xfffffff0
073:    mov64 r3, r10
074:    add64 r3, 0xfffffff8
075:    mov64 r4, 0x0
076:    call bpf_map_update_elem    ; fd=4, key=2, value=0xBA9BBC8D, flags=0
077:    lddw r7, 0xffffffff         ; ret = (u32)-1 (accepted)
079:    ja exit
.L3:                                ; check_key
```

On peut alors valider la clé LUM suivante : `LUM{BvWQEdCrMfA}`.

3.2 Clé de l'épreuve

Il faut maintenant trouver la clé permettant de valider l'épreuve. Pour cela il faut s'attaquer aux instructions à partir de 0x7a. Pour commencer, la longueur du payload est comparée à 37 et la variable `state` doit valoir 1 pour continuer la vérification :

```
07a:    mov64 r7, 0x0               ; ret = 0 (rejected)
07b:    lsh64 r8, 0x20
07c:    rsh64 r8, 0x20
07d:    jne r8, 0x25, exit          ; len != 37
07e:    jne r1, 0x1, exit           ; state != 1
```

En continuant à annoter l'assembleur, on peut voir que le payload attendu doit être de la forme : `KEY{.............................}`.

Les deux valeurs précédemment insérées dans la table eBPF sont alors récupérées et vont servir plus tard pour la vérification de la clé :

```
09b:    lddw r1, 0x4
09d:    mov64 r2, r10
09e:    add64 r2, 0xfffffff8
09f:    call bpf_map_lookup_elem    ; fd=4, key=1
0a0:    mov64 r8, 0x0
0a1:    mov64 r1, 0x0
0a2:    stxdw [r10+lum1], r1
0a3:    jeq r0, 0x0, .L5            ; value == NULL
0a4:    ldxdw r1, [r0, 0x0]
0a5:    stxdw [r10+lum1], r1
; ...
0a8:    mov64 r1, 0x2
0a9:    stxdw [r10+0xfff8], r1
0aa:    lddw r1, 0x4
0ac:    mov64 r2, r10
0ad:    add64 r2, 0xfffffff8
0ae:    call bpf_map_lookup_elem    ; fd=4, key=2
0af:    jeq r0, 0x0, .L6            ; value == NULL
0b0:    ldxdw r8, [r0+0x0]
.L6:    stxdw [r10+lum2], r8
```

La compilation du code eBPF semble avoir été faite avec l'option `-loop-unroll`. La suite du code vérifie que la clé fournie ne contient que des chiffres hexadécimaux puis découpe la clé en quatre mots de 32 bits qui sont stockés quartet par quartet, en conservant leur position dans le mot de 32 bits, dans le désordre sur la pile.

Pour s'en assurer, il est possible d'injecter une routine à la fin du filtre qui écrit la valeur des registres et le contenu de la pile dans la table eBPF, puis retourne une valeur différente de 0 de manière à pouvoir récupérer les valeurs depuis le serveur python. Avec cette technique il est même possible de debugger le filtre en mode pas à pas en décalant l'instruction de saut vers le code injecté de 8 octets à chaque « pas », puis en renvoyant un paquet UDP de la bonne longueur sur le port 1337 pour déclencher le filtre à nouveau.

Les deux premiers mots de la clé sont obtenus par un calcul simple utilisant des constantes formant "SSTIC CH" et "polymorf", ainsi que le champ longueur de l'en-tête UDP et deux mots de 32 bits du LUM logiquement inversés :

```
400:    ldxdw r1, [r10+lum1]            ; ~lum[1]
401:    xor64 r5, r1                    ; r5 = k[0] ^ ~lum[1]
; ...
409:    mov64 r9, r5
; ...
410:    mov64 r9, r5
; ...
412:    mov64 r9, r5
; ...
endparse:
41c:    ldabsh 0x26                     ; udp_len (37 + 8 = 45)
41d:    and64 r0, 0xff
41e:    mov64 r1, r0
41f:    mul64 r1, 0x706f6c79            ; "poly"
420:    xor64 r9, r1                    ; r9 ^= "poly" * udp_len
421:    lsh64 r9, 0x20
422:    rsh64 r9, 0x20
423:    jne r9, 0x53535449, clearstate  ; "SSTI" == k[0] ^ ~lum[1] ^ ("poly" * udp_len)
```

Et pour le deuxième mot de 32 bits :

```
433:    ldxdw r1, [r10+lum2]
434:    xor64 r2, r1                    ; r2 = k[1] ^ ~lum[2]
435:    mov64 r1, 0x6d6f7266            ; "morf"
436:    div64 r1, r0
437:    add64 r1, r2                    ; r1 = r2 + "morf" / udp_len
438:    lsh64 r1, 0x20
439:    rsh64 r1, 0x20
43a:    jne r1, 0x43204348, clearstate  ; "C CH" == k[1] ^ ~lum[2] + "morf" / udp_len
```

Les deux mots suivants sont issus d'un calcul plus complexe. On peut reconnaître un motif se répétant dont le calcul forme $x^2 \pmod n$, et $y \times x \pmod n$, ce qui correspondrait à l'algorithme de « square and multiply » utilisé pour le calcul d'exponentiation modulaire. Le code commence par la mise au carré de la valeur d'entrée $\pmod n$:

```
44d:    mov64 r1, r8            ; m[0] = p[0]
44e:    mul64 r1, r1            ; m[0] * m[0]
44f:    lddw r2, 0x8a46a52d     ; n = 2319885613
451:    mov64 r3, r1
452:    div64 r3, r2
453:    mul64 r3, r2
454:    sub64 r1, r3            ; m[1] = (m[0] * m[0]) % n
```

Avec les multiplications $\pmod n$ entremêlées :

```
45a:    mov64 r3, r8
45b:    div64 r3, r2
45c:    mul64 r3, r2
45d:    sub64 r8, r3            ; p[0] = (1 * m[0]) % n
; ...
477:    mul64 r1, r1            ; m[i-1] * m[i-1]
478:    mov64 r3, r1
479:    div64 r3, r2
47a:    mul64 r3, r2
47b:    sub64 r1, r3            ; m[i] = (m[i-1] * m[i-1]) % n
47c:    mul64 r1, r8            ; p[j-1]
47d:    mov64 r3, r1
47e:    div64 r3, r2
47f:    mul64 r3, r2
480:    sub64 r1, r3            ; p[j] = (p[j-1] * m[i]) % n
```

Après vérification pour les deux valeurs de n : 2319885613 et 3698100449, ce sont bien des nombres composites formés de deux nombres premiers chacun : $2319885613 = 43019 \times 53927$ et $3698100449 = 56921 \times 64969$. Le code implémente du RSA 32 bits dont la clé publique est 257. Le résultat pour chacun des mots de 32 bits forme "ALL " et "2017" en little-endian.
Le calcul des clés privés est trivial dès lors que p et q sont connus, il ne reste plus qu'à calculer la clé dans son intégralité.

Tout d'abord, les constantes utilisées :

```
unsigned char lum[] = "LUM{BvWQEdCrMfA}";
unsigned char sstic[] = "SSTIC CHALL 2017";
unsigned char poly[] = "polymorf";
unsigned short udplen = 0x2d;
unsigned long p1 = 43019, q1 = 53927;
unsigned long p2 = 56921, q2 = 64969;
unsigned int pub = 257;
```

Il faut ensuite convertir dans la bonne endianness, et faire l'inversion logique sur les parties de LUM utilisées :

```
unsigned int *p32lum, *p32sstic, *p32poly;

p32lum = (unsigned int *)lum;
p32sstic = (unsigned int *)sstic;
p32poly = (unsigned int *)poly;

p32lum[1] = __builtin_bswap32(~p32lum[1]);
p32lum[2] = __builtin_bswap32(~p32lum[2]);
p32sstic[0] = __builtin_bswap32(p32sstic[0]);
p32sstic[1] = __builtin_bswap32(p32sstic[1]);
p32poly[0] = __builtin_bswap32(p32poly[0]);
p32poly[1] = __builtin_bswap32(p32poly[1]);
```

Le calcul de la clé est alors direct :

```
unsigned int key[4];

key[0] = p32lum[1] ^ p32sstic[0] ^ (p32poly[0] * udplen);
key[1] = p32lum[2] ^ (p32sstic[1] - (p32poly[1] / udplen));
key[2] = p32lum[1] ^ modexp(p32sstic[2], modinv(pub, (p1 - 1) * (q1 - 1)), p1 * q1);
key[3] = p32lum[2] ^ modexp(p32sstic[3], modinv(pub, (p2 - 1) * (q2 - 1)), p2 * q2);

printf("Key is: %08x%08x%08x%08x\n", key[0], key[1], key[2], key[3]);
```

On obtient finalement la clé pour valider l'épreuve :

```
$ ./ebpf_key
Key is: 2d4ceda2fa2a0e08fc360b55291de7c9
```

3.3 Résolution rapide

La principale difficulté de cette épreuve provient du manque d'outils pour analyser et debugger le bytecode eBPF, couplé à du code relativement long et obfusqué.
Il est cependant possible de résoudre l'épreuve en limitant fortement l'analyse statique du code dès lors qu'on s'aperçoit de la possibilité de remonter la valeur des registres via une table eBPF.
La partie analyse statique se limite alors à l'identification des quatres blocs de code vérifiant la clé. Ils sont identifiables par les instructions de saut qui mènent vers la sortie du filtre en 0x4d4 alors que la valeur de retour est mise à 0 (registre `r7`).
Les deux derniers mots de 32 bits peuvent s'obtenir par `bruteforce` en convertissant le code assembleur vers du C avec une ligne de `sed`, ce qui prend moins d'une minute sur une machine « récente ».

4 Riscy Zones

L'épreuve contient quatre fichiers :
— TA.elf.signed
— password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted
— secret.lzma.encrypted
— trustzone_decrypt

Deux fichiers chiffrés sont présents, dont un pour lequel la clé est dans le nom. Le fichier `TA.elf.signed` est un ELF signé, compilé pour RISC-V. L'exécutable principal, `trustzone_decrypt`, est compilé pour tourner dans le Linux de la VM OR1k.
Pour se donner une idée du comportement de l'exécutable, la première chose à faire est de tracer son exécution avec `strace` en le faisant déchiffrer le fichier dont on connaît la clé :

```
/challenges/riscy_zones $ strace ./trustzone_decrypt 00112233445566778899AABBCCDDEEFF
    password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted /tmp/output
execve("./trustzone_decrypt", ["./trustzone_decrypt", "00112233445566778899AABBCCDDEEFF", "
    password_is_00112233445566778899"..., "/tmp/output"], [/* 12 vars */]) = 0
set_tid_address(0x300c30ec)             = 103
openat(AT_FDCWD, "./TA.elf.signed", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=8036, ...}) = 0
mmap2(NULL, 8036, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x300c6000
ioctl(1, TCGETS, {B38400 opost isig icanon echo ...}) = 0
writev(1, [{iov_base="[\33[34;1mi\33[0m] load TA.elf.signe"..., iov_len=46}, {iov_base="\n",
    iov_len=1}], 2[i] load TA.elf.signed in TrustedOS
) = 47
openat(AT_FDCWD, "/dev/sstic", O_RDONLY|O_LARGEFILE) = 4
ioctl(4, _IOC(_IOC_READ|_IOC_WRITE, 0x53, 0, 0x14), 0x7fd71c4c) = 0
writev(1, [{iov_base="[\33[32;1m+\33[0m] OS return code = "..., iov_len=71}, {iov_base="\n",
    iov_len=1}], 2[+] OS return code = 0x00000000, TA return code = 0x00000000
) = 72
close(4)                                = 0
```

On peut voir que l'exécutable charge l'application signée `TA.elf.signed` en mémoire et le passe au « Trusted Execution Environment » à travers un `ioctl()` sur `/dev/sstic`.
Le « Trusted Execution Environment » correspond à l'émulateur de la console du bas dans l'interface du jeu. Si l'on s'intéresse à l'outil permettant de valider les clés des épreuves : `tee_client.py`, on peut s'apercevoir que cet outil aussi fait appel à `/dev/sstic` avec la même valeur pour la requête `ioctl`. L'argument de la requête est une structure de 20 octets (size = 0x14) qui sert d'entrée et de sortie (`_IOC_READ|_IOC_WRITE`). Le type de requête correspondrait à un CDROM (type = 0x53). Le pilote associé génère alors une instruction `l.sys` qui est interprété par le JavaScript dans `jor1k-worker-min.js` :

```
case 8:
    553648128 == (d & 4294901760) ? v(3584, w[2060] | 0) : d & 1 ? v(3072, w[2060] | 0) : ma ? (
    d = t.handleSyscall(u[11], u[3], u[4], u[5], u[6]), u[11] = d[0], u[6] = d[1]) : v(1792, F);
    continue;
```

La structure est divisée en 5 champs, dont le premier correspond à un identifiant de commande :

```
d.prototype.handleSyscall = function(a, b, d, e, f) {
    this.PrintDebug("new command from normal world");
    this.PrintDebug("CMD                    " + c.ToHex(a));
    this.PrintDebug("INPUT BUFFER           " + c.ToHex(b));
    this.PrintDebug("INPUT LEN              " + c.ToHex(d));
    this.PrintDebug("OUTPUT BUFFER          " + c.ToHex(e));
    this.PrintDebug("OUTPUT LEN             " + c.ToHex(f));
```

On peut en déduire la structure C correspondante, qui fait bien 20 octets :

```
struct ta_io {
    int  command;
    int  in_len;
    char *in_buf;
    int  out_len;
    char *out_buf;
};
```

Les noms des différentes commandes disponibles sont gardés en clair dans le JavaScript, ce qui donne :
— 1 : Get Version
— 2 : Load Trusted Application
— 3 : Trusted Application Command
— 4 : Unload Trusted Application
— 5 : Check LUM
— 6 : Check Key

4.1 Commandes de la Trusted Application

La commande 3 permet de piloter l'application RISC-V en charge du déchiffrement. Pour la suite de la résolution de l'épreuve il faut compiler `binutils` pour OR1k et RISC-V afin de pouvoir désassembler les binaires avec `objdump`. La commande `strings` sur le binaire RISC-V permet de déduire les différentes commandes supportées :

```
$ strings TA.elf.signed | grep CMD
[DEBUG] CMD CMD_TA_INIT
[DEBUG] CMD CMD_GET_TA_VERSION
[DEBUG] CMD CMD_GET_TA_LUM
[DEBUG] CMD CMD_CHECK_PASSWORD
[DEBUG] CMD CMD_DECRYPT_BLOCK
[DEBUG] Unknown CMD
```

L'application étant compilée avec des informations de debug, cela devrait simplifier la tâche. Certains symboles sont présents, et correspondent à des `syscall` interprétés par l'émulateur RISC-V JavaScript :

```
$ readelf -s TA.elf.signed | grep GLOBAL
     4: 000113f8    12 FUNC    GLOBAL DEFAULT    2 TEE_HMAC
     5: 000113e0    12 FUNC    GLOBAL DEFAULT    2 TEE_writekey
     6: 000113c8    12 FUNC    GLOBAL DEFAULT    2 TEE_write
     7: 000113d4    12 FUNC    GLOBAL DEFAULT    2 TEE_readkey
     8: 000113ec    12 FUNC    GLOBAL DEFAULT    2 TEE_AES_decrypt
```

Le point d'entrée de l'application est en 0x11590. On remarque l'utilisation d'une table pour l'implémentation d'un `switch` pour des valeurs comprises entre 0 et 4 :

```
   115e8:   00279793            slli    a5,a5,0x2
   115ec:   05470713            addi    a4,a4,84    ; 10054 <jmptable>
   115f0:   00e787b3            add     a5,a5,a4
   115f4:   0007a783            lw      a5,0(a5)
   115f8:   00078067            jr      a5          ; switch(cmd)
```

On peut obtenir les adresses des différentes commandes avec `hexdump` :

```
$ hexdump -s $((0x54)) -n $((5*4)) -e '"" 1/4 "%x " "\n"' TA.elf.signed
11b98
11b00
11a08
11644
1192c
```

La fonction `TEE_write` va permettre d'identifier rapidement le rôle de ces fonctions. Elle n'est appelée qu'à un seul endroit de l'application, dans la fonction située à l'adresse 0x1155c :

```
write_console:
   1155c:   00054783            lbu     a5,0(a0)
   11560:   00050593            mv      a1,a0
   11564:   02078263            beqz    a5,11588 <write_console+0x28>
   11568:   00050793            mv      a5,a0
   1156c:   00000613            li      a2,0
   11570:   00178793            addi    a5,a5,1
   11574:   0007c703            lbu     a4,0(a5)
   11578:   00160613            addi    a2,a2,1
   1157c:   fe071ae3            bnez    a4,11570 <write_console+0x14>
   11580:   00100513            li      a0,1
   11584:   e45ff06f            j       113c8 <TEE_write>
   11588:   00000613            li      a2,0
   1158c:   ff5ff06f            j       11580 <write_console+0x24>
```

Chacune des cinq fonctions obtenues par la jmptable commence par un appel à `write_console`. On peut s'aider des chaînes de caractères de debug pour en identifier l'usage.
Cas 0 :

```
cmd_ta_init:
   11b98:   00010537            lui     a0,0x10
   11b9c:   09050513            addi    a0,a0,144 ; 10090 "[DEBUG] CMD CMD_TA_INIT\n"
   11ba0:   9bdff0ef            jal     ra,1155c <write_console>
```

Cas 1 :

```
cmd_get_ta_version:
   11b00:   00010537            lui     a0,0x10
   11b04:   0fc50513            addi    a0,a0,252 ; 100fc "[DEBUG] CMD CMD_GET_TA_VERSION\n"
   11b08:   a55ff0ef            jal     ra,1155c <write_console>
```

Cas 2 :

```
cmd_check_password:
   11a08:   00010537            lui     a0,0x10
   11a0c:   14050513            addi    a0,a0,320 ; 10140 "[DEBUG] CMD CMD_CHECK_PASSWORD\n"
   11a10:   b4dff0ef            jal     ra,1155c <write_console>
```

Cas 3 :

```
cmd_decrypt_block:
   11644:   00010537            lui     a0,0x10
   11648:   39050513            addi    a0,a0,912 ; 10390 "[DEBUG] CMD CMD_DECRYPT_BLOCK\n"
   1164c:   f11ff0ef            jal     ra,1155c <write_console>
```

Cas 4 :

```
cmd_get_ta_lum:
   1192c:   00010537            lui     a0,0x10
   11930:   12050513            addi    a0,a0,288 ; 10120 "[DEBUG] CMD CMD_GET_TA_LUM\n"
   11934:   c29ff0ef            jal     ra,1155c <write_console>
```

4.2 Première clé LUM

Une fonction qui retourne une clé LUM a été identifiée. L'assembleur est assez simple à suivre, le code de la commande (4) est utilisé pour le résultat qui est un tableau de 16 caractères, retranché de la position courante dans le tableau. Une opération de ou-exclusif est alors appliquée sur chacun des caractères du

résultat. Après extraction de la clé `XOR`, on peut recalculer le résultat :

```
unsigned char xr[] = { 72,86,79,122,103,-101,-80,-59,-46,-65,-48,-71,-45,-94,-96,-120 };
int main()
{
    unsigned char c = 4; int i;
    for (i=0;i<16;i++) printf("%c", (c - i) ^ xr[i]);
    printf("\n");
}
```

On peut alors valider le LUM suivant : `LUM{gdN8.D*@+UV}`.

```
$ ./lum1
LUM{gdN8.D*@+UV}
```

4.3 Format du séquestre de clé

Pour l'étape suivante, on va s'intéresser à la fonction de vérification du mot de passe donné pour le déchiffrement. La sortie de la commande `strace` va aider pour l'analyse :

```
openat(AT_FDCWD, "password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted", O_RDONLY|
    O_LARGEFILE) = 4
read(4, "E!O\317q/\342=O\22\30gB\370\31}\t\0\312j\216\17\367Yg\315|4%\337^\250"..., 112) = 112
writev(1, [{iov_base="[\33[34;1mi\33[0m] check password in"..., iov_len=36}, {iov_base="\n",
    iov_len=1}], 2[i] check password in TEE
) = 37
openat(AT_FDCWD, "/dev/sstic", O_RDONLY|O_LARGEFILE) = 5
ioctl(5, _IOC(_IOC_READ|_IOC_WRITE, 0x53, 0, 0x14), 0x7f9d3c94) = 0
```

Le mot de passe ainsi que les 112 premiers octets du fichier à déchiffrer sont utilisés dans la vérification de la clé. On peut retrouver cette lecture de 112 octets dans le binaire OR1k, ce qui nous permet d'identifier la fonction responsable de valider le mot de passe, ainsi que ses arguments.
L'OpenRISC implémente un delay slot, qu'il faut prendre en compte pour les instructions de saut et d'appel de fonction. La valeur de retour des fonctions (ainsi que les numéros d'appel système) se trouve dans `r11` et les arguments sont placés dans les registres à partir de `r3` :

```
    2580:   07 ff ff e8     l.jal strlen <main-0x3c>    ; strlen(argv[1]);
    2584:   a8 6e 00 00     l.ori r3,r14,0x0
    2588:   bc 0b 00 20     l.sfeqi r11,32              ; strlen(argv[1]) == 32
    258c:   10 00 00 0b     l.bf 25b8 <lenpassok>
; ...
lenpassok:
    25b8:   a8 6e 00 00     l.ori r3,r14,0x0
    25bc:   9c 81 00 04     l.addi r4,r1,4              ; hex_pass = r1 + 4;
    25c0:   04 00 00 f5     l.jal 2994 <decode_hex>     ; decode_hex(argv[1], hex_pass, 16);
    25c4:   9c a0 00 10     l.addi r5,r0,16
; ...
    260c:   9c 81 00 14     l.addi r4,r1,20             ; read_buf = r1 + 20;
    2610:   07 ff ff b0     l.jal read <main-0x8c>      ; read(encrypted_fd, read_buf, 112);
    2614:   9c a0 00 70     l.addi r5,r0,112
    2618:   bc 0b 00 70     l.sfeqi r11,112
    261c:   10 00 00 18     l.bf 267c <readok>
; ...
readok:
    267c:   9c 61 00 04     l.addi r3,r1,4              ; hex_pass = r1 + 4;
    2680:   04 00 01 ed     l.jal 2e34 <checkpassword>  ; checkpassword(hex_pass, read_buf);
    2684:   9c 81 00 14     l.addi r4,r1,20             ; read_buf = r1 + 20;
```

La fonction `checkpassword` permet d'avoir le format des données d'entrée pour le binaire RISC-V :

```
    2e5c:   9c 60 01 04     l.addi r3,r0,260
    2e60:   07 ff fd 88     l.jal malloc <main-0xdc>        ; out_buf = malloc(260);
    2e64:   ab 04 00 00     l.ori r24,r4,0x0
    2e68:   9c 60 01 18     l.addi r3,r0,280
    2e6c:   a9 cb 00 00     l.ori r14,r11,0x0
    2e70:   07 ff fd 84     l.jal malloc <main-0xdc>        ; in_buf = malloc(280);
    2e74:   aa 83 00 00     l.ori r20,r3,0x0
    2e78:   9c 80 00 00     l.addi r4,r0,0
    2e7c:   a8 6b 00 00     l.ori r3,r11,0x0
    2e80:   a8 b4 00 00     l.ori r5,r20,0x0
    2e84:   9e 40 01 04     l.addi r18,r0,260
    2e88:   07 ff fd 97     l.jal memset <main-0x78>        ; memset(in_buf, 0, 280);
    2e8c:   a8 4b 00 00     l.ori r2,r11,0x0
    2e90:   9c 80 00 00     l.addi r4,r0,0
    2e94:   a8 b2 00 00     l.ori r5,r18,0x0
    2e98:   07 ff fd 93     l.jal memset <main-0x78>        ; memset(out_buf, 0, 260);
    2e9c:   a8 6e 00 00     l.ori r3,r14,0x0
    2ea0:   18 60 00 00     l.movhi r3,0x0
    2ea4:   07 ff fd 72     l.jal puts <main-0xf0>
    2ea8:   a8 63 35 5c     l.ori r3,r3,0x355c              ; check password in TEE
    2eac:   18 c0 02 00     l.movhi r6,0x200
    2eb0:   9c 62 00 18     l.addi r3,r2,24
    2eb4:   d4 02 30 00     l.sw 0(r2),r6                   ; ((int *)in_buf)[0] = 0x02;
    2eb8:   a8 98 00 00     l.ori r4,r24,0x0
    2ebc:   07 ff fd 67     l.jal memcpy <main-0x104>       ; memcpy(in_buf+24, IV, 112);
    2ec0:   9c a0 00 70     l.addi r5,r0,112
    2ec4:   9c 62 00 06     l.addi r3,r2,6
    2ec8:   a8 96 00 00     l.ori r4,r22,0x0
    2ecc:   07 ff fd 63     l.jal memcpy <main-0x104>       ; memcpy(in_buf+6, password, 16);
    2ed0:   9c a0 00 10     l.addi r5,r0,16
    2ed4:   9c 80 10 00     l.addi r4,r0,4096
    2ed8:   a8 61 00 00     l.ori r3,r1,0x0
    2edc:   dc 02 20 04     l.sh 4(r2),r4                   ; ((short *)in_buf)[2] = 16;
    2ee0:   9c 80 70 00     l.addi r4,r0,28672
    2ee4:   d4 01 a0 04     l.sw 4(r1),r20
    2ee8:   dc 02 20 16     l.sh 22(r2),r4                  ; ((short *)in_buf)[11] = 112;
    2eec:   9c 80 00 03     l.addi r4,r0,3
    2ef0:   d4 01 10 08     l.sw 8(r1),r2
    2ef4:   d4 01 20 00     l.sw 0(r1),r4
    2ef8:   d4 01 90 0c     l.sw 12(r1),r18
    2efc:   07 ff fe f4     l.jal 2acc <ioctl_sstic>        ; ioctl_sstic((struct io *)r1);
```

Problèmes d'endianness mis à part, on obtient le format suivant pour le buffer d'entrée :

Command [4B]	Size0 [2B]	Data0 [Size0 B]	...	SizeN [2B]	DataN [SizeN B]

Il faut repasser sur le binaire RISC-V pour la suite de l'analyse. La fonction `cmd_check_password` commence par vérifier la taille des données, puis fait un appel système pour déchiffrer le séquestre avec la clé enregistrée sous "SSTIC_AES_KEY" :

```
   11a1c:   lhu     s1,22(s0)    ; s1 = size1;
   11a20:   ble     s2,a5,11a2c <cmd_check_password+0x24>   ; size0 <= 16
; ...
   11a2c:   li      a5,256
   11a30:   ble     s1,a5,11a3c <cmd_check_password+0x34>   ; size1 <= 256
; ...
   11a3c:   lui     a0,0x10
   11a40:   addi    a2,s0,24     ; a2 = &inbuf[24];
   11a44:   addi    a1,sp,64     ; a1 = outbuf;
   11a48:   mv      a3,s1        ; a3 = size1;
   11a4c:   addi    a0,a0,192    ; 100c0 "SSTIC_AES_KEY"
   11a50:   jal     ra,113ec <TEE_AES_decrypt>
```

Des clés sont enregistrées par l'application pendant l'exécution de la fonction d'initialisation `cmd_ta_init`, les clés sont stockées dans la section `.rodata` de l'application :

```
   11bac:   addi    a1,a1,172   ; 100ac "___SSTIC_2017___"
   11bb0:   li      a3,0        ; xorkey = 0;
   11bb4:   li      a2,16
   11bb8:   addi    a0,a0,192   ; 100c0 "SSTIC_AES_KEY"
   11bbc:   jal     ra,113e0 <TEE_writekey>
   11bc0:   lui     a1,0x10
   11bc4:   lui     a0,0x10
   11bc8:   li      a3,1        ; xorkey = 1;
   11bcc:   li      a2,16
   11bd0:   addi    a1,a1,208   ; 100d0 "\x7f\xdc\xb9\x86\x05\x87\x67\xec\x47\xf4\x17\xef\xbe\
    x85\xa1\x0c"
   11bd4:   addi    a0,a0,228   ; 100e4 "SSTIC_PASSWORD_HMAC_KEY"
   11bd8:   jal     ra,113e0 <TEE_writekey>
```

Le paramètre stocké dans le registre d'argument `a3` est un drapeau précisant si une opération de ou-exclusif doit être appliqué à la clé. On peut le retrouver dans l'appel système 3 de l'émulateur RISC-V en JavaScript, dans la variable `l` :

```
case 3:
    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;
```

La fonction de déchiffrement AES utilise le mode ECB, comme on peut le voir dans l'implémentation de l'appel système 4 :

```
case 4:
    d = h(a, d, 256);
    d = this.getKey(d);
    c = e(a, g, l);
    g = (new x.ModeOfOperation.ecb(d)).decrypt(c);
    for (c = 0; c < g.length && !(c > l); c++) k = g[c], a.Write8(f + c, k);
    break;
```

On peut maintenant déchiffrer le séquestre avec `OpenSSL` :

```
$ dd if=password_is_00112233445566778899AABBCCDDEEFF.txt.encrypted bs=112 count=1 2>/dev/null |
    openssl enc -d -aes-128-ecb -K $(echo -n ___SSTIC_2017___ | xxd -p) -nopad
==BEGIN PASSWORD HMAC==
4bcea2ecf77bda18804b395a5b0a81c9c745b5add567073c3afb4ec5e7a478fa
==END PASSWORD HMAC==
```

Après avoir vérifié les chaînes d'encapsulation du HMAC, la fonction calcule le HMAC du mot de passe fourni en entrée :

```
   11aa8:   00090693            mv      a3,s2       ; a3 = pass_len;
   11aac:   00040613            mv      a2,s0       ; a2 = &password;
   11ab0:   00010593            mv      a1,sp       ; outbuf = &stack;
   11ab4:   0e450513            addi    a0,a0,228   ; 100e4 "SSTIC_PASSWORD_HMAC_KEY"
   11ab8:   941ff0ef            jal     ra,113f8 <TEE_HMAC>
```

Le HMAC se base sur SHA256 pour la fonction de hachage :

```
case 5:
    d = h(a, d, 256);
    d = this.getKey(d);
    l = e(a, g, l);
    c = new q("SHA-256", "ARRAYBUFFER");
    c.setHMACKey(String.fromCharCode.apply(null, d), "TEXT");
    c.update(l);
    l = c.getHMAC("BYTES");
    for (c = 0; c < l.length; c++) k = l.charCodeAt(c), a.Write8(f + c, k);
    break;
```

On peut recalculer la clé du HMAC grâce aux informations précédemment obtenues : la clé brute est `"\x7f\xdc\xb9\x86\x05\x87\x67\xec\x47\xf4\x17\xef\xbe\x85\xa1\x0c"`, et la clé `XOR` utilisée est `[51, 137, 244, 253, 53, 246, 17, 181, 15, 206, 70, 166, 129, 206, 151, 113]`.

4.4 Seconde clé LUM

Le résultat du ou-exclusif peut se calculer trivialement :

```
unsigned char cle[] = "\x7f\xdc\xb9\x86\x05\x87\x67\xec\x47\xf4\x17\xef\xbe\x85\xa1\x0c";
unsigned char xor[] = { 51, 137, 244, 253, 53, 246, 17, 181, 15, 206, 70, 166, 129, 206, 151,
    113 };
int main()
{
    int i;
    for (i=0;i<16;i++) printf("%c", cle[i] ^ xor[i]);
    printf("\n");
}
```

On peut alors valider le LUM suivant : `LUM{0qvYH:QI?K6}`.

```
$ ./lum2
LUM{0qvYH:QI?K6}
```

Finalement, on peut aussi vérifier le HMAC lui-même :

```
$ echo -n 00112233445566778899AABBCCDDEEFF | xxd -r -p | openssl dgst -sha256 -hmac "LUM{0qvYH:
    QI?K6}"
(stdin)= 4bcea2ecf77bda18804b395a5b0a81c9c745b5add567073c3afb4ec5e7a478fa
```

4.5 Signature de la Trusted Application

L'application RISC-V est « signée » avec un HMAC. La clé de ce HMAC est différente de celle utilisée pour le séquestre. On la retrouve dans la fonction d'initialisation de l'émulateur RISC-V en JavaScript :

```
this.loadKey("TA_HMAC_KEY", a("I'm an approuved TA builder !"));
```

On peut donc recalculer la « signature » de l'application, et donc la modifier si nécessaire :

```
$ tail -c 64 ../2/challenges/riscy_zones/TA.elf.signed ; echo
885fb119b7c185b196e53045cd12f640ce84a6fbbeeb7f1a3c432f0a800273d2
$ dd if=../2/challenges/riscy_zones/TA.elf.signed bs=$((8036-64)) count=1 2>/dev/null | openssl
    dgst -sha256 -hmac "I'm an approuved TA builder !"
(stdin)= 885fb119b7c185b196e53045cd12f640ce84a6fbbeeb7f1a3c432f0a800273d2
```

4.6 Fonction de déchiffrement

Pour valider l'épreuve, il faudra pour la suite trouver un moyen de recalculer la clé à partir du document chiffré `secret.lzma.encrypted` et de son HMAC-SHA256 :

```
$ dd if=secret.lzma.encrypted bs=112 count=1 2>/dev/null | openssl enc -d -aes-128-ecb -K $(echo
     -n ___SSTIC_2017___ | xxd -p) -nopad
==BEGIN PASSWORD HMAC==
504861089d10a8d5900cdf3bad962004282e73c20d2d5ab95b67ae6b3356748b
==END PASSWORD HMAC==
```

Le mot de passe fourni est stocké dans le TEE lorsqu'il est considéré comme valide. Ensuite, pour chaque commande de déchiffrement issue depuis le binaire OR1k, la clé est récupérée par l'application RISC-V. Le numéro de bloc est fourni avec les données à déchiffrer et entre dans le calcul de dérivation de la clé. La clé fournie est découpée en 4 mots de 32 bits sur lesquels est appliqué un algorithme de diffusion qui dépend du numéro de bloc à déchiffrer :

```
11660:    lw      s1,4(s0)     ; round number
11664:    jal     ra,113d4 <TEE_readkey>
11668:    lw      a3,68(sp)    ; k[1]
1166c:    lw      t5,76(sp)    ; k[3]
11670:    lui     a5,0xadaab
11674:    lui     t0,0x52555
11678:    addi    s2,a5,-1622 ; adaaa9aa
1167c:    addi    t0,t0,1621  ; 52555655
11680:    addi    a4,s1,23     ; a4 = round + 23;
11684:    and     a2,t5,s2     ; a2 = k[3] & 0xadaaa9aa;
11688:    and     a5,a3,t0     ; a5 = k[1] & 0x52555655;
1168c:    andi    a4,a4,31     ; a4 = (round + 23) % 32;
11690:    or      a5,a5,a2     ; a5 = (k[1] & 0x52555655) | (k[3] & 0xadaaa9aa)
11694:    neg     a2,a4        ; a2 = ~((round + 23) % 32)
11698:    srl     a4,a5,a4
1169c:    sll     a5,a5,a2
116a0:    or      a4,a4,a5     ; a4 = ROR32((k[1] & 0x52555655) | (k[3] & 0xadaaa9aa), (round
 + 23) % 32)
```

Les 3 autres mots de 32 bits de la clé diffusée se calculent selon le même principe, ce qui permet d'obtenir :

```
unsigned int mask1 = 0x52555655;
unsigned int mask2 = 0xadaaa9aa;

dk[0] = ROR(key[1] & mask1 | key[3] & mask2, 23 + round, 32);
dk[1] = ROR(key[0] & mask1 | key[2] & mask2, 19 + round, 32);
dk[2] = ROR(key[1] & mask2 | key[3] & mask1, 17 + round, 32);
dk[3] = ROR(key[0] & mask2 | key[2] & mask1, 13 + round, 32);
```

Pour la suite de l'analyse du code, on peut noter la dernière position d'écriture des registres utilisés pour le stockage du résultat final, l'ordre des registres étant `t1`, `a2`, `a1`, etc. Assez peu d'opérations sont effectuées à chaque fois, pour les deux premiers registres on a :

```
116a4:   01875313          srli    t1,a4,0x18  ; t1 = dk[0] >> 24
116a8:   04130313          addi    t1,t1,65    ; t1 += 65; res[15] = t1 ^ bloc[0];
116ac:   01075793          srli    a5,a4,0x10  ; a5 = dk[0] >> 16
116b0:   0ff37613          andi    a2,t1,255   ; a2 = t1 & 0xFF;
116b4:   0ff7f793          andi    a5,a5,255   ; a5 &= 0xff
116b8:   00f60633          add     a2,a2,a5    ; a2 += a5
116bc:   04012e83          lw      t4,64(sp)
116c0:   04812383          lw      t2,72(sp)
116c4:   04860613          addi    a2,a2,72    ; a2 += 72, res[14] = a2 ^ bloc[1];
```

La suite du code suit le même schéma :

$$c_n = \begin{cases} 0 & n < 0 \\ 67 + n * 7 + c_{n-1} + dk_n \pmod{256} & 0 \leq n < 16 \end{cases}$$

Pour finir le déchiffrement, il faut de nouveau passer sur le binaire OR1k. La fonction en charge du déchiffrement commence par lire le premier bloc de données, puis un second bloc qui sera donné à l'ap-

plication RISC-V :

```
    30a4:    l.addi r4,r1,4
    30a8:    l.ori r3,r26,0x0
    30ac:    l.jal read <main-0x8c>  ; r11 = read(encrypted_fd, sp+4, 16);
    30b0:    l.addi r5,r0,16
    30b4:    l.sfeqi r11,16          ; r11 != 16
    30b8:    l.bnf 32f0 <badiv>
    30bc:    l.addi r18,r0,0
    30c0:    l.j 32c4 <looptail>
    30c4:    l.movhi r30,0x300
decryptloop:
; ...
looptail:
    32c4:     l.ori r3,r26,0x0
    32c8:     l.addi r4,r1,20
    32cc:     l.jal read <main-0x8c>  ; r11 = read(encrypted_fd, sp+20, 16);
    32d0:     l.addi r5,r0,16
    32d4:     l.sfeqi r11,0           ; r11 == 0
    32d8:     l.bnf 30c8 <decryptloop>
```

Une opération de ou-exclusif est appliquée entre les deux blocs avant la fin de la boucle, puis le bloc courant est copié à l'emplacement du vecteur d'initialisation. C'est le mode CBC standard à la différence que le ou-exclusif est appliqué avec le bloc précédent lu à l'envers :

```
    3190:   8f 22 00 08     l.lbz r25,8(r2)     ; decrypted[0]
; ...
    31b0:   8d a1 00 13     l.lbz r13,19(r1)    ; lastbloc[15]
; ...
    31d8:   e1 b9 68 05     l.xor r13,r25,r13
; ...
    31fc:   d8 01 68 24     l.sb 36(r1),r13     ; output[0] = decrypted[0] ^ lastbloc[15]
; ...
    3234:   8d c2 00 17     l.lbz r14,23(r2)    ; decrypted[15]
; ...
    3254:   8c 41 00 04     l.lbz r2,4(r1)      ; lastbloc[0]
; ...
    3270:   e0 4e 10 05     l.xor r2,r14,r2
; ...
    3280:   d8 01 10 33     l.sb 51(r1),r2      ; output[15] = decrypted[15] ^ lastbloc[0]
; ...
    32a0:   84 41 00 14     l.lwz r2,20(r1)     ; r2 = ((u32 *)curbloc)[0];
    32a4:   9e 52 00 01     l.addi r18,r18,1    ; round++;
    32a8:   d4 01 10 04     l.sw 4(r1),r2       ; ((u32 *)lastbloc)[0] = r2;
    32ac:   84 41 00 18     l.lwz r2,24(r1)     ; r2 = ((u32 *)curbloc)[1];
    32b0:   d4 01 10 08     l.sw 8(r1),r2       ; ((u32 *)lastbloc)[1] = r2;
    32b4:   84 41 00 1c     l.lwz r2,28(r1)     ; r2 = ((u32 *)curbloc)[2];
    32b8:   d4 01 10 0c     l.sw 12(r1),r2      ; ((u32 *)lastbloc)[2] = r2;
    32bc:   84 41 00 20     l.lwz r2,32(r1)     ; r2 = ((u32 *)curbloc)[3];
    32c0:   d4 01 10 10     l.sw 16(r1),r2      ; ((u32 *)lastbloc)[3] = r2;
```

Tous les éléments sont maintenant réunis pour écrire notre propre déchiffreur si besoin. Cependant, ce qui va plus nous intéresser est de pouvoir calculer la clé à partir d'un bloc déchiffré choisi.

4.7 Inversion du keystream

L'algorithme de génération du keystream s'inverse bien, on obtient : $dk_n = c_n - (67 + n * 7 + c_{n-1})$; pour retomber sur la clé, il faut ensuite faire des rotations vers la gauche, puis masquer les 2 mots obtenus et finalement fusionner le resultat.
En prenant un bloc déchiffré connu, `expected`, la clé se retrouve par le code suivant :

```
#define ROR(x, n, b) (((x) >> ((n)%(b))) | ((x) << (b - ((n)%(b)))))
#define ROL(x, n, b) (((x) << ((n)%(b))) | ((x) >> (b - ((n)%(b)))))

unsigned int mask1 = 0x52555655;
unsigned int mask2 = 0xadaaa9aa;
unsigned char addkey = 65, *ptr;
unsigned int k[4], nk[4];
int i, round = 0;

memset(k, 0, 4 * sizeof(unsigned int));
addkey += 16 * 7;
for (i = 15; i >= 0; i--) {
    int d = i % 4;
    int e = i / 4;
    addkey -= 7;
    if (i) {
        k[e] |= ((expected[15 - i] - addkey - expected[16 - i]) & 0xff) << (24 - d * 8);
    } else {
        k[e] |= ((expected[15 - i] - addkey) & 0xff) << (24 - d * 8);
    }
}
nk[0] = ROL(k[1], 19 + round, 32) & mask1 | ROL(k[3], 13 + round, 32) & mask2;
nk[1] = ROL(k[0], 23 + round, 32) & mask1 | ROL(k[2], 17 + round, 32) & mask2;
nk[2] = ROL(k[1], 19 + round, 32) & mask2 | ROL(k[3], 13 + round, 32) & mask1;
nk[3] = ROL(k[0], 23 + round, 32) & mask2 | ROL(k[2], 17 + round, 32) & mask1;
```

Il faut maintenant trouver un bloc dont on connaît le clair. Une possibilité pourrait être d'utiliser le padding en espérant qu'il soit égal à 16, cela faciliterait grandement la tâche. Malheureusement ce n'est pas le cas, le padding est en réalité assez faible (5), cette piste ne fera que faire perdre du temps. Une autre piste est de se baser sur l'indice donné par le fichier chiffré, il est au format LZMA.

4.8 Clé de l'épreuve

L'en-tête LZMA fait 13 octets, suivis d'un octet nul de séparation avec le flux LZMA. Par défaut, les options sont : $lc = 3$, $lp = 0$ et $pb = 2$, avec une taille de dictionnaire de 8MB et la taille décompressée à -1. Si on part du principe que le fichier chiffré utilise les options par défaut, on obtient alors 14 octets de clair sur les 16 nécessaires à un chiffrement par bloc de 128 bits.
Comme le HMAC de la clé est connu, on peut alors lancer un bruteforce sur 16 bits et comparer le résultat du HMAC avec celui attendu.

Les données connues sont les suivantes :

- IV : `39962d55d138efefc6e4a14d5cd7352e`
- Bloc0 : `88db9a9733655cbf063b590244f075db`
- LZMA : `5d00008000ffffffffffffffff00XXXX`

On obtient : `fbee4d4b7e3b4786162b9e2ceeddXXXX`$= rev(IV) \oplus Bloc_0 \oplus LZMAhead$.
Le bruteforce est assez rapide et donne l'entrée `fbee4d4b7e3b4786162b9e2ceedd9c54`.
Ce qui correspond à la clé `5921cd9fd3a82bd9244ece5328c6c95f`. Le HMAC est bien celui attendu :

```
$ echo -n 5921cd9fd3a82bd9244ece5328c6c95f | xxd -r -p | openssl dgst -sha256 -hmac "LUM{0qvYH:
    QI?K6}"
(stdin)= 504861089d10a8d5900cdf3bad962004282e73c20d2d5ab95b67ae6b3356748b
$ dd if=secret.lzma.encrypted bs=112 count=1 2>/dev/null | openssl enc -d -aes-128-ecb -K $(echo
     -n ___SSTIC_2017___ | xxd -p) -nopad
==BEGIN PASSWORD HMAC==
504861089d10a8d5900cdf3bad962004282e73c20d2d5ab95b67ae6b3356748b
==END PASSWORD HMAC==
```

La clé fonctionne pour valider l'épreuve.

4.9 Troisième clé LUM

Une fois le fichier déchiffré, on obtient une image :

Si on regarde les données EXIF de l'image on obtient le dernier LUM :

```
$ identify -format '%[EXIF:*]' riscy_zones/secret.jpg
exif:ImageDescription=Congratulations : YHZ{+g%Yi.vzG8Z}
$ identify -format '%[EXIF:*]' riscy_zones/secret.jpg | rot13
rkvs:VzntrQrfpevcgvba=Pbatenghyngvbaf : LUM{+t%Lv.imT8M}
```

On peut alors valider le LUM suivant : `LUM{+t%Lv.imT8M}`.

5 Unstable Machines

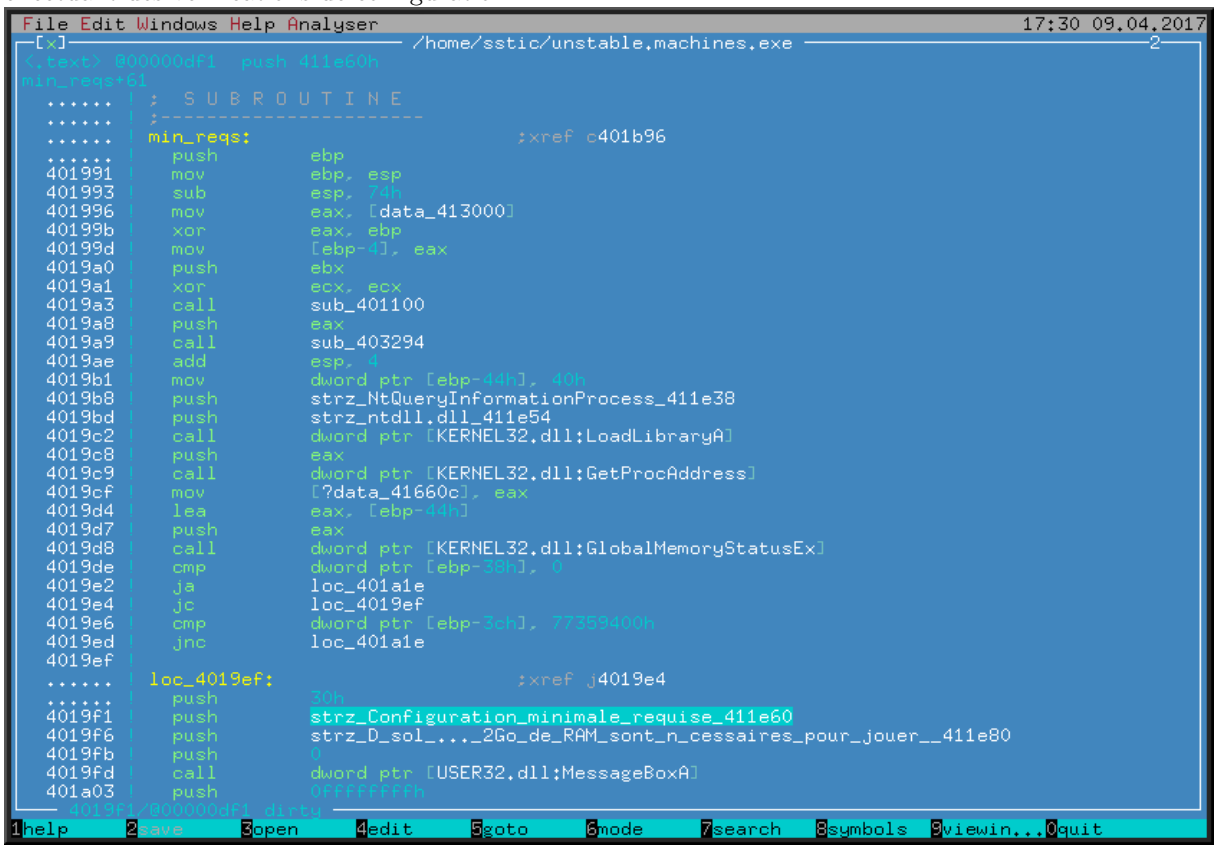
Avec les deux clés précédentes validées, la dernière épreuve se débloque. On obtient un PE32 : `unstable.machines.exe`.

La première étape est de réussir à le lancer, le binaire ne semblant pas trop apprécier `wine`, qui plus est dans une VM limitée :

On peut retrouver la configuration minimale requise à l'aide de `strings` :

```
$ strings ../sstic/unstable.machines.exe | grep -A6 'Configuration minimale'
Configuration minimale requise
Désolé... 2Go de RAM sont nécessaires pour jouer!
Désolé... Un OS 64-bits est nécessaire pour jouer!
Désolé... 50Go de disque dur sont nécessaires pour jouer!
Désolé... Un OS plus récent que Vista est nécessaire pour jouer!
Désolé... Au moins 2 coeurs processeur sont nécessaires pour jouer!
Real machine
```

Le binaire faisant près d'1MB, il est préférable de passer sur un désassembleur intéractif, tel que `HT Editor`. En suivant l'utilisation de la chaîne de caractère du message d'erreur, on arrive sur une fonction effectuant des vérifications de configuration :



La fonction `main` du binaire ne pose pas de problème à identifier, l'exécutable étant normal de ce point de vue. En partant du point d'entrée, on peut reconnaître l'appel à `main` par l'instruction `push 400000h`

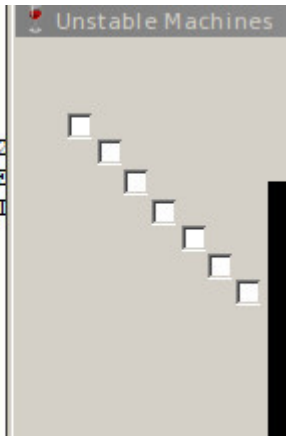
qui précède l'appel.
La fonction qui vérifie les prérequis est la première à être appelée depuis `main`. Patcher l'appel ne devrait pas changer le comportement de l'application. Un assembleur est intégré à `HT Editor` pour effectuer le patch :

```
******  ;-----------------------
******  ;  S U B R O U T I N E
******  ;-----------------------
******  main:                           ;xref c4033ed
******  55                               push        ebp
401b91  8bec                             mov         ebp, esp
401b93  83ec1c                           sub         esp, 1ch
401b96  33c0                             xor         eax, eax
401b98  90                               nop
401b99  90                               nop
401b9a  90                               nop
401b9b  681b150000                       push        151bh
401ba0  6844424100                       push        data_414244
401ba5  e8d6020000                       call        sub_401e80
401baa  83c408                           add         esp, 8
```

Le programme se lance sans problème, l'interface est cependant cryptique :

En jouant avec les cases à cocher, on peut s'apercevoir que l'application semble attendre une séquence bien particulière de cochage et décochage. Afin de déterminer la séquence, on peut s'aider de `PE Explorer` pour obtenir les identifiants des éléments composants la fenêtre :

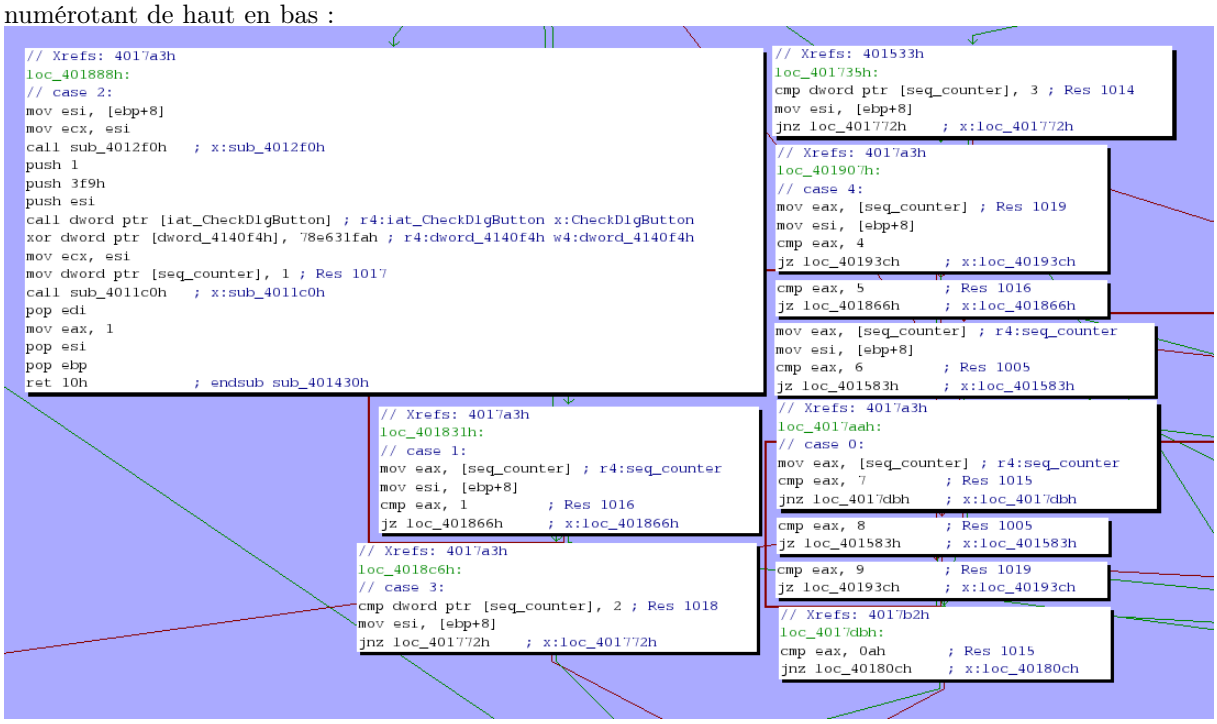
```
CAPTION "Unstable Machines"
LANGUAGE LANG_FRENCH, SUBLANG_FRENCH
FONT 8, "MS Shell Dlg"
{
  DEFPUSHBUTTON   "Yo, dude!", 1, 192, 205, 82, 22
  CONTROL "", 1002, "STATIC", SS_BITMAP | SS_CENTE
  EDITTEXT   "", 1004, 15, 178, 244, 12, NOT WS_VI
  AUTOCHECKBOX   "Check1", 1005, 17, 23, 8, 8
  AUTOCHECKBOX   "Check1", 1014, 27, 31, 8, 8
  AUTOCHECKBOX   "Check1", 1015, 36, 40, 8, 8
  AUTOCHECKBOX   "Check1", 1016, 45, 49, 8, 8
  AUTOCHECKBOX   "Check1", 1017, 55, 57, 8, 8
  AUTOCHECKBOX   "Check1", 1018, 64, 66, 8, 8
  AUTOCHECKBOX   "Check1", 1019, 73, 74, 8, 8
```



On trouve l'appel à la création de fenêtre, `CreateDialogParam`, plus loin dans la fonction `main`. On peut en déduire la fonction de gestion des événements :

```
401bce   mov     [?data_416628], eax
401bd3   call    dword ptr [COMCTL32.dll:0011]
401bd9   push    0
401bdb   push    DialogFunc
401be0   push    0
401be2   push    65h
401be4   push    dword ptr [ebp+8]
401be7   call    dword ptr [USER32.dll:CreateDialogParamW]
401bed   push    dword ptr [ebp+14h]
401bf0   mov     [ebp+8], eax
```

Une représentation sous forme de graphe repositionnable, que l'on peut obtenir avec `metasm`, aide pour trouver la séquence. On peut noter l'utilisation d'un compteur situé à l'adresse 0x416624 pour toutes les ressources associées aux cases à cocher. Après avoir mis les `basic blocs` dans l'ordre attendu, on obtient la séquence : 1017 1016 1018 1014 1019 1016 1005 1015 1005 1019 1015, soit : 5 4 6 2 7 4 1 3 1 7 3 en numérotant de haut en bas :



Le programme demande alors un mot de passe :

5.1 Premier LUM

D'autres opérations sont effectuées pendant la réalisation de la séquence, notamment le calcul d'un mot de 32 bits situé à l'adresse 0x4140F4 initialement à 0xFFFFFFFF. Pendant le calcul de ce mot, le résultat intermédiaire est utilisé pour appliquer une opération de ou-exclusif sur une constante de 128 bits présente dans les données du programme à l'adresse 0x4140e4, puis copiée à l'adresse 0x416614. Le calcul remis dans l'ordre donne :

```
unsigned char d_4140e4[] = {
    0x64, 0x14, 0xf1, 0x2c, 0x56, 0x77, 0xb3, 0x26,
    0x1d, 0x98, 0x3c, 0xd3, 0x09, 0xa4, 0x81, 0x80 };
unsigned int cst = 0xffffffff, *ptr32;
int i;

ptr32 = (unsigned int *)d_4140e4;
cst ^= 0x78E631FA;
cst ^= 0x664A215F;
cst -= 0x01010101;
ptr32[2] ^= cst;
cst += 0x776952CF;
ptr32[0] ^= cst;
cst += 0x12FA9641;
ptr32[1] ^= cst;
cst ^= 0x664A215F;
cst += 0xF4B02F4B;
cst |= 0x01010101;
cst += 0xF4B02F4B;
cst += 0x12FA9641;
ptr32[1] ^= cst;
cst ^= 0xF4B01F4B;
ptr32[3] ^= cst;

printf("Result %08x\n", cst);

for (i=0;i<16;i++) {
    printf("%c", d_4140e4[i]);
}
printf("\n");
```

La valeur finale est 0xfde7f446, que l'on va noter pour la suite. Une clé LUM apparaît aussi dans le mot de 128 bits après les opérations de ou-exclusif :

```
$ ./lum1
Result fde7f446
LUM{2KREDvn3OPf}
```

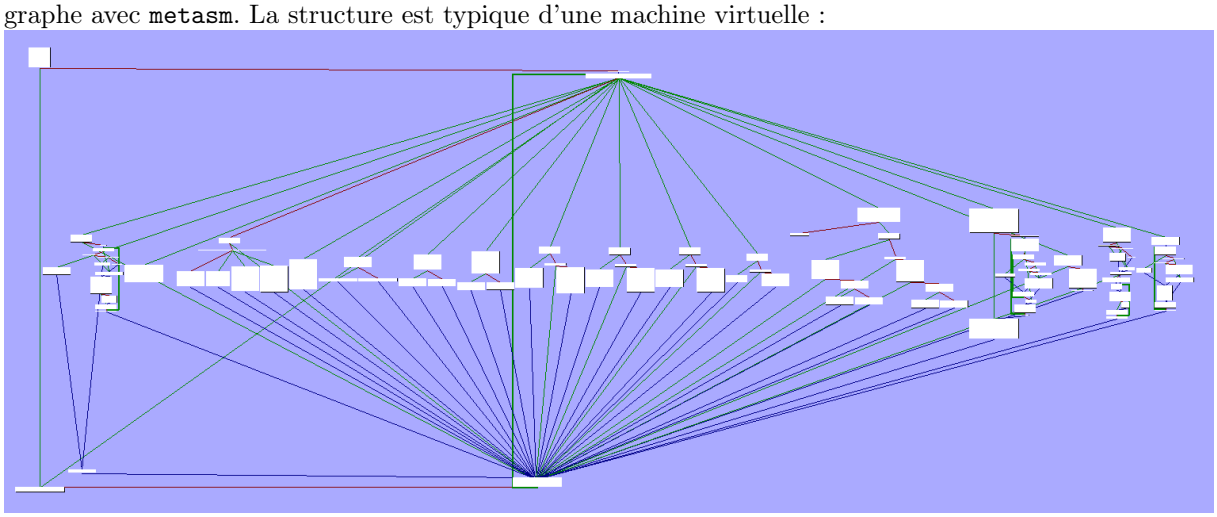
On peut alors valider le LUM suivant : `LUM{2KREDvn3OPf}`.

5.2 Première machine virtuelle

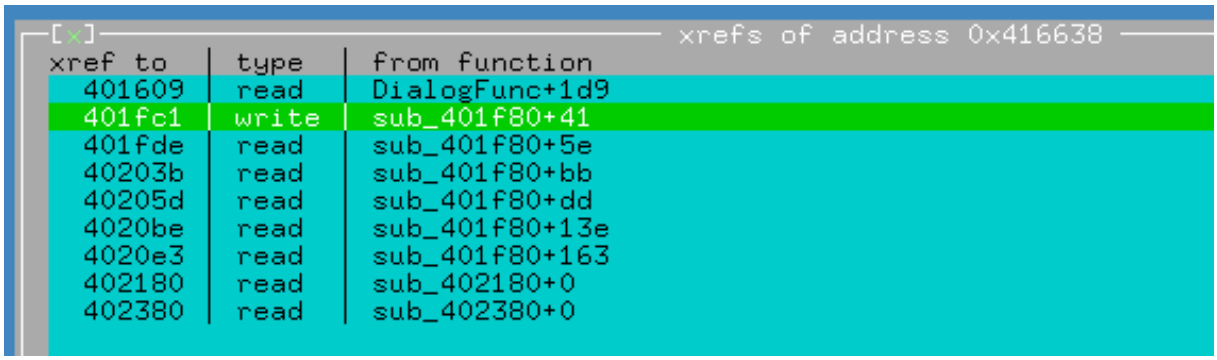
Le programme demande maintenant un mot de passe, on peut s'attendre à ce que ce mot de passe soit la clé de l'épreuve. Pour la suite de l'analyse il faut identifier la fonction en charge de la vérification, ce qui se fait en suivant l'import `GetDlgItemTextA`.
Le code est assez direct, le texte entré dans la boite d'édition est copié à l'offset 0x1020 d'un pointeur du troisième élément d'une liste chaînée dont l'adresse de base est située en 0x416638. La fonction `check_password` est alors appelée. La suite du code applique un ou-exclusif sur 32 octets de données présentes à l'adresse 0x4140C4, par bloc de 32 bits, avec la constante précédemment calculée : 0xfde7f446. Le résultat est alors comparé avec le contenu de l'adresse qui servait au stockage du contenu de la boite d'édition, et donc du mot de passe entré.
L'assembleur annoté dans `HT Editor` donne :

```
File Edit Windows Help Analyser                                    10:40 10.04.2017
[x]                        /home/sstic/unstable.machines.exe                      2
<.text> @00000a16  mov eax,[eax+10h]
DialogFunc+1e6
  401603 !   call        dword ptr [USER32.dll:GetDlgItemTextA]
  401609 !   mov         eax, [?data_416638]
  40160e !   movdqu      xmm0, [?data_4165e8]
  401616 !   mov         eax, [eax+10h]
  401619 !   mov         eax, [eax+10h]
  40161c !   mov         eax, [eax+0ch]
  40161f !   add         eax, 1020h
  401624 !   mov         [input_ptr], eax
  401629 !   movdqu      [eax], xmm0
  40162d !   movdqu      xmm0, [?data_4165f8]
  401635 !   movdqu      [eax+10h], xmm0
  40163a !   call        check_password
  40163f !   movdqu      xmm0, [expected_lo]
  401647 !   mov         ecx, [?data_416628]
  40164d !   mov         edx, ?data_4165e8
  401652 !   mov         eax, [cst_fde7f446]
  401657 !   mov         esi, 1ch
  40165c !   movdqu      [ecx], xmm0
  401660 !   movdqu      xmm0, [expected_hi]
  401668 !   movdqu      [ecx+10h], xmm0
  40166d !   xor         [ecx], eax
  40166f !   xor         [ecx+4], eax
  401672 !   xor         [ecx+8], eax
  401675 !   xor         [ecx+0ch], eax
  401678 !   xor         [ecx+10h], eax
  40167b !   xor         [ecx+14h], eax
  40167e !   xor         [ecx+18h], eax
  401681 !   xor         [ecx+1ch], eax
  401684 !   mov         eax, [input_ptr]
  401689 !   movdqu      xmm0, [eax]
  40168d !   movdqu      [?data_4165e8], xmm0
  401695 !   movdqu      xmm0, [eax+10h]
  40169a !   movdqu      [?data_4165f8], xmm0
  4016a2 !
  ...... ! loc_4016a2:                          :xref j4016b1
  ...... !   mov         eax, [edx]
  4016a4 !   cmp         eax, [ecx]
  4016a6 !   jnz         loc_4016f8
  4016a8 !   add         edx, 4
  4016ab !   add         ecx, 4
  4016ae !   sub         esi, 4
  4016b1 ! password match
  ...... !   jnc         loc_4016a2
  401616/000000a16 dirty
1help 2save 3open 4edit 5goto 6mode 7search 8symbols 9viewin... 0quit
```

La fonction `check_password` implémente un `switch-case`, le mieux est de la visualiser sous forme de graphe avec `metasm`. La structure est typique d'une machine virtuelle :



Au début de cette fonction, deux sous-fonctions sont appelées. La première située en 0x402380 met à 0 une zone mémoire située en 0x416640 et calcule deux valeurs, stockées en 0x416660 et 0x41665C, par rapport à des données présentes dans la liste chaînée précédemment identifiée en 0x416638.
Le calcul suit le même motif dans les deux cas : $y = (k \oplus (k + x)) - k$.
Cette fonction est sa propre réciproque, elle sert à obfusquer la pile et les registres de la machine virtuelle.
La deuxième sous-fonction appelée, située à l'adresse 0x402180, retourne 1 octet lié à l'adresse stockée dans le registre situé en 0x416660 et l'incrémente. Cette fonction étant appelée à chaque tour de boucle, on peut en déduire que ce registre est le `program counter` de la VM.
Pour la suite de l'analyse il faut retrouver la structure derrière les éléments de la liste chaînée. En utilisant les cross-référence on retombe sur la fonction qui l'initialise :



Après annotation, on peut voir apparaître la fonction en charge de l'allocation des éléments ainsi qu'une copie de données effectuée dans une zone mémoire pointée par un des champs de l'élément :

```
401fb8 !   push        ecx
401fb9 !   call        alloc_ll_elt
401fbe !   add         esp, 0ch
401fc1 !   mov         [ll_head], eax
401fc6 !   mov         edx, [ebp-8]
401fc9 !   movzx       eax, byte ptr [edx]
401fcc !   cmp         eax, 0feh
401fd1 !   jz          loc_401ffe
401fd3 !   mov         ecx, [ebp-0ch]
401fd6 !   push        ecx
401fd7 !   mov         edx, [ebp-8]
401fda !   add         edx, 9
401fdd !   push        edx
401fde !   mov         eax, [ll_head]
401fe3 !   mov         ecx, [eax+0ch]
401fe6 !   push        ecx
401fe7 !   call        memmove
401fec !   add         esp, 0ch
```

Le prototype de la fonction en charge de l'allocation des éléments est : `struct ll_elt *init_elt(char id, uintptr_t vaddr, size_t size);`. L'allocation de la zone mémoire elle-même est prise en charge par la fonction située à l'adresse 0x401D70, il est important de noter que 4096 octets supplémentaires sont alloués systématiquement. Cette fonction initialise la mémoire allouée selon un algorithme particulier :

```
// RE init VM1 memory @401D70
unsigned int tmp = 0xf4b01f4b, *mem_area32 = malloc(size);
for(i = 0; i < (size + 4096) / 4; i++) {
    unsigned char f1 = 32 - 3 * i;
    unsigned char f0 = 3 * i;
    unsigned int d1, d0;
    d1 = tmp >> f1;
    d0 = tmp << f0;
    tmp = d1 ^ 0xf4b01f4b ^ d0;
    mem_area32[i] = tmp;
}
```

Trois zones sont ainsi créées, les adresses sont issues d'une opération de ou-exclusif par rapport à des valeurs calculées qui donnent : 0xf4b00f4b, 0xf4b01f4b, 0xf4b02f4b. Le prénom de l'auteur de cette épreuve étant très probablement Fabien :

Identifiant	Addresse virtuelle	Taille
0x85	0x05571C00	1024
0xFE	0x05572000	256
0x61	0x05572100	4352

Le second segment n'est pas réécrit avec les données du binaire, tandis que les deux autres zones contiennent respectivement le contenu des adresses 0x41424D et 0x41465F.
Maintenant que l'on connaît les adresses virtuelles de la VM, on peut vérifier l'état des registres identifiés à l'initialisation avec `gdb` :

```
(gdb) b *0x40291e
Breakpoint 1 at 0x40291e
(gdb) c
Continuing.

Thread 1 "unstable.machin" hit Breakpoint 1, 0x0040291e in ?? ()
1: x/i $eip
=> 0x40291e:    call   0x402180
(gdb) x/1wx 0x416660
0x416660:       0x00f0efba
(gdb) p/x (0xfde7f446 ^ (*0x416660 + 0xfde7f446)) - 0xfde7f446
$8 = 0x5571c00
(gdb) p/x (0xfde7f446 ^ (*0x41665C + 0xfde7f446)) - 0xfde7f446
$9 = 0x5572068
```

On peut en déduire que le registre stocké à l'adresse 0x41665C est le pointeur de pile, dont la zone mémoire correspond au second segment. On retrouve l'adresse du `PC` au début du premier segment. Ce qui signifie que le code de la machine virtuelle se trouve à l'adresse 0x41424D de l'application.
On peut aussi en déduire que la zone mise à 0 au début de la fonction de vérification du mot de passe correspond à l'initialisation de 8 registres génériques.
Le dernier segment correspond aux données, on avait vu précédemment que le mot de passe entré par l'utilisateur était stocké dans cette zone, on le retrouve d'ailleurs dans le debugger :

```
(gdb) x/s *(*(*(*0x416638+16)+16)+12)+0x1020
0x130ec8:       "GOLDFIST"
```

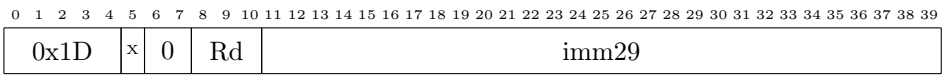
La fonction responsable de la machine virtuelle étant connu, il faut maintenant déterminer le jeu d'instructions associé. Le travail est rendu fastidieux du fait de l'obfuscation systématique des registres ainsi que par des morceaux de code inutiles comme l'ouverture d'un fichier `C:\Rayman\Rayman.ini` et des instructions mal décodées du fait de sauts d'un octet qu'aucun désassembleur ne semble gérer correctement.

5.3 Jeu d'instruction de la première VM

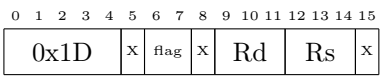
Au final on arrive à 17 instructions, dont le codage est le suivant :

Assignation et stockage :

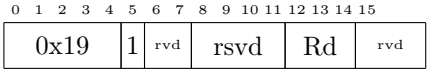
$R_d := imm29$



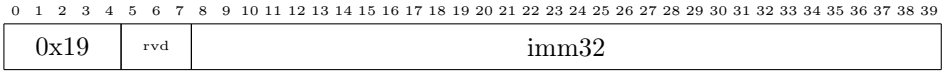
$IF[flag = 1]$
$THEN R_d := R_s$
$IF[flag = 2]$
$THEN R_s := (R_d)$
$IF[flag = 3]$
$THEN R_d := (R_s)$



$SP := SP + 4$
$(SP) := R_d$

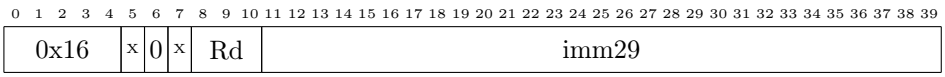


$SP := SP + 4$
$(SP) := imm32$

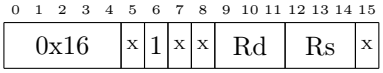


Arithmétique et logique :

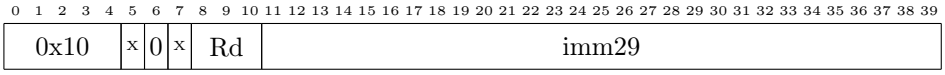
$R_d := R_d + imm29$



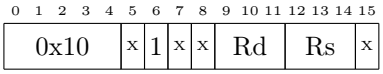
$R_d := R_d + R_s$



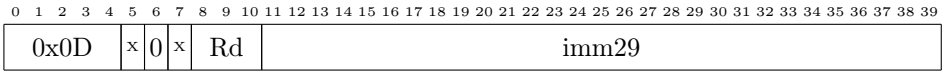
$R_d := R_d - imm29$



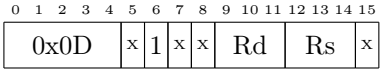
$R_d := R_d - R_s$



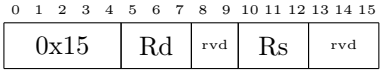
$R_d := R_d \oplus imm29$



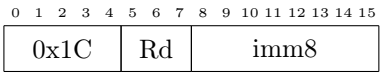
$R_d := R_d \oplus R_s$



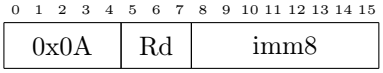
$R_d := R_d \ll [(R_s) \wedge 255]$



$R_d := R_d \gg imm8$

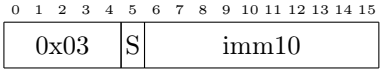


$R_d := R_d \wedge imm8$

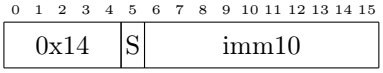


Appel et saut :

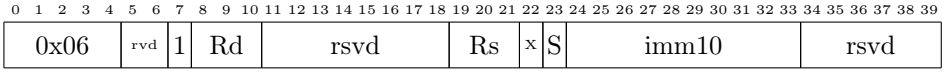
$SP := SP + 4$
$(SP) := PC$
$IF[s = 0]$
$THENPC := PC + imm10$
$IF[s = 1]$
$THENPC := PC - imm10$



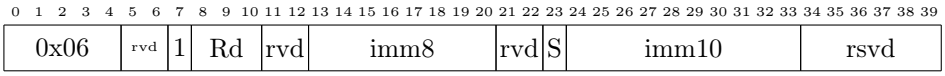
$IF[s = 0]$
$THENPC := PC + imm10$
$IF[s = 1]$
$THENPC := PC - imm10$



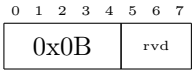
$IF[[s = 0] \wedge [R_d = R_s]]$
$THENPC := PC + imm10$
$IF[[s = 1] \wedge [R_d = R_s]]$
$THENPC := PC - imm10$



$IF[[s = 0] \wedge [R_d = imm8]]$
$THENPC := PC + imm10$
$IF[[s = 1] \wedge [R_d = imm8]]$
$THENPC := PC - imm10$



$PC := (SP)$
$SP := SP - 4$



$HALT$

0 1 2 3 4 5 6 7	
0x09	rvd

Divers :

$R_0 := Hash((SP), 4 \times R_0)$

0 1 2 3 4 5 6 7	
0x0F	rvd

$FOR[i := 0 \dots R_0]$
$R_{i+1} := Imgdata()$
$Img_y := Img_y + 1$

0 1 2 3 4 5 6 7	
0x12	rvd

$R_0 := FSM((SP), (SP + 4))$

0 1 2 3 4 5 6 7	
0x05	rvd

$S := R_0 \wedge 0x8000$
$IF[s = 0]$
$THENImg_y := Img_y + [R_0 \wedge 0x7FFF]$
$IF[s = 1]$
$THENImg_y := Img_y - [R_0 \wedge 0x7FFF]$

0 1 2 3 4 5 6 7	
0x00	rvd

5.4 Code assembleur de la première VM

On peut maintenant désassembler le code de la machine virtuelle :

```
5571c00:    mov     r0, #0x5572100
5571c05:    add     r0, #0x1000
5571c0a:    call    <5571c0e>       ; func_00
5571c0c:    jump    <5571cef>       ; loop
```

Le code commence par faire pointer `r0` sur le segment de données, à l'offset 0x1000, puis appelle une première fonction.
La fonction en question fait appel à l'instruction de hachage identifiée précédemment :

```
func_00:
5571c0e:    mov     r1, r0
5571c10:    push    #0x5b1c63ed
5571c15:    push    #0x7ebb8eb3
5571c1a:    push    #0xb8ab7606
5571c1f:    push    #0x8a06fff9
5571c24:    push    #0xf0a21668
5571c29:    push    #0x962e5304
5571c2e:    push    #0x36644553
5571c33:    push    #0x40375a0
5571c38:    mov     r0, #0x8
5571c3d:    hash    r0, [sp], 4*r0
5571c3e:    stw     r0, [r1]
```

```
5571c40:    push    #0xe9b2165a
5571c45:    push    #0xe3a7f8b
5571c4a:    push    #0xf4c919ff
5571c4f:    push    #0xfd067499
5571c54:    push    #0x2868793e
5571c59:    push    #0xe2a3e528
5571c5e:    push    #0x7b1e58e8
5571c63:    push    #0x1951d388
5571c68:    mov     r0, #0x8
5571c6d:    hash    r0, [sp], 4*r0
5571c6e:    add     r1, #0x4
5571c73:    stw     r0, [r1]
5571c75:    push    #0x4b0b009c
5571c7a:    push    #0x32b16ace
5571c7f:    push    #0x4b0b565e
5571c84:    push    #0x4b0b57c1
5571c89:    push    #0x4b0b57ce
5571c8e:    push    #0x4b0b5730
5571c93:    push    #0xa2e81d8a
5571c98:    push    #0x4b0b565e
5571c9d:    push    #0x4b0b5146
5571ca2:    mov     r0, #0x9
5571ca7:    hash    r0, [sp], 4*r0
5571ca8:    add     r1, #0x4
5571cad:    stw     r0, [r1]
5571caf:    push    #0x5b01ef00
5571cb4:    push    #0xf4bb1
5571cb9:    push    #0x5c80eb85
5571cbe:    push    #0xb602e003
5571cc3:    push    #0x508fa8d5
5571cc8:    push    #0xc5fc8137
5571ccd:    push    #0x43f919c3
5571cd2:    push    #0xd8aa82e8
5571cd7:    mov     r0, #0x8
5571cdc:    hash    r0, [sp], 4*r0
5571cdd:    add     r1, #0x4
5571ce2:    stw     r0, [r1]
5571ce4:    add     r7, #0x80
5571ce9:    add     r7, #0x4
5571cee:    ret
```

La suite du code ressemble de nouveau à un `switch-case` à l'intérieur d'une boucle qui commence par charger trois octets depuis l'image de l'application. Le début du code désassemblé est le suivant :

```
loop:
5571cef:    mov     r0, #0x3
5571cf4:    steg    r0, [r1:]               ; y+=3;
5571cf5:    jeq     r1, #0x78, <5571d3b>
5571cfa:    jeq     r1, #0xfe, <5571d4b>
5571cff:    jeq     r1, #0x33, <5571d69>
5571d04:    jeq     r1, #0x12, <5571d9b>
5571d09:    jeq     r1, #0xa1, <5571da6>
5571d0e:    jeq     r1, #0xaf, <5571dc7>
5571d13:    jeq     r1, #0x8e, <5571de9>
5571d18:    jeq     r1, #0x13, <5571e0b>
5571d1d:    jeq     r1, #0xbb, <5571e2d>
5571d22:    jeq     r1, #0x7c, <5571ed9>
5571d27:    jeq     r1, #0x32, <5571e77>
5571d2c:    jeq     r1, #0x2f, <5571ea7>
5571d31:    jeq     r1, #0xdc, <5571ef8>    ; the_end
5571d36:    jeq     r1, #0x59, <5571e52>
```

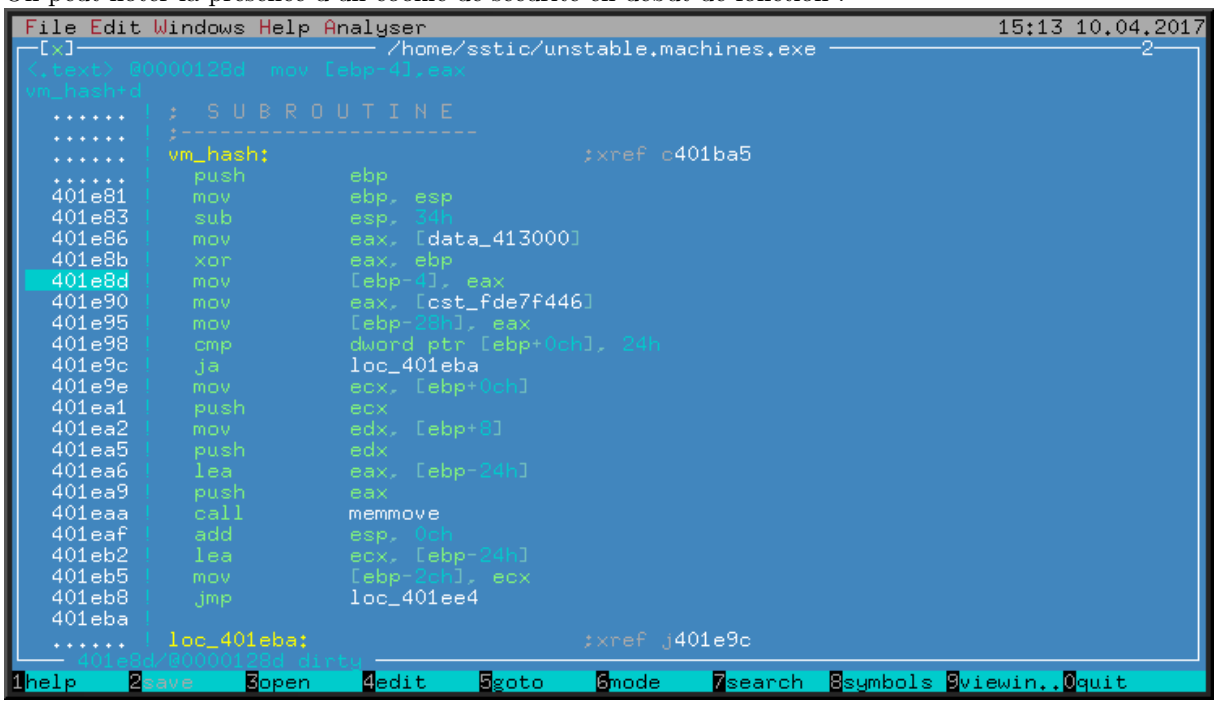
On peut regarder le premier cas pour vérifier s'il s'agit bien d'une machine virtuelle :

```
5571d3b:    mov     r4, #0x5572100
5571d40:    add     r4, #0x1040
5571d45:    add     r4, r2                  ; arg_1
5571d47:    stw     r3, [r4]                ; *(0x5573140+arg_1) = arg_2
5571d49:    jump    <5571cef>               ; loop
```

Si on considère 0x5573140 comme l'adresse de stockage des registres d'une machine virtuelle, alors on obtient pour l'opcode 0x78 : $r_{arg1} = arg2$.

5.5 Instruction de hachage et cookie de sécurité

L'instruction `hash` obtenue précédemment est implémentée par la fonction située à l'adresse 0x401E80. On peut noter la présence d'un cookie de sécurité en début de fonction :



La fonction de hachage est relativement simple, après avoir copié le contenu à condenser en mémoire, le code effectue quelques opérations de ou-exclusif, décalage, multiplication et ou-logique sur chacun des blocs disponibles. La fonction de hachage réécrite en C donnerait :

```
    unsigned int *ptr32 = arg1, val;
    int i;

    val = cst_fde7f446 ^ 0xf4b03f4b;
    for (i = 0; i < arg2 * 4; ++i) {
        ptr32[i] ^= 0xf4b00f4b;
        val = (32 * (val ^ ptr32[i])) | ((val ^ ptr32[i]) >> 27);
    }
```

Le calcul direct du résultat de la fonction de hachage pour ces constantes donne les valeurs suivantes :

```
./hash_init
Hash: 0x9aacc69b 0x89a31d8a 0x33ae5090 0xb0a5ce54
```

On peut aller vérifier le résultat de ces calculs avec `gdb` :

```
(gdb) x/4wx (*(*(*(*0x416638+16)+16)+12)+0x1000)
0x130ea8:       0x9aacc69b      0x89a31d8a      0xa5b2cd1c      0xb0a5ce54
(gdb)
```

La troisième valeur ne correspond pas, mais c'est aussi la seule valeur calculée sur 9 blocs plutôt que 8 pour les autres.
Il faut aller s'intéresser à la copie des blocs en mémoire pour comprendre. Un tableau est alloué sur la pile, il est situé à l'adresse `ebp-0x24` tandis que le cookie est stocké dans `ebp-0x4`. La condition pour l'appel à `memove` avec le tableau sur la pile est que la taille doit être inférieure ou égale à 0x24, alors que le tableau ne fait que 32 octets :

```
401e98 !   cmp         dword ptr [ebp+0ch], 24h
```

Le troisième appel à cette fonction a pour deuxième argument $9 \times 4 = 36$, ce qui va déclencher une erreur lors de la sortie de la fonction du fait du cookie de sécurité. La fonction de vérification du cookie est située à l'adresse 0x403264. En cas d'erreur, le code suivant est exécuté :

```
...... !   push        ebp
4034b4 !   mov         ebp, esp
4034b6 !   sub         esp, 324h
4034bc !   push        47h
4034be !   call        wrapper_KERNEL32.dll:IsProcessorFeaturePresent_40d1ca
4034c3 !   test        eax, eax
4034c5 !   jnz         loc_4034d1
4034c7 !   mov         esp, ebp
4034c9 !   pop         ebp
4034ca !   pop         ecx
4034cb !   add         esp, 0ch            ; really ?!
4034ce !   pop         eax
4034cf !   ret
```

La valeur 71 donnée en argument de `IsProcessorFeaturePresent` ne correspond à rien de connu, ce qui fait que l'appel va échouer. Le code va retourner après avoir modifié le pointeur de pile. Au moment du `ret`, le pointeur de pile est situé au niveau du tableau ayant provoqué le débordement, les éléments de ce tableau ayant subi une opération de ou-exclusif avec la valeur 0xf4b00f4b.

5.6 Première ROP-chain

On peut recalculer le contenu de la pile en appliquant le ou-exclusif sur les données à condenser :

```
00401a0d
00401d15
e9a356c1
00401c7b
00401c85
00401c8a
00401d15
79fa2185
00404bd7
```

On reconnaît plusieurs adresses de retour, ainsi que deux constantes, probablement utilisées lors de `pop`. Pour reconstruire la chaîne ROP, on peut récupérer le contenu de chacune des adresses jusqu'à arriver sur une opcode de retour (0xC2/3, 0xCA/B). Le résultat après passage de `objdump` donne alors :

```
    mov    ebx,DWORD PTR fs:0x30
    mov    ebx,DWORD PTR [ebx]  ; // 0x00010000 if debug, else 0x00000000
    not    ebx                  ; ebx = 0xffffffff;
    pop    esi                  ; esi = 0xe9a356c1;
    add    eax,esi
    xor    eax,ebx
    ror    eax,0x2a
    sub    eax,ebx
    pop    esi                  ; esi = 0x79fa2185;
    add    eax,esi  ; ret = ROR32((ret + 0xe9a356c1) ^ ebx, 0x2a) - ebx + 0x79fa2185
    push   ecx
```

Une fois ce calcul rajouté à la fin du troisième appel à la fonction de hachage, on obtient bien la même valeur que dans `gdb` : 0xa5b2cd1c. On pourra noter que la combinaison `wine` avec `gdb` ne modifie pas la valeur du PEB du process. Lorsqu'on va chercher le résultat de ce calcul depuis un debugger natif, le résultat du calcul est : 0xa5b3cd5c, qu'il ne faudra surtout pas prendre pour le calcul de la clé.

5.7 Stéganographie

À ce point de l'analyse, on a identifié que la première machine virtuelle en implémente une autre. Les instructions de cette seconde VM se trouvent a priori dans l'image de l'application.

Une des instructions de la première VM permet de modifier une variable globale située à l'adresse 0x41663C, cette même variable est utilisée comme argument `y` pour l'appel à la fonction `GetPixel`. L'analyse du code de la première VM permet de déduire que les instructions de la seconde VM sont codées dans les lignes de l'image. On peut aussi voir que des instructions de saut existent, leur implémentation modifie la valeur courante de la ligne.

La fonction permettant de récupérer des octets dans des lignes de l'image se trouve à l'adresse 0x4023F0. Les données sont stockées dans le bit de poids faible des composantes RVB de l'image. La position et la composante dépendent du numéro de ligne sélectionné. Le premier pixel sélectionné dépend de la valeur courante de la ligne, puis est incrémenté de 42 pour chaque nouveau bit. Pour l'implémentation il faut garder deux choses à l'esprit : les images au format BMP ont l'axe des ordonnées inversé dans l'image, et la taille des lignes est un multiple de 4. L'image faisant 678×306, on obtient l'algorithme suivant :

```
unsigned char getbyte(int fd, int y)
{
    unsigned char c[3], ret = 0;
    int i, x, composante;

    x = ((y ^ 0xfde7f446) + 66) & 0xff;
    composante = (x + 0xfde7f446) % 3;
    for (i=0; i< 8; i++) {
        int rows = ((678 * 3 + 3) / 4) * 4;

        lseek(fd, IMG_OFFSET + (305 - y) * rows + 3 * x, SEEK_SET);
        read(fd, c, 3);
        ret <<= 1;
        ret |= c[2 - composante] & 1;
        x += 42;
        composante++;
        composante %= 3;
    }
    return ret;
}
```

5.8 Second LUM

Une fois l'algorithme de stéganographie implémenté, on peut extraire les données de l'image. A tout hasard, on peut passer le résultat dans `strings` pour voir si un message s'y trouve :

```
$ ./unstegano | strings
LUM{C1UAidv_pzJ}M
BGRn(
```

On peut alors valider le LUM suivant : `LUM{C1UAidv_pzJ}`.

5.9 Seconde machine virtuelle

La lecture du code de la première VM permet de retrouver les instructions de la seconde. Le format des instructions est plus simple que pour la première VM :

Opcode [1B]	Param 1 [1B]	Param 2 [1B]

On peut compter 14 instructions différentes. On peut déjà noter que l'instruction 0x78 correspond au cas par défaut. Si l'opcode courante ne correspond à aucune des instructions comparées, le cas par défaut sera exécuté, soit une assignation de registre par une valeur immédiate. On peut récapituler les instructions trouvées :

Opcode	Mnemonic	C code
0xfe	mov.r	$r_{p1} = r_{p2}$;
0x33	cmp	if (r_{p1} == p2) zflag = 1 ;
0x12	jmp	s = p2»7 ; goto PC - (s*2-1) * (((p2&0x7f)«8) | p1) ;
0xa1	jeq	s = p2»7 ; if (zflag) goto PC - (s*2-1) * (((p2&0x7f)«8) | p1) ;
0xaf	add	r_{p1} += r_{p2} ;
0x8e	sub	r_{p1} -= r_{p2} ;
0x13	xor	r_{p1} ^= r_{p2} ;
0xbb	mov.i	r_0 = password32[r_1] ;
0x7c	mov.c	r_0 = *(0x0557404b) + 2 ;
0x32	fsm	r_0 = fsm(p1, p2) ;
0x2f	mov.h	r_0 = hashres32[((r_1 » r_2)&0x3)]
0xdc	exit	exit() ;
0x59	mov.o	password32[r_1] = r_2 ;
0x78	mov.l	r_{p1} = p2 ;

Le résultat de l'instruction `mov.c` est une valeur constante. L'adresse correspond au troisième segment, et tombe dans la zone initialisée lors de la création de la liste chaînée. On connaît déjà l'algorithme utilisé pour l'initialisation, il s'agit donc juste de retrouver la valeur pour l'offset 0x1fab :

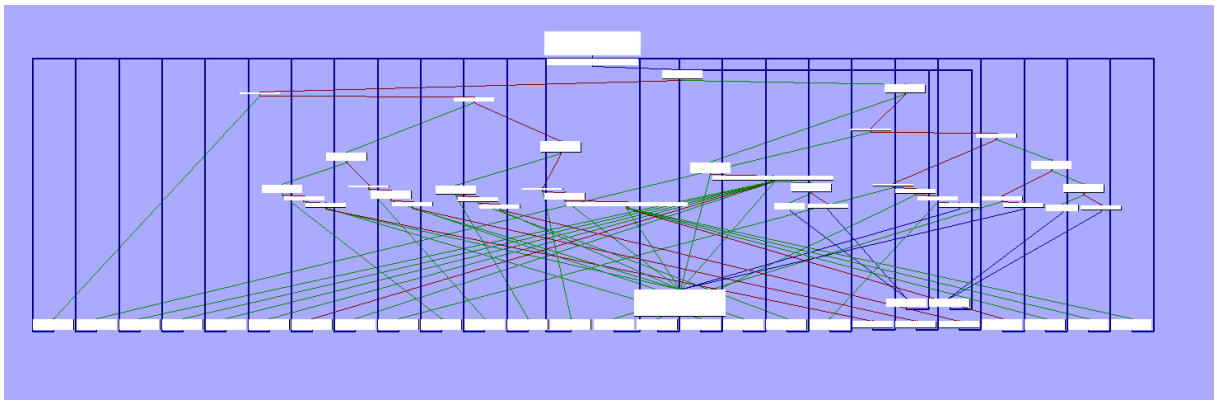
```
// RE init VM1 memory @401D70
unsigned int tmp = 0xf4b01f4b, *mem_area32 = malloc(size);
for(i = 0; i < (size + 4096) / 4; i++) {
    unsigned char f1 = 32 - 3 * i;
    unsigned char f0 = 3 * i;
    unsigned int d1, d0;
    d1 = tmp >> f1;
    d0 = tmp << f0;
    tmp = d1 ^ 0xf4b01f4b ^ d0;
    if (i*4 == 0x1f48) {
        val = tmp >> 24;
    } else if (i*4 == 0x1f4c) {
        val |= tmp << 8;
    }
}
printf("%08x\n", val + 2);
```

Ce qui permet d'obtenir :

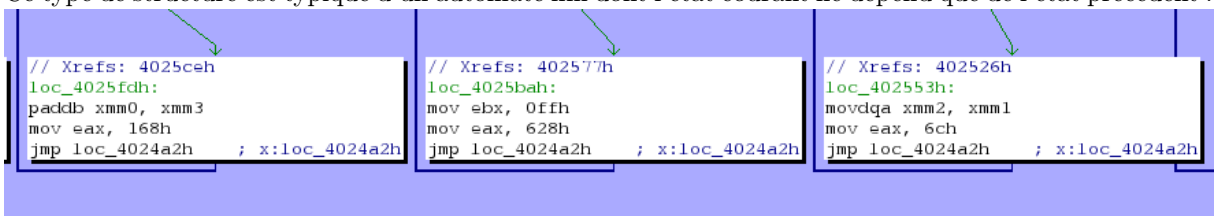
```
$ ./ct2
d2d413b3
```

5.10 Automate fini et SSE4.2

Il ne reste plus qu'une seule instruction inconnue. Cette instruction est implémentée par la fonction située à l'adresse 0x402490.
Une visualisation sous forme de graphe permet d'en reconnaître la structure globale :



Cette fois ce n'est pas une nouvelle machine virtuelle. À chaque tour de boucle une instruction est exécutée en fonction de la valeur du registre `eax`, puis ce registre est mis à jour avec une nouvelle valeur. Ce type de structure est typique d'un automate fini dont l'état courant ne dépend que de l'état précédent :



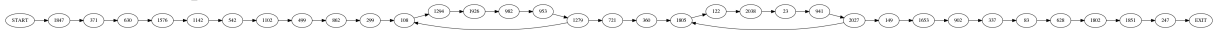
En se basant sur le fait que l'automate exécute une seule instruction par état avant de mettre à jour sont état courant, on peut récupérer l'ensemble des états et instructions exécutées avec `objdump` et `grep` :

```
$ objdump -M intel -S --start-address=0x402490 --stop-address=0x40277e unstable.machines.exe.
    patch | grep -B1 'mov    eax,'
  402496:       89 55 f4                mov    DWORD PTR [ebp-0xc],edx
  402499:       b8 37 07 00 00          mov    eax,0x737
--
  4024ea:       66 0f c2 e8 00          cmpeqpd xmm5,xmm0
  4024ef:       b8 ad 03 00 00          mov    eax,0x3ad
--
# ...
--
  40275d:       66 0f 6f ec             movdqa xmm5,xmm4
  402761:       b8 17 00 00 00          mov    eax,0x17
--
  402772:       94                      xchg   esp,eax
  402773:       8b 45 f0                mov    eax,DWORD PTR [ebp-0x10]
```

On dénombre alors 33 états avec leurs 33 instructions, pour la plupart issues du jeu d'instructions SSE4.2. On peut aussi noter l'avant dernière utilisation du registre `eax` dans un échange avec le pointeur de pile, qui est une technique utilisée dans le but de mettre en échec l'analyse statique auotmatique du binaire. Bien entendu, `objdump` n'est pas concerné par ce genre de tricks.
Deux blocs ne correspondent pas aux autres, les deux cas correspondent au même type de code assembleur : un test est fait sur le registre `bx` et le nouvel état est assigné en fonction du fait qu'il soit nul ou non. Ces deux états permettent d'implémenter des boucles.

Pour reconstituer le code de manière plus claire, on peut soit faire de l'analyse statique, par exemple en annotant la sortie du désassembleur puisque la fonction est relativement courte, ou soit le faire dynamiquement avec un script pour `gdb`.

Au final le choix s'est porté sur l'analyse statique lors de la résolution de l'épreuve, car à ce moment précis je n'avais pas accès à un debugger valable tel que `gdb`. L'idée était de retrouver la valeur de l'état courant permettant d'arriver à chaque `basic block`. Après avoir reconstruit le graphe associé avec `dot`, on obtient la séquence suivante :



Une fois les instructions remisent dans l'ordre, on obtient ce pseudo-code assembleur :

```
mov    DWORD PTR [ebp-0xc],edx      ; ebp-0xc = arg0
mov    BYTE PTR [ebp-0x8],cl        ; ebp-0x8 = arg1
xorpd  xmm4,xmm4                    ; xmm4 = 0
movd   xmm0,DWORD PTR [ebp-0x8]     ; xmm0 = arg1
mov    ebx,255
movd   xmm3,ebx
pand   xmm0,xmm3                    ; xmm0 &= 0xff
mov    ebx,1
movd   xmm1,DWORD PTR [ebp-0xc]     ; xmm1 = arg0
movd   xmm3,ebx
pshufd xmm1,xmm1,0x0                ; xmm1 = REP(arg0)
movdqa xmm2,xmm1                    ; xmm2 = xmm1
{
    pslld  xmm1,xmm3                ; xmm1 <<= 1
    paddb  xmm4,xmm3                ; xmm4++
    movdqa xmm5,xmm4
    cmpeqpd xmm5,xmm0               ; xmm4 == arg1
    pextrw ebx,xmm5,0x0
    test   bx,bx
}                                   ; arg0 << arg1
paddb  xmm0,xmm3                    ; xmm0 = arg1 + 1
xorpd  xmm4,xmm4                    ; xmm4 = 0
{
    psrld  xmm2,xmm3                ; xmm2 >>= 1
    paddb  xmm4,xmm3                ; xmm4++
    movdqa xmm5,xmm4
    cmpeqpd xmm5,xmm0               ; xmm4 == arg1 + 1
    pextrw ebx,xmm5,0x0
    test   bx,bx
}                                   ; arg0 >> (arg1 + 1)
punpckhdq xmm1,xmm2
punpckhdq xmm2,xmm1
xorpd  xmm1,xmm2                    ; xmm1 = REP((arg0 << arg1) ^ (arg0 >> (arg1 + 1)))
pextrw ebx,xmm1,0x0
mov    DWORD PTR [ebp-0x14],ebx     ; lower16
pextrw ebx,xmm1,3
shl    ebx,16
mov    DWORD PTR [ebp-0x10],ebx     ; upper16
mov    eax,DWORD PTR [ebp-0x10]
xor    eax,DWORD PTR [ebp-0x14]     ; eax = (arg0 << arg1) ^ (arg0 >> (arg1 + 1))
```

La fonction est en réalité très simple et fait l'opération suivante :

```
unsigned int fsm(unsigned int val, char shift)
{
    return (val << shift) ^ (val >> (shift + 1));
}
```

On peut désormais reconstituer le code effectif résultant des deux machines virtuelles et de l'automate fini. Le code travaille sur deux mots de 32 bits de la clé entrée par l'utilisateur à chaque tour de boucle principale, pour un total de 4 tours, soit 256 bits en tout. Pour chaque paire de mots, une boucle intérieure de 64 tours effectue une série d'opérations, toutes étant inversibles. Le pseudo-code assembleur final est le suivant :

```
0   mov r7, #0x00
loop:
3   cmp r7, #0x04
6   jeq <a5>                ; exit
9   mov r1, r7
c   add r1, r1
f   mov r0, password32[r1]
12  mov r3, r0
15  mov r8, #0x01
18  add r1, r8
1b  mov r0, password32[r1]
1e  mov r4, r0
21  mov r5, #0x00
24  mov r6, #0x00
innerloop:
27  cmp r6, #0x40
2a  jeq <84>                ; outinner
2d  mov r1, r4
30  mov r2, #0x04
33  mov r0, (r1 << r2) ^ (r1 >> (r2 + 1))
36  mov r8, r0
39  add r8, r4
3c  mov r1, r5
3f  mov r2, #0x00
42  mov r0, hashres32[(r1 >> r2)&0x3]
45  mov r9, r0
48  add r9, r5
4b  xor r8, r9
4e  add r3, r8
51  mov r0, #0xd2d413b3
54  add r5, r0
57  mov r1, r3
5a  mov r2, #0x04
5d  mov r0, (r1 << r2) ^ (r1 >> (r2 + 1))
60  mov r8, r0
63  add r8, r3
66  mov r1, r5
69  mov r2, #0x0b
6c  mov r0, hashres32[(r1 >> r2)&0x3]
6f  mov r9, r0
72  add r9, r5
75  xor r8, r9
78  add r4, r8
7b  mov r8, #0x01
7e  add r6, r8
81  jmp <27>                ; innerloop
outinner:
84  mov r2, r3
87  mov r1, r7
8a  add r1, r1
8d  mov password32[r1], r2
90  mov r2, r4
93  mov r8, #0x01
96  add r1, r8
99  mov password32[r1], r2
9c  mov r8, #0x01
9f  add r7, r8
a2  jmp <3>                 ; loop
exit:
a5  exit
```

5.11 Inversion du code de la deuxième VM

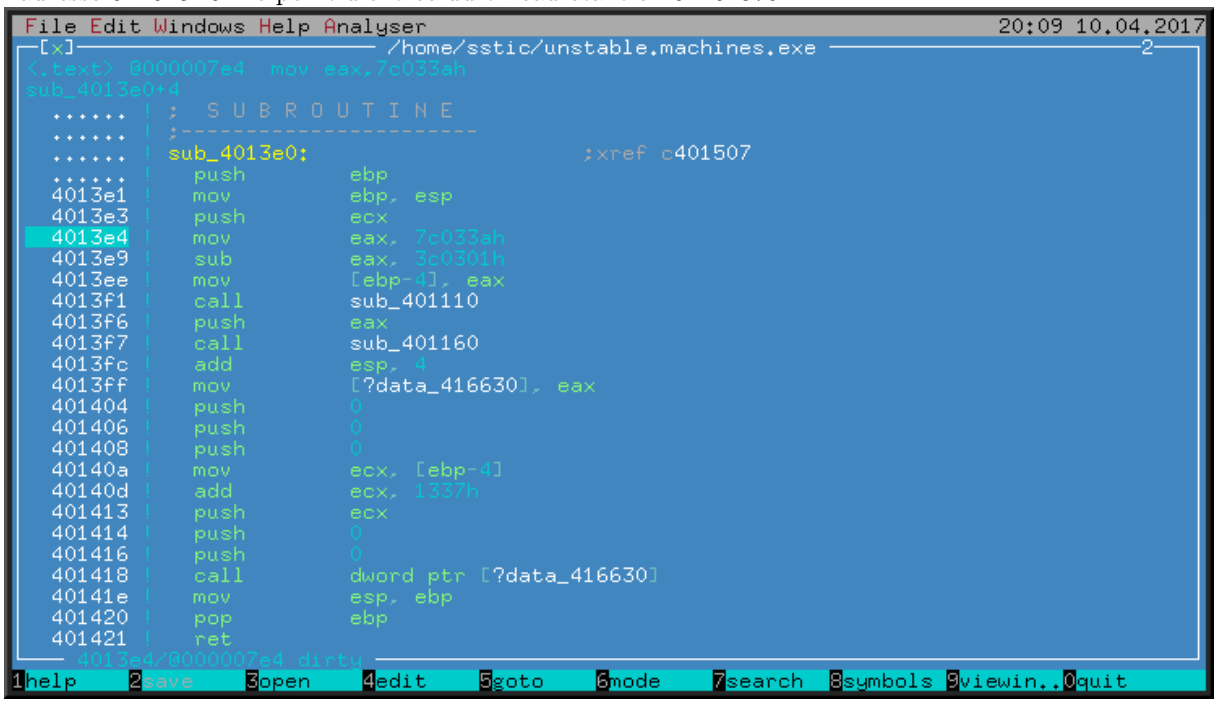
Après simplification du code assembleur et inversion de la fonction, on obtient le code C suivant permettant de calculer une clé par rapport à un résultat attendu :

```
unsigned int lut4[] = { 0x9AACC69B, 0x89A31D8A, 0xa5b2cd1c, 0xB0A5CE54 };
unsigned int val, tmp, input[8], output[9];
int i;

// RE VM2 from stegano data in BMP
for (i = 3; i >= 0; i--) {
    unsigned int r3, r4, r5;
    int j;
    r3 = input[2*i];
    r4 = input[2*i+1];
    r5 = 64*0xd2d413b3;
    for (j = 0; j < 64; j++) {
        val = lut4[(r5>>0xb)&0x3] + r5;
        r5 -= 0xd2d413b3;
        tmp = lut4[r5&0x3] + r5;
        r4 -= (val ^ (((r3 << 4) ^ (r3 >> 5)) + r3));
        r3 -= (tmp ^ (((r4 << 4) ^ (r4 >> 5)) + r4));
    }
    output[2*i] = r3;
    output[2*i+1] = r4;
}
```

5.12 Second thread d'exécution

Un deuxième thread est présent. L'application utilise une technique d'obfuscation de ses imports couramment utilisé par les logiciels malveillants. L'idée est d'aller récupérer l'adresse de la librairie dynamique qui contient l'import désiré en passant par le PEB. Le programme calcul alors un condensat sur toutes les entrées disponibles jusqu'à trouver l'adresse de la librairie ciblée. Une fois cette librairie trouvée, le programme calcul cette fois le condensat de toutes les entrées exportées, jusqu'à tomber sur la fonction voulue, dans notre cas `CreateThread`. Ces opérations sont effectuées par la fonction située à l'adresse 0x4013E0. Le point d'entrée du thread étant en 0x401370 :



L'adresse de la fonction est obfusquée, elle se retrouve par le calcul : $0x7C033A - 0x3C0301 + 0x1337 = 0x401370$. Sinon on peut aussi facilement retrouver le thread sous `gdb` :

```
Attaching to program: /usr/bin/wine-preloader, process 8619
[New LWP 8650]

(gdb) thread apply all bt

Thread 2 (LWP 8650):
#0  0xf75145eb in ?? ()
#1  0x7ec700bf in ?? ()
#2  0x7ec70135 in ?? ()
#3  0x004013d6 in ?? ()
#4  0x7efa2ffd in ?? ()
#5  0x7efa002e in ?? ()
#6  0x7efa8b2d in ?? ()
#7  0xf75c26f2 in ?? ()
#8  0xf751de2e in ?? ()

Thread 1 (LWP 8619):
#0  0x0040163a in ?? ()
```

Le thread attend que la variable globale qui stocke la valeur de la ligne courante de l'image passe à -1, en vérfiant toutes les 100ms. L'instruction de fin de la première VM passe justement cette variable globale à -1 avant d'appeler la fonction `Sleep` pour 3 secondes. Lorsque cette variable, qui correspond au `program counter` de la VM2, passe à -1, le thread commence par effectuer une opération de ou-exclusif par mots de 32 bits à partir de l'adresse 0x414100, jusqu'à l'adresse 0x414240, avec la valeur 0xF4B02F4B.

5.13 Troisième LUM

Le résultat du décodage des données à l'adresse 0x414100 est direct et permet d'obtenir le dernier LUM du challenge :

```
$ ./dec_414100 | strings
LUM{+zhVQqJy03q}
```

On peut alors valider le LUM suivant : `LUM{+zhVQqJy03q}`.

5.14 Deuxième ROP-chain

Une fois les données décodées par le thread, la fonction située à l'adresse 0x401360 est appelée :

```
401360     push        ebp
401361     mov         ebp, esp
401363     mov         esp, [ebp+8]
401366     ret
```

On se retrouve de nouveau avec une chaîne ROP. Cependant, cette chaîne-ci contient une subtilité : la seconde adresse de retour sur la pile, 0x00401c8d, pointe vers une instruction `lret`. Ce reliquat du mode 16 bits d'intel permet de modifier le sélecteur de code segment en lui assignant la deuxième valeur présente sur la pile.
Sur les processeurs AMD64, le bit de poids faible du quartet de poids fort sélectionne le mode 32/64 bits, assigner une valeur de 0x33 au sélecteur `cs` contre une valeur précédente à 0x23 signifie que l'on passe en mode 64 bits (le tout en ring 3).

Le reste de la chaîne ROP est donc en 64 bits. À la fin de la chaîne on retrouve le même motif, avec une adresse de retour située en 0x0000000000401c8d, et la valeur 0x23 sur la pile. De cette manière, le thread repasse en 32 bits avant de finalement revenir dans le corps du thread.

En utilisant le même principe que pour la première chaîne, on peut extraire les différentes instructions exécutées en se basant sur les adresses de retour, et passer le tout à `objdump`. On obtient alors le code assembleur suivant :

```
pop    rdx                  ; rdx = 0x56687A2B7B4D554C;
pop    rcx                  ; rcx = 0x7D713330794A7151;

pop    rdx                  ; rdx = 0x756AF83DE1FFE263;
pop    rcx                  ; rcx = 0xFA34AE1002547B89;
not    rdx                  ; rdx = ~rdx;
bswap  rcx                  ; rcx = BSWAP64(rcx);
pop    rax                  ; rax = 0x00000000004140C0;

mov    eax,DWORD PTR [rax]  ; array = *rax;
mov    r8,QWORD PTR [rax]
xor    r8,rcx
add    r8,rdx
mov    QWORD PTR [rax],r8   ; array[0] = (array[0] ^ rcx) + rdx;

add    rax,0x8
mov    r8,QWORD PTR [rax]
xor    r8,rdx
bswap  r8
sub    r8,rcx
mov    QWORD PTR [rax],r8   ; array[1] = BSWAP64(array[1] ^ rdx) - rdx;

add    rax,0x8
mov    r8,QWORD PTR [rax]
bswap  r8
rol    rcx,0xd              ; rcx = ROL64(rcx, 0xd);
ror    r8,cl
mov    QWORD PTR [rax],r8   ; array[2] = ROR64(BSWAP64(array[2]), rcx);

add    rax,0x8
mov    r8,QWORD PTR [rax]
xor    r8,rcx
ror    rdx,0x15             ; rdx = ROR64(rdx, 0x15);
xchg   rcx,rdx
ror    r8,cl
add    r8,rdx
mov    QWORD PTR [rax],r8   ; array[3] = ROR64(array[3] ^ rcx, rdx) + rcx;
```

L'adresse 0x4140C0 correspond au pointeur vers l'adresse contenant le mot de passe entré, modifié par la seconde machine virtuelle. C'est l'étape finale avant la vérification par rapport aux données identifiées au début de l'analyse. Le code s'inverse facilement, et on obtient le code C suivant :

```
uint64_t rdx, rcx;
uint64_t *ptr64;

// RE @401360
ptr64 = (uint64_t *)input;
rdx = ~0x756AF83DE1FFE263;
rcx = __builtin_bswap64(0xFA34AE1002547B89);

ptr64[0] = (ptr64[0] - rdx) ^ rcx;
ptr64[1]  = __builtin_bswap64(ptr64[1] + rcx) ^ rdx;
rcx = ROL(rcx, 0xd, 64);
ptr64[2]  = __builtin_bswap64(ROL(ptr64[2], rcx&0xff, 64));
rdx = ROR(rdx, 0x15, 64);
ptr64[3]  = ROL(ptr64[3] - rcx, rdx&0xff, 64) ^ rcx;
```

5.15 Clé de l'épreuve

Après avoir mis bout à bout tous les morceaux de code développés pour inverser le calcul de vérification de la clé, on obtient un résultat en ASCII, ce qui est plutôt une bonne nouvelle puisque la boite d'édition de l'application ne supporte que le CP1252.

```
$ ./sstic17
CONST1 fde7f446
LUT4[0] 9aacc69b
LUT4[1] 89a31d8a
ROP chain: 00401a0d 00401d15 e9a356c1 00401c7b 00401c85 00401c8a 00401d15 79fa2185 00404bd7
LUT4[2]: a5b2cd1c
LUT4[3] b0a5ce54
CONST2 d2d413b3
EXPECTED:
25 67 81 29 75 6b e4 d4 e1 7c 9f 34 26 e8 1d 33 8e 64 e3 c4 66 8d 65 7a 20 e8 63 e7 bf 1a dd e6
EXPECTED before ROP 64 bits:
73 7d 2f 1b b1 37 34 c3 20 84 3c 36 87 4a 24 51 b2 47 3d 32 c6 b3 62 71 eb 71 35 1e 49 3e 71 fc
INPUT:
33 66 36 39 31 66 33 64 36 65 62 36 30 62 33 34 33 63 39 33 31 63 32 32 65 30 62 61 61 39 32 66
KEY is: 3f691f3d6eb60b343c931c22e0baa92f
```

On peut alors essayer la clé dans l'application :

Finalement, on peut valider la clé `3f691f3d6eb60b343c931c22e0baa92f` dans la VM du jeu et passer à la dernière épreuve.

6 LabyQRinth

On obtient le fichier `final.txt`. On pourrait espérer prendre un repos bien mérité, mais en fait non ...

Le fichier contient :

```
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 peut reconnaître un QR code dont les cases noires sont remplacées par des chiffres hexadécimaux. Après conversion du QR code vers une image type `bitmap`, on peut le décoder avec une des nombreuses applications existantes pour ça :

```
Please use this Nibble ADD key: 5571C2017
```

L'épreuve s'appelle « LabyQRinth », le défi consiste en fait à trouver le chemin dans le labyrinthe défini par les parenthèses du deQRypt() :

On obtient alors les données à déchiffrer :

```
ace80e6fcca1776558babe2c51d6b657658a8abcf973d1bb5c938ac9da252fd4c973
```

La clé étant une « Nibble ADD key », il faut alors faire une soustraction, quartet par quartet, des données à déchiffrer avec la clé. On peut écrire le code pour déchiffrer les données :

```c
int main()
{
    char a[] = "ace80e6fcca1776558babe2c51d6b657658a8abcf973d1bb5c938ac9da252fd4c973";
    char k[] = "5571C2017";
    int i, j;
    unsigned char o = 0;

    for (i = 0; i < sizeof(a); i+=sizeof(k)-1) {
        for (j = 0; j < sizeof(k)-1; j++) {
            char c[2];
            long d, e;
            c[1] = 0;
            if (i+j>=sizeof(a)) break;
            c[0] = a[i+j];
            d = strtol(c, NULL, 16);
            c[0] = k[j];
            e = strtol(c, NULL, 16);
            o = (o << 4) | ((d >= e) ? d - e: 16 + d - e);
            if ((i+j)%2) {
                printf("%c", o);
                o = 0;
            }
        }
    }
    printf("\n");
}
```

Pour finalement obtenir :

```
$ ./fin
WwLnWZkEAeMjPaoKEs5K7rld@sstic.org
```

L'adresse email `WwLnWZkEAeMjPaoKEs5K7rld@sstic.org` permet de valider le challenge !

FIN.