Challenge SSTIC 2014 : solution

Yoann Allain
yoann.allain@free.fr
12 mai 2014

Résumé

Ce document présente une démarche possible pour résoudre le challenge SSTIC 2014. Comme toujours, la validation du challenge nécessite de pouvoir extraire, depuis un fichier téléchargé sur le site de la conférence, une adresse email de la forme @sstic.org.

Trois étapes doivent être résolues de façon séquentielle pour terminer le challenge. L'étude d'une trace USB constitue la première étape. Cette capture contient un fichier envoyé d'un ordinateur sur un téléphone Android via le protocole ADB. Le but de cette étape est d'extraire ce fichier à analyser. Le fichier obtenu est un binaire ELF pour architecture ARMv8 64 bits. Cette architecture est assez récente et peu d'outils existent pour celle-ci. Dans cette deuxième étape, il faut donc obtenir un moyen d'émuler et d'analyser ce binaire inhabituel. L'émulation de ce programme nous demande l'entrée d'un mot de passe qui sert visiblement à déchiffrer un fichier. L'analyse permet de comprendre que ce binaire embarque un émulateur de machine virtuelle qui effectue le déchiffrement d'un fichier embarqué. Dans un premier temps, le but est ici d'obtenir le code de la machine virtuelle et de le désassembler. La fin de cette étape consiste à analyser le code exécuté par cette machine virtuelle afin de mettre en œuvre une attaque à clair connu pour déterminer le mot de passe protégeant le fichier embarqué. Celui-ci est en fait une archive qui contient un fichier de téléchargement d'un micrologiciel vers une machine inconnue distante et le fichier binaire d'un micrologiciel exemple pour cette machine. Le dernière étape consiste donc à l'analyse en boîte noire d'une machine distante pour laquelle il faudra déterminer le jeu d'instructions et l'architecture afin d'obtenir le code du système d'exploitation de la machine. Par l'analyse de ce dernier, on pourra finalement effectuer une élévation de privilèges afin d'afficher le contenu d'une zone mémoire initialement protégée. Cette zone mémoire contient entre autres, l'objectif final du challenge, c'est à dire l'adresse mail demandée.

Contenu

Etude de la trace USB

Récupération et premiers pas

Le fichier usbtrace.xz peut être téléchargé à l'adresse http://static.sstic.org/challenge2014/usbtrace.xz

```
$ wget --quiet http://static.sstic.org/challenge2014/usbtrace.xz
$ rm ../usbtrace.xz
$ ll
total 104
drwxrwxr-x  2 user user  4096 May 12 02:24 ./
drwxr-xr-x 25 user user  4096 May 12 02:25 ../
-rw-rw-r--  1 user user 97192 Apr 17 04:41 usbtrace.xz
$ md5sum usbtrace.xz
3783cd32d09bda669c189f3f874794bf  usbtrace.xz
$ file usbtrace.xz
usbtrace.xz: XZ compressed data
$ unxz usbtrace.xz
$ file usbtrace
usbtrace: UTF-8 Unicode text, with very long lines
$
```

Le fichier est donc lisible comme un fichier texte. Il contient notamment les lignes suivantes :

```
$ head usbtrace
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 ?

--
$
```

puis un très grand nombre de lignes de la forme suivante:

```
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
ffff8800d0fbf180 1765815053 S Bi:2:008:5 -115 211 <
ffff8800d0fbf180 1765815140 C Bi:2:008:5 0 211 = 7569643d 32303030 28736865
6c6c2920 6769643d 32303030 28736865 6c6c2920 67726f75 70733d31 30303328
67726170 68696373 292c3130 30342869 6e707574 292c3130 3037286c 6f67292c
31303039 286d6f75 6e74292c 31303131 28616462 292c3130 31352873 64636172
645f7277 292c3130 32382873 64636172 645f7229 2c333030 31286e65 745f6274
5f61646d 696e292c 33303032 286e6574 5f627429 2c333030 3328696e 6574292c
33303036 286e6574 5f62775f 73746174 73292063 6f6e7465 78743d75 3a723a73
68656c6c 3a7330
ffff8800d0fbf180 1765815196 S Bi:2:008:5 -115 24 <
ffff8800d0fbf9c0 1765815271 S Bo:2:008:3 -115 24 = 4f4b4159 fd010000
fb000000 00000000 00000000 b0b4bea6
ffff8800d0fbf9c0 1765815339 C Bo:2:008:3 0 24 >
ffff8800d0fbf180 1765815757 C Bi:2:008:5 0 24 = 57525445 fb000000 fd010000
02000000 17000000 a8adabba
ffff8800d0fbf180 1765815804 S Bi:2:008:5 -115 2 <
ffff8800d0fbf180 1765815888 C Bi:2:008:5 0 2 = 0d0a
```

Chacune de ces lignes semble correspondre à un échange entre l'ordinateur et le téléphone :

```
[meta-données] [données utiles]
```

Parmi ces lignes on remarque assez rapidement la logique suivante:

```
[meta-données] [taille données utiles] = [données utiles en représentation
hexadécimale]
```

Les données utiles semblent être encodées en représentation hexadécimale. On va donc dans un premier temps essayer de filtrer les données utiles avec les commandes grep et cut et puis les convertir en données binaires à l'aide de Python:

```
$ grep " = " usbtrace| cut -f2 -d'=' > usbtrace.filtered.data
$

$ python
Python 2.7.4 (default, Apr 19 2013, 18:32:33)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import binascii
>>> f_out = open('usbtrace.filtered.data.bin', 'w')
>>> f_in = open('usbtrace.filtered.data', 'r')
>>> for line in f_in:
...     binaryform = binascii.unhexlify(''.join(line.split()))
...     f_out.write(binaryform)
...
>>> f_in.close()
>>> f_out.close()
>>> quit()
$
```

Ceci nous donne un fichier binaire contenant les données utiles de la trace USB. Un éditeur hexadécimal ou la commande strings nous permet d'examiner le contenu du fichier qui se révèle être intéressant:

```
shell:id
uid=2000(shell) gid=2000(shell)
groups=1003(graphics),1004(input),1007(log),1009(mount),1011(adb),1015(sdcar
d_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),3006(net_bw_
stats) context=u:r:shell:s0

shell:uname -a
Linux localhost 4.1.0-g4e972ee #1 SMP PREEMPT Mon Feb 24 21:16:40 PST 2015
armv8l GNU/Linux

/sdcard/Documents/
CSW-2014-Hacking-9.11_uncensored.pdf
NATO_Cosmic_Top_Secret.gpg
/data/local/tmp
/data/local/tmp/badbios.bin
shell:chmod 777 /data/local/tmp/badbios.bin
```

Ces quelques lignes laissent croire que l'échange a permis de récupérer des informations sur le téléphone Android: En effet on retrouve des éléments classiques d'identification d'un système Android comme la présence d'un noyau Linux pour processeur ARM en l'occurrence "armv8", la présence de groupes d'utilisateurs typiques d'un système Android comme : adb; sdcard_r; sdcard_rw.
Des noms de fichiers attirent l'attention comme par exemple CSW-2014-Hacking-9.11_uncensored.pdf qui éveille tout de suite ma curiosité: serait-on en train d'assister à la fuite d'un des documents les plus dangereux depuis les fuites de Snowden? Ce challenge semble de plus en plus excitant! ;)
De même badbios.bin nous rappelle l'immense buzz paranoïde que nous a fait subir Dragos sur Twitter pour notre Noël 2013! Connaîtra-t-on enfin la vérité?

Dans cette trace il y a aussi quelques mots qui apparaissent très fréquemment:

```
OKAY
DONE
QUIT
WRTE
DENT
SEND
LIST
CLSE
```

Après une petite recherche (http://blogs.kgsoft.co.uk/2013_03_15_prg.htm), on voit que ceux-ci correspondent aux types de paquets utilisées dans le protocole ADB (Android Debug Bridge). La trace est donc probablement une trace logcat, mais n'ayant pas connaissance d'outils manipulant ce type de trace (comme un dissecteur Wireshark par exemple), je décide de l'examiner avec mes yeux ! ;-)

Il en ressort que l'échange ADB débute par l'exécution de quelques commandes sur le téléphone (uname, id), puis le listing d'une partie du système de fichiers du téléphone. Un fichier semble ensuite être téléchargé sur le téléphone sous le nom /data/local/tmp/badbios.bin et enfin un commande (chmod) est utilisée pour positionner correctement les attributs du nouveau fichier.

Interception du colis

L'objectif après cette analyse va donc être de récupérer le fichier en question. Pour cela on repère la commande SEND dans la trace, et on s'intéresse aux lignes qui suivent:
On remarque de très longues lignes qui contiennent apparemment 4096 octets de données utiles. Celles-ci correspondent aux données qui composent le transfert du fichier, on va donc s'employer à les filtrer. Et comme le dernier bloc du transfert a une taille inférieure aux 4k maximum induit par la fragmentation, on le repère et on l'ajoute au reste.

```
$ grep "4096 = " usbtrace |wc -l
19
$ grep "4096 = " usbtrace | cut -f2 -d'=' > usbtrace.file
$ grep "233 = "  usbtrace | cut -f2 -d'=' >> usbtrace.file
$
```

L'envoi du fichier se compose de 19 blocs de 4096 octets et 1 bloc de 233 octets. Un tour de Python pour en récupérer une version binaire de l'envoi de celui-ci:

```
$ python
Python 2.7.4 (default, Apr 19 2013, 18:32:33)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import binascii
>>> f_in = open('usbtrace.file', 'r')
>>> f_out = open('usbtrace.file.bin' , 'w')
>>> for line in f_in:
...     binaryform = binascii.unhexlify(''.join(line.split()))
...     f_out.write(binaryform)
...
>>> f_in.close()
>>> f_out.close()
>>> quit()
$
```

On a donc un extrait de transfert de fichier via le protocole ADB. Cependant pour transférer des fichiers via la commande SEND, le protocole prévoit le découpage du fichier en blocs de MAX_DATA_SIZE octets préfixés par 'DATAnnnn' où nnnn représente la taille du bloc courant. On remarque que notre transfert contient apparemment deux blocs :

```
$ grep -c DATA usbtrace.file.bin
2
$ hexdump -C usbtrace.file.bin | grep -A1 DATA
00000020  31 44 41 54 41 00 00 01  00 7f 45 4c 46 02 01 01
|1DATA.....ELF...|
00000030  00 00 00 00 00 00 00 00  00 02 00 b7 00 01 00 00
|................|
--
00010020  31 af 55 11 9a e8 1c a5  8e 44 41 54 41 b0 30 00
|1.U......DATA.0.|
00010030  00 3b 1c 6a fd e2 e3 98  56 25 b5 99 02 a6 36 10
|.;.j....V%....6.|
$
```

On va donc extraire les données binaires du fichier à l'aide de notre bon vieux dd:

La première partie:

```
$ dd if=usbtrace.file.bin of=badbios.part1.bin skip=$((0x21+0x8))
count=$((0x10000)) bs=1
65536+0 records in
65536+0 records out
65536 bytes (66 kB) copied, 0.154509 s, 424 kB/s
$
```

et la seconde:

```
$ dd if=usbtrace.file.bin of=badbios.part2.bin skip=$((0x10029+0x8))
count=$((0x30B0)) bs=1
12464+0 records in
12464+0 records out
12464 bytes (12 kB) copied, 0.033319 s, 374 kB/s
$
```

Il ne nous reste plus qu'à assembler les deux morceaux et c'est ainsi que l'on découvre un fichier ELF pour architecture ARMv8:

```
$ cat badbios.part1.bin badbios.part2.bin > badbios.bin
$ md5sum badbios.bin
b6097e562cb80a20dfb67a4833b1988a  badbios.bin
$ file badbios.bin
badbios.bin: ELF 64-bit LSB executable, version 1 (SYSV), statically linked,
stripped
$ readelf -Shl badbios.bin
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's 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

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)

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  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 to Segment mapping:
  Segment Sections...
   00     .text .rodata
   01     .data
   02
$
```

L'analyse de ce fichier sera donc la prochaine étape.

Etude de badbios.bin

Rétroconception du petit « badboy »

Quand on récupère un binaire dans ce genre de challenge, on peut être tenté de l'exécuter. Mais par prudence, on va tout d'abord chercher à l'analyser. Cependant, on peut très vite se rendre compte que pour désassembler ce programme avec IDA, il faut récupérer la toute dernière version à savoir la 6.5 qui contient depuis décembre 2013 le support pour processeurs ARM64 c'est-à-dire l'architecture ARMv8 ou encore AArch64. Tout ceci n'est pas très fair-play pour les participants car la licence IDA Pro n'est pas gratuite! Utiliser IDA Pro 6.5, c'est tricher! ;)

Bref une autre solution pour désassembler badbios.bin est d'utiliser QEMU! La version 2 de QEMU dispose du support pour l'architecture ARMv8, on peut récupérer les sources et les compiler en configurant de la sorte (se référer à la page https://wiki.debian.org/Arm64Qemu):

```
user@debian:/tmp/qemu-2d03b49$ ./configure --target-list=aarch64-linux-
user,aarch64-softmmu --prefix=/home/user/sstic2014/aarch64
```

Ainsi on aura une version de qemu pour architectures ARM 64 bits qui permettra d'émuler le binaire badbios.bin.
De plus on peut aussi s'équiper un peu mieux et compiler les binutils pour disposer d'outils supplémentaires:

```
user@debian:/tmp/binutils$ ./configure --prefix=/home/user/sstic2014/aarch64
--target=aarch64-linux-gnu --host=x86_64-linux-gnu --build=x86_64-linux-gnu
```

On peut ensuite exécuter le programme badbios.bin :

```
$ ./bin/qemu-aarch64 ../badbios.bin
:: Please enter the decryption key: azerty
   Wrong key format.
$ ./bin/qemu-aarch64 ../badbios.bin
:: Please enter the decryption key: 123456789ABCDEF
   Wrong key format.
$ ./bin/qemu-aarch64 ../badbios.bin
:: Please enter the decryption key: 1234567890ABCDEF
:: Trying to decrypt payload...
   Invalid padding.
$
```

On comprend alors que le programme demande un clé et cherche à déchiffrer des données. On peut facilement retrouver le format de cette clé qui se compose de 16 caractères hexadécimaux (en majuscule!). On peut ensuite utiliser la fonctionnalité strace de QEMU pour obtenir plus d'informations:

```
$ ./bin/qemu-aarch64 -strace ../badbios.bin
```

```
6255
mmap(0x0000000000400000,12288,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS
|MAP_FIXED,0,0) = 0x0000000000400000
6255 mprotect(0x0000000000400000,12288,PROT_EXEC|PROT_READ) = 0
6255
mmap(0x0000000000500000,69632,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS
|MAP_FIXED,0,0) = 0x0000000000500000
6255 mprotect(0x0000000000500000,69632,PROT_READ|PROT_WRITE) = 0
6255 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000801000
6255 mmap(NULL,65536,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000802000
6255 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000812000
6255 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000813000
6255 write(1,0x813000,36):: Please enter the decryption key:  = 36
6255 munmap(0x0000004000813000,36) = 0
6255 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000814000
6255 read(0,0x814000,16)1234567890ABCDEF
 = 16
6255 munmap(0x0000004000814000,16) = 0
6255 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000815000
6255 write(1,0x815000,32):: Trying to decrypt payload...
 = 32
6255 munmap(0x0000004000815000,32) = 0
6255 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000816000
6255 write(2,0x816000,20)   Invalid padding.
 = 20
6255 munmap(0x0000004000816000,20) = 0
6255 exit_group(0)
$
```

Ainsi on remarque la création de plusieurs mappings mémoire, avec des attributs en exécution pour certains. On va donc relancer l'exécution et tenter de dumper ces zones mémoires qui ont sûrement un intérêt. Voici un exemple où l'on suspend l'exécution du programme pour récupérer avec dd la zone mémoire présente en 0x40000.

```
$ ./bin/qemu-aarch64 -strace ../badbios.bin
6271
mmap(0x0000000000400000,12288,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS
|MAP_FIXED,0,0) = 0x0000000000400000
6271 mprotect(0x0000000000400000,12288,PROT_EXEC|PROT_READ) = 0
6271
mmap(0x0000000000500000,69632,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS
|MAP_FIXED,0,0) = 0x0000000000500000
6271 mprotect(0x0000000000500000,69632,PROT_READ|PROT_WRITE) = 0
6271 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000801000
6271 mmap(NULL,65536,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000802000
6271 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000812000
6271 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000813000
```

```
6271 write(1,0x813000,36):: Please enter the decryption key:  = 36
6271 munmap(0x0000004000813000,36) = 0
6271 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000814000
6271 read(0,0x814000,16)^Z
[1]+  Stoppé                 ./bin/qemu-aarch64 -strace ../badbios.bin
$ dd if=/proc/$(pidof qemu-aarch64)/mem bs=1 skip=$((0x0000000000400000))
count=12288 of=badbios_mmap_0x400000.bin
dd: "/proc/6271/mem": ne peut pas ignorer jusqu'au décalage indiqué
12288+0 enregistrements lus
12288+0 enregistrements Écrits
12288 octets (12 kB) copiÉs, 0,030685 s, 400 kB/s
$
```

En procédant de la même façon pour les autres zones avec les attributs en exécution, on peut espérer récupérer la totalité du code exécuté si celui-ci n'est pas modifié ultérieurement.

Une troisième fonctionnalité de QEMU est de pouvoir produire une trace complète d'exécution avec l'option "-d exec,in_asm" qui produit un énorme fichier contenant la liste des blocs de translations (TB) rencontrés par QEMU lors de l'exécution ainsi que le listing assembleur de chaque nouveau bloc d'instructions exécuté :

```
$ ./bin/qemu-aarch64 -d exec,in_asm ../badbios.bin
```

Sur la trace produite il est ensuite possible de filtrer les données de la trace d'exécution pour produire un graphe du binaire badbios.bin

Faute de temps pour expliquer un peu mieux les étapes suivantes je vais résumer les étapes qui m'ont permis de progresser dans cette partie du challenge:

1. Après analyse du code exécuté, on conclut que le programme badbios émule en fait une machine virtuelle:
   - Cette machine virtuelle déchiffre son code ou ses données par bloc de 64 octets juste avant utilisation et les rechiffre juste après utilisation.

2. A l'aide de aarch-linux-gnu-gdb issu de la compilation des binutils, on va pouvoir forcer le déchiffrement complet des données de la machine virtuelle émulée. Au lieu de déchiffrer 64 octets de données, on forcera le déchiffrement de la zone de mapping située à l'adresse 0x0000004000802000. On y retrouve entre autres les chaînes affichées lors de l'exécution du programme. Voici la composition de cette zone de données :

Offset	***Taille***	***Type de données***
0h	*40h*	*Registres de la machine virtuelle*
40h	*2C0h*	*Code du programme*
300h	*100h*	*Données du programme*
8000h	*2000h*	*Données chiffrées*

3. Considérant l'hypothèse d'être face à une machine virtuelle, on va donc rechercher la fonction d'interprétation des instructions de cette machine. On va ensuite déterminer les fonctions de base suivantes :

- GetPC
- SetRegisterValue
- GetRegisterValue
- ReadMemory
- WriteMemory

Puis on va retrouver la table des instructions de la machine virtuelle qui contient les 31 fonctions utilisées par le programme badbios pour émuler son code. Ces fonctions utilisent les fonctions de base précédemment identifiées. Par exemple l'instruction d'addition de deux registres utilise globalement l'algorithme suivant :

```
instruction_add_register(registre_dest, registre_src_1, registre_src_2)
{
      val_1 = GetRegisterValue(registre_src_1)
      val_2 = GetRegisterValue(registre_src_2)
      SetRegisterValue(resgistre_dest, val_1+val_2)
}
```

On peut donc assez facilement retrouver la sémantique des 31 instructions, les plus difficiles à analyser étant celles du branchement conditionnel et aussi la dernière instruction qui compte le nombre de bits à 1 et renvoie un booléen suivant la parité du nombre de bit à 1 trouvés dans le registre.

4. La dernière étape consiste enfin à écrire un désassembleur qui nous permet d'obtenir le listing suivant que l'on pourra ensuite analyser.

```
40 : 00 00 01 00 addis      r01, 0
44 : 00 00 21 01 addi       r01, 2
48 : 00 00 02 00 addis      r02, 0
4c : 00 00 12 01 addi       r02, 1
50 : 00 00 03 00 addis      r03, 0
54 : 00 32 E3 01 addi       r03, 32e
58 : 00 00 04 00 addis      r04, 0
5c : 00 02 44 01 addi       r04, 24
60 : 00 1D       syscall ; write ":: Please enter the decryption key::"
62 : 00 00 01 00 addis      r01, 0
66 : 00 00 11 01 addi       r01, 1
6a : 22 0A       xor        r02, r02
6c : 00 00 03 00 addis      r03, 0
70 : 00 3F C3 01 addi       r03, 3fc                ; asciiKeyBuf
74 : 00 00 04 00 addis      r04, 0
78 : 00 01 04 01 addi       r04, 10
7c : 00 1D       syscall                            ; read key 16 bytes
7e : 00 00 05 02 mov        r05, r01
82 : 00 00 03 00 addis      r03, 0
86 : 00 01 03 01 addi       r03, 10
8a : 35 13       sub        r05, r03
8c : 02 B4 6A 08 BNZ        2b4  r05, [F3, S0]             ; wrongkeyfmt
90 : 00 00 0F 00 addis      r15, 0
94 : 00 01 0F 01 addi       r15, 10
98 : 00 00 0E 00 addis      r14, 0
9c : 00 3F CE 01 addi       r14, 3fc                ; asciiKeyBuf
a0 : 00 00 0D 00 addis      r13, 0
a4 : 00 32 6D 01 addi       r13, 326                ; keyBuf
a8 : 0D 17       dec        r13
aa : 00 00 02 00 addis      r02, 0
ae : 00 03 02 01 addi       r02, 30
```

```
      b2 : 00 00 03 00 addis      r03, 0
      b6 : 00 03 93 01 addi       r03, 39
      ba : 00 00 04 00 addis      r04, 0
      be : 00 04 14 01 addi       r04, 41
      c2 : 00 00 05 00 addis      r05, 0
      c6 : 00 04 65 01 addi       r05, 46

loop_ca:
      ca : 00 00 EC 04 ld         r12, [r14 + 0]
      ce : 00 2C 01 02 mov        r01, r12
      d2 : 21 13       sub        r01, r02
      d4 : 02 B4 82 08 BNeg       2b4  r01, [F4, S0]             ; wrongkeyfmt
      d8 : 00 2C 01 02 mov        r01, r12
      dc : 31 13       sub        r01, r03
      de : 01 06 C2 08 BGTZ       106  r01, [F6, S0]             ; label_106

      e2 : 00 2C 01 02 mov        r01, r12
      e6 : 41 13       sub        r01, r04
      e8 : 02 B4 82 08 BNeg       2b4  r01, [F4, S0]             ; wrongkeyfmt
      ec : 00 2C 01 02 mov        r01, r12
      f0 : 51 13       sub        r01, r05
      f2 : 02 B4 A2 08 BsGTZ      2b4  r01, [F5, S0]             ; wrongkeyfmt
      f6 : 4C 13       sub        r12, r04
      f8 : 00 00 01 00 addis      r01, 0
      fc : 00 00 A1 01 addi       r01, a
     100 : 1C 12       add        r12, r01
     102 : 01 08 00 08 Branch     108 ; label_108

label_106:
     106 : 2C 13       sub        r12, r02

label_108:
     108 : 00 00 07 00 addis      r07, 0
     10c : 00 01 07 01 addi       r07, 10
     110 : F7 13       sub        r07, r15
     112 : 00 00 01 00 addis      r01, 0
     116 : 00 00 11 01 addi       r01, 1
     11a : 71 0C       and        r01, r07
     11c : 01 2C 62 08 BNZ        12c  r01, [F3, S0]             ; label_12c
     120 : 00 00 07 00 addis      r07, 0
     124 : 00 00 47 01 addi       r07, 4
     128 : 7C 0D       shl        r12, r07
     12a : 0D 16       inc        r13

label_12c:
     12c : 00 00 D1 04 ld         r01, [r13 + 0]
     130 : C1 0B       or         r01, r12
     132 : 00 00 D1 07 sb         r01, [r13 + 0]
     136 : 0E 16       inc        r14
     138 : 0F 17       dec        r15
     13a : 00 CA 7E 08 BNZ        ca  r15, [F3, S0]         ; loop_ca ------^

     13e : 00 00 01 00 addis      r01, 0
     142 : 00 00 21 01 addi       r01, 2
     146 : 00 00 02 00 addis      r02, 0
     14a : 00 00 12 01 addi       r02, 1
     14e : 00 00 03 00 addis      r03, 0
     152 : 00 35 43 01 addi       r03, 354
     156 : 00 00 04 00 addis      r04, 0
     15a : 00 02 04 01 addi       r04, 20
     15e : 00 1D       syscall   ; write ":: Trying to decrypt payload...\n"

init:
     160 : 00 00 01 00 addis      r01, 0
     164 : 00 32 61 01 addi       r01, 326 ; keyBuf
     168 : 00 00 1A 02 mov        r10, [r01 + 0]           ; keyBuf.0
     16c : 00 04 1B 02 mov        r11, [r01 + 4]           ; keyBuf.1
     170 : 11 0A       xor        r01, r01                 ; r01 = 0
```

```
     172 : 00 00 02 00 addis      r02, 0
     176 : 08 00 02 01 addi       r02, 8000               ; r02 = 8000
     17a : 00 00 03 00 addis      r03, 0
     17e : 00 00 83 01 addi       r03, 8                  ; r03 = 8
     182 : 44 0A       xor        r04, r04                ; r04 = 0
     184 : 0B 00 0C 00 addis      r12, b000
     188 : 00 00 0C 01 addi       r12, 0                  ; r12 = b0000000
     18c : 00 00 0D 00 addis      r13, 0
     190 : 00 00 1D 01 addi       r13, 1                  ; r13 = 1

loop_194:
     194 : 00 24 08 02 mov        r08, r10                ; keyBuf.0
     198 : 00 28 09 02 mov        r09, r11                ; keyBuf.1
     19c : C8 0C       and        r08, r12      ; r08 = b0000000 & keyBuf.0
     19e : D9 0C       and        r09, r13      ; r09 = keyBuf.1 & 1
     1a0 : 98 0A       xor        r08, r09
     1a2 : 89 1E       TBitCnt    r09, r08
     1a4 : 00 00 08 00 addis      r08, 0
     1a8 : 00 00 18 01 addi       r08, 1                  ; r08 = 1
     1ac : 00 00 07 00 addis      r07, 0
     1b0 : 00 01 F7 01 addi       r07, 1f                 ; r07 = 1f
     1b4 : 00 24 06 02 mov        r06, r10
     1b8 : 86 0C       and        r06, r08
     1ba : 76 0D       shl        r06, r07    ; r06 = ((0x24 & 1) << 0x1f)
     1bc : 8B 0E       shr        r11, r08                ; keyBuf.1
     1be : 6B 0B       or         r11, r06                ; keyBuf.1
     1c0 : 8A 0E       shr        r10, r08                ; keyBuf.0
     1c2 : 79 0D       shl        r09, r07
     1c4 : 9A 0B       or         r10, r09                ; keyBuf.0
     1c6 : 03 17       dec        r03              ; r03 = 8 7 6 5 4 3 2 1
     1c8 : 00 28 07 02 mov        r07, r11
     1cc : 87 0C       and        r07, r08
     1ce : 37 0D       shl        r07, r03
     1d0 : 74 0B       or         r04, r07
     1d2 : 01 F6 66 08 BNZ        1f6  r03, [F3, S0]              ; label_1f6

     1d6 : 00 00 07 00 addis      r07, 0
     1da : 08 00 07 01 addi       r07, 8000
     1de : 17 12       add        r07, r01
     1e0 : 00 00 78 04 ld         r08, [r07 + 0]     ; load 8000 + r01
     1e4 : 48 0A       xor        r08, r04
     1e6 : 00 00 78 07 sb         r08, [r07 + 0]     ; store 8000 + r01
     1ea : 00 00 03 00 addis      r03, 0
     1ee : 00 00 83 01 addi       r03, 8
     1f2 : 01 16       inc        r01
     1f4 : 44 0A       xor        r04, r04

label_1f6:
     1f6 : 00 00 08 00 addis      r08, 0
     1fa : 02 00 08 01 addi       r08, 2000          ; r08 = 2000
     1fe : 18 13       sub        r08, r01
     200 : 01 94 B0 08 BsGTZ      194  r08, [F5, S0]       ; loop_194 --------^

     204 : 00 00 0D 00 addis      r13, 0
     208 : 08 00 0D 01 addi       r13, 8000
     20c : 00 00 0C 00 addis      r12, 0
     210 : 02 00 0C 01 addi       r12, 2000
     214 : 00 00 0B 00 addis      r11, 0
     218 : 00 08 0B 01 addi       r11, 80
     21c : AA 0A       xor        r10, r10
     21e : 00 00 09 00 addis      r09, 0
     222 : 00 00 89 01 addi       r09, 8

loop_226:
     226 : 0A 16       inc        r10
     228 : 0C 17       dec        r12
     22a : 02 DA D8 08 BGTZ       2da  r12, [F6, S0]              ; invalidpad
     22e : 00 30 0A 02 mov        r10, r13
```

```
     232 : CA 12       add        r10, r12
     234 : 00 00 A1 04 ld         r01, [r10 + 0]
     238 : 02 26 42 08 BZ         226  r01, [F2, S0]       ; loop_22 ----------^

     23c : B1 13       sub        r01, r11
     23e : 02 DA 62 08 BNZ        2da  r01, [F3, S0]             ; invalidpad
     242 : 9A 13       sub        r10, r09
     244 : 02 DA D4 08 BGTZ       2da  r10, [F6, S0]             ; invalidpad

good_end:
     248 : 01 0C       and        r01, r00
     24a : 00 00 02 00 addis      r02, 0
     24e : 00 3F 02 01 addi       r02, 3f0          ; offset payload.bin
     252 : 00 00 03 00 addis      r03, 0
     256 : 00 24 13 01 addi       r03, 241
     25a : 00 00 04 00 addis      r04, 0
     25e : 00 1B 64 01 addi       r04, 1b6
     262 : 00 1D       syscall         ; syscall openat(-0x64, payload.bin, …)
     264 : 02 B2 82 08 BNeg       2b2  r01, [F4, S0]             ; haltvm
     268 : 00 00 02 02 mov        r02, r01          ; r02 contains fd
     26c : 00 00 01 00 addis      r01, 0
     270 : 00 00 21 01 addi       r01, 2
     274 : 00 00 03 00 addis      r03, 0
     278 : 08 00 03 01 addi       r03, 8000
     27c : 00 2C 04 02 mov        r04, r12
     280 : 00 1D       syscall         ; write 2c bytes of buffer 8000 to fd
     282 : 00 00 01 00 addis      r01, 0
     286 : 00 00 31 01 addi       r01, 3
     28a : 00 1D       syscall                       ; syscall 3 close fd
     28c : 00 00 01 00 addis      r01, 0
     290 : 00 00 21 01 addi       r01, 2
     294 : 00 00 02 00 addis      r02, 0
     298 : 00 00 12 01 addi       r02, 1
     29c : 00 00 03 00 addis      r03, 0
     2a0 : 00 3C 23 01 addi       r03, 3c2
     2a4 : 00 00 04 00 addis      r04, 0
     2a8 : 00 02 D4 01 addi       r04, 2d
     2ac : 00 1D       syscall   ; write ":: Decrypted payload written to
payload.bin.\n"
     2ae : 02 B2 00 08 Branch     2b2 ; haltvm

haltvm:
     2b2 : 00 1C       hlt

wrongkeyfmt:
     2b4 : 00 00 01 00 addis      r01, 0
     2b8 : 00 00 21 01 addi       r01, 2
     2bc : 00 00 02 00 addis      r02, 0
     2c0 : 00 00 22 01 addi       r02, 2
     2c4 : 00 00 03 00 addis      r03, 0
     2c8 : 00 37 43 01 addi       r03, 374
     2cc : 00 00 04 00 addis      r04, 0
     2d0 : 00 01 54 01 addi       r04, 15
     2d4 : 00 1D       syscall         ; write "   Wrong key format.\n"
     2d6 : 02 B2 00 08 Branch     2b2 ; haltvm

invalidpad:
     2da : 00 00 01 00 addis      r01, 0
     2de : 00 00 21 01 addi       r01, 2
     2e2 : 00 00 02 00 addis      r02, 0
     2e6 : 00 00 22 01 addi       r02, 2
     2ea : 00 00 03 00 addis      r03, 0
     2ee : 00 38 A3 01 addi       r03, 38a
     2f2 : 00 00 04 00 addis      r04, 0
     2f6 : 00 01 44 01 addi       r04, 14
     2fa : 00 1D       syscall         ; write "   Invalid padding.\n"
     2fc : 02 B2 00 08 Branch     2b2 ; haltvm
```

On peut donc remarquer que le programme tente un déchiffrement d'un fichier payload.bin. Dans la suite de cette étape le défi sera donc d’exploiter la faiblesse de la cryptographie employée afin d'obtenir le fichier payload.bin.

Attaque cryptographique de l'algorithme utilisé par la VM

Après avoir désassemblé le code exécuté par la machine virtuelle, on peut donc distinguer les différentes phases du programme exécuté par celle-ci:
- Affichage de la chaîne ":: Please enter the decryption key::" et lecture de la clé entrée par l'utilisateur
- Vérification du format de la clé et transcodage de celle-ci sur 8 octets : le format de la clé est 16 caractères hexadécimaux en majuscule.
    - Affichage de la chaîne " Wrong key format.\n" et fin du programme en cas d'erreur sur le format de la clé.
- Affichage de la chaîne ":: Trying to decrypt payload...\n" et déchiffrement du fichier embarqué.
- Vérification du bourrage de fin pour le fichier déchiffré : présence du caractère 0x80 suivi de caractère(s) 0x0.
    - Affichage de la chaîne " Invalid padding.\n" et fin du programme en cas de non-conformité du fichier déchiffré.
- Création du fichier "payload.bin" avec le contenu du fichier déchiffré et affichage de la chaîne ":: Decrypted payload written to payload.bin.\n"
- Fin du programme.

L'analyse de la partie déchiffrement montre l'utilisation d'un chiffrement par flot:

Ainsi on peut constater que les bits de clé produits sont issus de la clé initiale de 64 bits. Par ailleurs 63 bits de la clé initiale et un bit généré à partir de la clé servent à chiffrer les 8 premiers octets (64 bits) du fichier. Connaissant les 8 premiers octets du chiffré, on peut donc effectuer une attaque à clair connu, en choisissant un type de fichier clair. Par exemple pour un fichier PDF, les 8 premiers octets sont connus, il s'agit généralement d'un commentaire contenant la chaîne "%PDF-1.4\n%" . Connaissant le clair et connaissant le chiffré, on peut donc déterminer la clé:
Exemple avec un fichier PDF:

```
/dev$ hexdump -C /usr/share/cups/data/default.pdf
00000000  25 50 44 46 2d 31 2e 35  0a 25 b5 ed ae fb 0a 33  |%PDF-
1.5.%.....3|
/dev$
```

```
Début du fichier PDF        25 50 44 46 2d 31 2e 35
Début du fichier Chiffré    00 bc 68 15 b5 6b 1b 41
Clé voulue                  25 ec 2c 53 98 5a 35 74
```

La clé voulue n'est pas directement la clé entrée par l'utilisateur : il y a un décalage à droite opéré sur la clé entrée. Sur les 64 bits de la clé entrée par l'utilisateur, après transcodage de celle-ci, 63 bits sont utilisés pour chiffrer le clair, donc pour un clair connu, il y aura deux clés possibles et donc un bit à deviner.
Après analyse de l'algorithme de transcodage de la clé, on peut écrire le script suivant pour calculer deux clés en fonction d'un clair connu:

```
/dev$ cat findkey.py
#!/usr/bin/env python
import sys,os
```

```
# ciphered file first bytes
cipher_hdr = ['\x00', '\xbc', '\x68', '\x15', '\xb5', '\x6b', '\x1b',
'\x41']

# function to reverse byte bits
def rev8bits(x):
      width = 8
      return sum(1<<(width-1-i) for i in range(width) if((x>>i) & 1))

def findkey(name):
      # check size
      f_size = os.path.getsize(name)
      if (f_size <= 8):
            return

      f = open(name, 'r')
      f_hdr = []
      for i in range(8):
            f_hdr.append(f.read(1))

      key = range(8)
      carry = 0
      for i in range(8):
            raw_key_byte = ord(f_hdr[i]) ^ ord(cipher_hdr[i])
            reversed_byte = rev8bits(raw_key_byte)
            key[(i+4)%8] = ((reversed_byte<<1) + carry) & 0xff
            carry = raw_key_byte & 1

      print "Found keys  :",
      print "%02X%02X%02X%02X%02X%02X%02X%02X" %
(key[0],key[1],key[2],key[3],key[4],key[5],key[6],key[7]),
      print "or",
      print "%02X%02X%02X%02X%02X%02X%02X%02X" %
(key[0],key[1],key[2],key[3],key[4]+1,key[5],key[6],key[7]),
      print "for file %s" % (name)

def usage():
      print "Usage :"
      print "%s <clearfile>" % (sys.argv[0])

if (len(sys.argv) < 2):
      usage()
      exit(-1)

findkey(sys.argv[1])
exit(0)

/dev$
```

Lorsque l'on exécute ce script sur les fichiers réguliers de notre système avec la commande :

```
/dev$ find / -type f -exec python findkey.py '{}' \; |
```

On remarque la présence des lignes suivantes pour les fichiers de type archive ZIP:

```
[...]
Found keys  : 0BADB10514DEAD11 or 0BADB10515DEAD11 for file
/media/secret_files/DragosPlotProofs.zip
```

```
Found keys  : FBADB10514DEAD11 or FBADB10515DEAD11 for file
/media/crypto/HowToCipherWithBase64andRuleTheWorld.zip
Found keys  : 0BAD310514DEAD11 or 0BAD310515DEAD11 for file
/media/badbios_source_code.zip
Found keys  : 0BAD910514DEAD11 or 0BAD910515DEAD11 for file
/media/intox/SourceSampleCodesToInsertInOpenSSL.zip
Found keys  : 0BAD912514DEAD11 or 0BAD912515DEAD11 for file
/media/Doc/HowtoWriteCCTPForDummies_AllVersions.zip
Found keys  : 5BADB02514DEAD11 or 5BADB02515DEAD11 for file
/media/WhyNoFuckingZipFileIsPresentOnMyFreshlyInstalledLinux.zip
[...]
```

Les mots de passe proposés pour les fichiers ZIP semblent vouloir proclamer la mort de BadBios, n'en déplaise à Dragos! Essayons la clé la plus explicite:

```
$ ./bin/bin/qemu-aarch64  ./badbios.bin
:: Please enter the decryption key: 0BADB10515DEAD11
:: Trying to decrypt payload...
:: Decrypted payload written to payload.bin.
$
```

Analyse en boîte noire de la machine distante

« Michael-Jacksonisation » de l'analyse en boîte noire

Le fichier payload.bin déchiffré à l'étape précédente est donc une archive ZIP.

L'archive contient deux petits fichiers: upload.py et fw.hex:

```
user@debian:~$ 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"

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()
```

et

```
user@debian:~$ cat fw.hex
:100000002100111F2001107CC0D220101000210132
:10001000117C2200120FC03C20101000210111B2EF
:1002000220012229C07620111000C0B4C0B65A00B8
:10030002100112420011 0B2C0BE51AAC10A210022
:10040001129200110B2C09421001109200110A82B
:10005000C08AB084580059115A2230002101110081
:10006000220012017310A006F0806002B3F6300087
:10007000510022001201230013FFE4806114940A4E
:10008000844A7404E49461145113E480E581F5809A
:10009000F48160027430AFE2D00FB002B0005800BB
:1000A00059115A22300051005200230013FF24003E
:1000B000140160245003E58061155113E580E68149
:1000C000F680F58165565553E585E6923665F692DC
:1000D000622475A2A7DCD00FC801B3FCC802D00F00
:1000E000C803D00F2100110122011200E301711198
:1000F000E40184424034D00F3222230013013444FF
:10010000E4025444A0087441A0066003B3F0300038
:10011000D00F241014002500150F2600160A270002
:1001200017019214522573 26A80623001337B00432
:10013000230013306232333 3F20360077247A006A4
:1001400063579443B3DCD00F242714102500150AFD
:10015000366627001701700760079214832471 1315
:10016000944528001820333 3F8034662A3EA280098
:1001700018306882F8035444A7DED00F59656168CF
:10018000526973634973476F6F6421004669726DEA
:1001900077617265207631 2E33332E372073746188
:1001A0007274696E672E0A0048616C74696E672EFE
:1001B0000A00942B506FAE0CBB1F39B4D8CA05FD92
:1001C0008A0F5AE8B5D40D6CE86AA6ACC492F8F16F
:0C01D00072A77CE6D5A5680921D4410087
:00000001FF
user@debian:~$
```

Le premier fichier permet de télécharger sur une machine distante, un microcontrôleur, le second fichier qui semble contenir un micrologiciel pour cette machine. Les commentaires du fichier Python nous fournissent quelques informations bien utiles sur le microcontrôleur: Le mapping mémoire est fourni:

```
#
#   == 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)              /
#
```

On distingue plusieurs informations utiles:
- Le microcontrôleur semble utiliser un adressage 16bits.
- Deux modes d'exécution (privilégié et non-privilégié) semblent exister sur la machine.
- Une zone du mapping se nomme "Secret memory area" et sera probablement notre objectif pour cette étape.

Cependant comme indiqué dans le fichier Python, aucune autre information n'est disponible sur cette machine distante. Le script consiste à envoyer le contenu d'un fichier fw.hex au microcontrôleur situé à l'adresse IP 178.33.105.197 sur le port TCP 10101 et afficher l'éventuelle réponse de la machine distante.

De son côté, fw.hex semble être un fichier au format Intel HEX, format de fichier assez répandu dans le monde de l'embarqué et du micro-logiciel. Le fichier est donc composé de lignes de caractères hexadécimaux, décrivant les données du firmware ainsi que l'adresse destination et un contrôle d'intégrité pour cette ligne de données.

Une fois l'aperçu de ces deux fichiers réalisé, il convient de les utiliser (avec Python en version 3) pour observer le déroulement du téléchargement:

```
user@debian:~$ python3 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.

user@debian:~$
```

On distingue la réponse du microcontrôleur:

```
"

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

"
```

Ces chaînes de caractères sont des indices importants pour la suite de l'analyse. En effet maintenant que l'on a une trace d'exécution de la machine, on va donc essayer de modifier le firmware afin d'influer sur la réponse de celle-ci.

Pour manipuler le firmware au format Intel HEX, il existe un librairie Python Intel HEX qui permet d'effectuer les transformations depuis le format binaire vers le format Intel HEX et vice-versa. Disponible à l'adresse https://pypi.python.org/pypi/IntelHex/1.1 elle s'installe très facilement mais surtout dispose d'un répertoire de scripts contenant

notamment les scripts hex2bin.py et bin2hex.py qui vont nous être très utiles.

Commençons par obtenir la version binaire du fichier firmware fw.hex :

```
user@debian:~/Downloads/intelhex-1.5/scripts$ python hex2bin.py
ERROR: Hex file is not specified
Hex2Bin convertor utility.
Usage:
    python hex2bin.py [options] INFILE [OUTFILE]

Arguments:
    INFILE      name of hex file for processing.
    OUTFILE     name of output file. If omitted then output
                will be writing to stdout.

Options:
    -h, --help              this help message.
    -v, --version           version info.
    -p, --pad=FF            pad byte for empty spaces (ascii hex value).
    -r, --range=START:END   specify address range for writing output
                            (ascii hex value).
                            Range can be in form 'START:' or ':END'.
    -l, --length=NNNN,
    -s, --size=NNNN         size of output (decimal value).

user@debian:~/Downloads/intelhex-1.5/scripts$ python hex2bin.py fw.hex fw.bin
user@debian:~/Downloads/intelhex-1.5/scripts$ hexdump -C fw.bin
00000000  21 00 11 1b 20 01 10 8c  c0 d2 20 10 10 00 21 01  |!... ..... ...!.|
00000010  11 7c 22 00 12 0f c0 3c  20 10 10 00 21 01 11 b2  |.|"....< ...!...|
00000020  22 00 12 29 c0 76 20 11  10 00 c0 b4 c0 b6 5a 00  |"..).v .......Z.|
00000030  21 00 11 24 20 01 10 b2  c0 be 51 aa c1 0a 21 00  |!..$ .....Q...!.|
00000040  11 29 20 01 10 b2 c0 94  21 00 11 09 20 01 10 a8  |.) .....!... ...|
00000050  c0 8a b0 84 58 00 59 11  5a 22 30 00 21 01 11 00  |....X.Y.Z"0.!...|
00000060  22 00 12 01 73 10 a0 06  f0 80 60 02 b3 f6 30 00  |"...s.....`...0.|
00000070  51 00 22 00 12 01 23 00  13 ff e4 80 61 14 94 0a  |Q."...#.....a...|
00000080  84 4a 74 04 e4 94 61 14  51 13 e4 80 e5 81 f5 80  |.Jt...a.Q.......|
00000090  f4 81 60 02 74 30 af e2  d0 0f b0 02 b0 00 58 00  |..`.t0........X.|
000000a0  59 11 5a 22 30 00 51 00  52 00 23 00 13 ff 24 00  |Y.Z"0.Q.R.#...$.|
000000b0  14 01 60 24 50 03 e5 80  61 15 51 13 e5 80 e6 81  |..`$P...a.Q.....|
000000c0  f6 80 f5 81 65 56 55 53  e5 85 e6 92 36 65 f6 92  |....eVUS....6e..|
000000d0  62 24 75 a2 a7 dc d0 0f  c8 01 b3 fc c8 02 d0 0f  |b$u.............|
000000e0  c8 03 d0 0f 21 00 11 01  22 01 12 00 e3 01 71 11  |....!..."......q.|
000000f0  e4 01 84 42 40 34 d0 0f  32 22 23 00 13 01 34 44  |...B@4..2"#...4D|
00000100  e4 02 54 44 a0 08 74 41  a0 06 60 03 b3 f0 30 00  |..TD..tA..`...0.|
00000110  d0 0f 24 10 14 00 25 00  15 0f 26 00 16 0a 27 00  |..$...%...&...'.|
00000120  17 01 92 14 52 25 73 26  a8 06 23 00 13 37 b0 04  |....R%s&..#..7..|
00000130  23 00 13 30 62 32 33 33  f2 03 60 07 72 47 a0 06  |#..0b233..`.rG..|
00000140  63 57 94 43 b3 dc d0 0f  24 27 14 10 25 00 15 0a  |cW.C....$'..%...|
00000150  36 66 27 00 17 01 70 07  60 07 92 14 83 24 71 13  |6f'...p.`....$q.|
00000160  94 45 28 00 18 20 33 33  f8 03 46 62 a3 ea 28 00  |.E(.. 33..Fb..(.|
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.|
000001b0  0a 00 94 2b 50 6f ae 0c  bb 1f 39 b4 d8 ca 05 fd  |...+Po....9.....|
000001c0  8a 0f 5a e8 b5 d4 0d 6c  e8 6a a6 ac c4 92 f8 f1  |..Z....l.j......|
000001d0  72 a7 7c e6 d5 a5 68 09  21 d4 41 00              |r.|...h.!.A.|
000001dc
user@debian:~/Downloads/intelhex-1.5/scripts$
```

L'affichage du firmware permet de retrouver une des chaînes présente dans la réponse:

```
"Firmware v1.33.7 starting."
```

```
Celle-ci se trouve à l'offset 0x18C dans le fichier binaire:

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.|
```

De plus on distingue la chaîne "RiscIsGood!", ce qui laisse à penser que le microcontrôleur utilise une architecture de type RISC (Reduced Instruction Set Computing), c'est-à-dire un microprocesseur utilisant un jeu d'instructions réduit et comme le dit Wikipedia: facile à décoder et comportant uniquement des instructions simples! Comme cela nous arrange bien, pour la suite de l'analyse, nous faisons l'hypothèse qu'il s'agit donc bien d'une architecture RISC.
Premièrement, essayons de modifier le comportement du firmware:
La chaîne située à l'offset 0x18C "Firmware v1.33.7 starting." est affichée dans la réponse. On peut supposer comme dans de nombreuses architectures, que l'adresse absolue de la chaîne à afficher est fournie comme paramétre d'une fonction (comme printf). Le firmware étant probablement téléchargé au début de la zone mémoire lui étant dédié, c'est-à-dire en 0x0000 d'après le mapping mémoire, l'adresse de la chaîne devant être utilisée par le fimware serait donc 0x018C.

Une recherche du pattern 0x18C (il convient d'effectuer la recherche en little-endian et en big-endian) ne donne aucun résultat sur le firmware fw.bin. Mais dans de nombreuses architectures RISC, la taille des adresses correspond aussi à la taille d'une instruction. Dans notre cas et d'après le mapping mémoire, les adresses semblent avoir une taille de 16bits, ce qui donnerait une taille d'instruction de 16bits. Or pour utiliser une adresse de 16 bits grâce à des instructions de 16 bits, il faut au moins deux instructions. C'est pourquoi dans les architectures RISC, il est très courant d'utiliser deux instructions pour charger une adresse:
- Une instruction pour charger la partie haute de l'adresse.
- Une instruction pour charger la partie basse de l'adresse.

Si l'on recherche la partie basse de l'adresse de la chaîne, on la retrouve très rapidement à l'offset 0x7 du fichier fw.bin au sein d'une instruction supposée de 16bits "108C"
Grâce à un éditeur hexadécimal comme ghex, remplaçons la valeur 8c par 7c afin d'essayer de faire afficher la chaîne "YeahRiscIsGood!" située à l'offset 0x17C.

```
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.|
```

Puis pour utiliser ce firmware modifié, il faut tout d'abord le convertir au format Intel HEX avec le script fourni dans la librairie Python précédemment utilisée et enfin utiliser le script de téléchargement fourni:

```
user@debian:~/Downloads/intelhex-1.5/scripts$ python bin2hex.py fw.bin
fw.hex
user@debian:~/Downloads/intelhex-1.5/scripts$
```

Le script utilisé convertit le format binaire vers le format Intel HEX et ajuste l'octet de la somme de contrôle pour la ligne modifiée.

```
user@debian:~/Downloads/intelhex-1.5/scripts$ python3 upload.py
```

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

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

System reset.
YeahRiscIsGood!Firmware v1Execution completed in  8339 CPU cycles.
Halting.

user@debian:~/Downloads/intelhex-1.5/scripts$
```

Bingo, le résultat est là: notre chaîne "YeahRiscIsGood!" est affichée, ce qui prouve que le firmware n'embarque pas de signature cryptographique et donc que l'on peut le modifier à souhait.
Maintenant que l'on a réussi à modifier l'affichage de la chaîne au sein des 8 premiers octets, ne gardons que ceux-là pour fabriquer un nouveau firmware et l'essayer :

```
user@debian:~/Downloads/intelhex-1.5/scripts$ hexdump -C shortfw.bin
00000000  21 00 11 1b 20 01 10 8c  00 00 00 00 00 00 00 00  |!...
..........|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
*
000001dc
user@debian:~/Downloads/intelhex-1.5/scripts$ python bin2hex.py shortfw.bin
fw.hex && python3 upload.py
-------------------------------------------
----- Microcontroller firmware uploader -----
-------------------------------------------

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

System reset.
-- Exception occurred at 0008: Invalid instruction.
   r0:018C     r1:001B    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:0008 fault_addr:0000 [S:0 Z:0] Mode:user
CLOSING: Invalid instruction.

user@debian:~/Downloads/intelhex-1.5/scripts$
```

Cette fois, la chaîne n'est pas affichée, ce qui signifie que les 8 premiers ne contiennent pas d'appel à la fonction d'affichage. Et surtout information très intéressante, la machine a stoppé son exécution sur une instruction invalide située à l'offset 8. Or à l'offset 8, nous avons placé la valeur 0. L'instruction 0 serait donc une instruction invalide. Celle-ci sera très importante pour la suite de notre analyse: en effet grâce à cette instruction nous sommes en mesure de contrôler la machine distante et la faire fonctionner comme un Oracle. Lorsque la machine s'arrête sur une erreur, elle envoie dans sa réponse de nombreuses informations utiles pour notre analyse :
- un état de ses registres généraux (r0 à r15)
- la valeur du registre pc (program counter)

- le mode d'exécution (user ou kernel)
- l'état des flags de comparaison (sign flag et zero flag)

La suite de l'analyse va donc consister à déterminer la taille de l'instruction. La taille des registres étant de 16 bits d'après le dump, nous faisons l'hypothèse que l'instruction fait aussi 16 bits car l'architecture utilisée est de type RISC. Dans notre essai précédents il y aurait donc 4 instructions : 21 00; 11 1b; 20 01 et 10 8c.
Vu l'état des registres après ces 4 instructions, on peut faire l'hypothèse suivante:

```
2 1 00 = movhi    r1, 0x0     ; mov immediate shifted
1 1 1b = movi     r1, 0x1b    ; mov immediate
2 0 01 = movhi    r0, 0x1     ; mov immediate shifted
1 0 8c = movi     r0, 0x8c    ; mov immediate
```

où une instruction est généralement composée ainsi de quatre nibbles:

```
[instruction][destination register] [immediate value]
```

Ou bien :

```
[instruction][destination register] [source 1 register][source 2 register]
```

Pour vérifier cette hypothèse, il suffit de modifier notre firmware précédent, afin de faire varier les valeurs immédiates et les registres destination. Puis on ré-envoie le firmware et grâce à la réponse de notre nouvel Oracle on peut confirmer ou infirmer nos hypothèses. On procéde par itération pour déterminer les autres instructions et ainsi on découvre un ensemble assez complet des instructions de notre machine:

Nibble Instruction	***Arguments Instruction***	***Description***
0 : invalid		
1 : movi	*Valeur immédiate*	*Charge la valeur dans l'octet de poids faible du registre destination*
2 : movhi	*Valeur immédiate*	*Charge la valeur dans l'octet de poids fort du registre destination*
3 : xor	*Registers numbers*	*)*
4 : or	*Registers numbers*	*) Effectue l'opération entre les*
5 : and	*Registers numbers*	*) deux registres sources et place*
6 : add	*Registers numbers*	*) le résultat dans le*
7 : sub	*Registers numbers*	*) registre destination*
8 : mul	*Registers numbers*	*)*
9 : div	*Registers numbers*	*)*
a : conditional branch	*Relative branch value*	*Effectue un saut conditionnel*
b : jump	*Relative jump value*	*Effectue un saut inconditonnel*
c : call	*Relative call value*	*Effectue un appel de fonction, et place l'adresse de retour dans le registre r15*
d : branch register	*Register to branch to*	*Effectue un saut à l'adresse contenue dans le registre*
e : load	*Registers numbers*	*Charge le registre destination avec l'octet lu à l'adresse somme des deux registres sources*
f : store	*Registers numbers*	*Charge l'adresse somme des deux registres sources avec la valeur de l'octet de poids faible du registre destination*

L'instruction C (call) nécessite une attention particulière car notre firmware exemple en utilise une variante en tant que syscall, permettant de faire appel à des fonctions privilégiées du système (mode kernel):

syscall 1 : reset, stoppe l'exécution du microcontrôleur
syscall 2 : print, affiche les n caractères situés à l'adresse contenue dans r0, avec n contenu dans r1
syscall 3 : getCPUcycles, récupère le nombre de cycles CPU effectués et écrit la valeur sur 16 bits à l'adresse pointée par r0

On peut donc écrire un désassembleur, le second dans ce challenge, et l'utiliser sur le firwmare exemple pour obtenir et commenter le listing suivant :

```
/dev$ cat mcu_fw_disas.txt
       0 : 21 00   movhi      r1, 0
       2 : 11 1B   mov        r1, 1b
       4 : 20 01   movhi      r0, 100
```

```
       6 : 10 8C   mov        r0, 8c
       8 : C0 D2   call     dc (@ a in r15)               ; syscall_2

       a : 20 10   movhi      r0, 1000
       c : 10 00   mov        r0, 0
       e : 21 01   movhi      r1, 100
      10 : 11 7C   mov        r1, 7c
      12 : 22 00   movhi      r2, 0
      14 : 12 0F   mov        r2, f
      16 : C0 3C   call     54 (@ 18 in r15)              ; sub_54
get_key_from_string

      18 : 20 10   movhi      r0, 1000
      1a : 10 00   mov        r0, 0
      1c : 21 01   movhi      r1, 100
      1e : 11 B2   mov        r1, b2
      20 : 22 00   movhi      r2, 0
      22 : 12 29   mov        r2, 29
      24 : C0 76   call     9c (@ 26 in r15)              ; sub_9c : uncipher

      26 : 20 11   movhi      r0, 1100
      28 : 10 00   mov        r0, 0
      2a : C0 B4   call     e0 (@ 2c in r15)              ; syscall_3 : rets 1100
get cpuCycles in r0
      2c : C0 B6   call     e4 (@ 2e in r15)              ; sub_e4 read_word_at_r0
      2e : 5A 00   and        r10, r0, r0                 ; r0 = 2093 ie 8339 cycles
      30 : 21 00   movhi      r1, 0
      32 : 11 24   mov        r1, 24
      34 : 20 01   movhi      r0, 100
      36 : 10 B2   mov        r0, b2
      38 : C0 BE   call     f8 (@ 3a in r15)              ; sub_f8 : strchr
      3a : 51 AA   and        r1, r10, r10
      3c : C1 0A   call     148 (@ 3e in r15); sub_148 : convert_word_to_string
      3e : 21 00   movhi      r1, 0
      40 : 11 29   mov        r1, 29
      42 : 20 01   movhi      r0, 100
      44 : 10 B2   mov        r0, b2
      46 : C0 94   call     dc (@ 48 in r15)              ; syscall_2 : print
      48 : 21 00   movhi      r1, 0
      4a : 11 09   mov        r1, 9
      4c : 20 01   movhi      r0, 100
      4e : 10 A8   mov        r0, a8
      50 : C0 8A   call     dc (@ 52 in r15)              ; syscall_2
      52 : B0 84   jmp        d8                          ; syscall_reset

sub_54:        r0 : destbuf, r1 : Str, r2:sizeStr   get_key_from_string
      54 : 58 00   and        r8, r0, r0
      56 : 59 11   and        r9, r1, r1
      58 : 5A 22   and        r10, r2, r2
      5a : 30 00   xor        r0, r0, r0
      5c : 21 01   movhi      r1, 100
      5e : 11 00   mov        r1, 0
      60 : 22 00   movhi      r2, 0
      62 : 12 01   mov        r2, 1
loc_64:       ; init r0_buf with 0,1,2...
      64 : 73 10   sub        r3, r1, r0
      66 : A0 06   bz           6e                        ; loc_6e
      68 : F0 80   str        r0, [r8+r0]
      6a : 60 02   add        r0, r0, r2
      6c : B3 F6   jmp        64                    ; loc_64  --------^
loc_6e:
      6e : 30 00   xor        r0, r0, r0
      70 : 51 00   and        r1, r0, r0
      72 : 22 00   movhi      r2, 0
      74 : 12 01   mov        r2, 1
      76 : 23 00   movhi      r3, 0
      78 : 13 FF   mov        r3, ff
loc_7a:
```

```
      7a : E4 80   ld         r4, [r8+r0]
      7c : 61 14   add        r1, r1, r4
      7e : 94 0A   div        r4, r0, r10
      80 : 84 4A   mul        r4, r4, r10
      82 : 74 04   sub        r4, r0, r4
      84 : E4 94   ld         r4, [r9+r4]
      86 : 61 14   add        r1, r1, r4
      88 : 51 13   and        r1, r1, r3
      8a : E4 80   ld         r4, [r8+r0]
      8c : E5 81   ld         r5, [r8+r1]
      8e : F5 80   str        r5, [r8+r0]
      90 : F4 81   str        r4, [r8+r1]
      92 : 60 02   add        r0, r0, r2
      94 : 74 30   sub        r4, r3, r0
      96 : AF E2   bz           7a                        ; loc_7a  ---------^
      98 : D0 0F   ret

sub_9a
      9a : B0 02   jmp        9e                     ; loc_9e : cipher


sub_9c:  with r0 keybuf, r1 buf, r2 size buf r1 : uncipher
      9c : B0 00   jmp        9e                     ; loc_9e
loc_9e:
      9e : 58 00   and        r8, r0, r0
      a0 : 59 11   and        r9, r1, r1
      a2 : 5A 22   and        r10, r2, r2
      a4 : 30 00   xor        r0, r0, r0
      a6 : 51 00   and        r1, r0, r0
      a8 : 52 00   and        r2, r0, r0
      aa : 23 00   movhi      r3, 0
      ac : 13 FF   mov        r3, ff
      ae : 24 00   movhi      r4, 0
      b0 : 14 01   mov        r4, 1
loc_b2:                   ; bufkey 256 bytes max
      b2 : 60 24   add        r0, r2, r4
      b4 : 50 03   and        r0, r0, r3
      b6 : E5 80   ld         r5, [r8+r0]
      b8 : 61 15   add        r1, r1, r5
      ba : 51 13   and        r1, r1, r3
      bc : E5 80   ld         r5, [r8+r0]
      be : E6 81   ld         r6, [r8+r1]
      c0 : F6 80   str        r6, [r8+r0]
      c2 : F5 81   str        r5, [r8+r1]
      c4 : 65 56   add        r5, r5, r6
      c6 : 55 53   and        r5, r5, r3
      c8 : E5 85   ld         r5, [r8+r5]
      ca : E6 92   ld         r6, [r9+r2]
      cc : 36 65   xor        r6, r6, r5
      ce : F6 92   str        r6, [r9+r2]
      d0 : 62 24   add        r2, r2, r4
      d2 : 75 A2   sub        r5, r10, r2
      d4 : A7 DC   bz           b2                        ; loc_b2   -------^
      d6 : D0 0F   ret

syscall_reset:
      d8 : C8 01   syscall    1 ; syscall reset
      da : B3 FC   jmp        d8

syscall_2:                                          ; syscall print
      dc : C8 02   syscall    2
      de : D0 0F   ret

syscall_3:
      e0 : C8 03   syscall    3
      e2 : D0 0F   ret

sub_e4:        r0: buffer, read_word_at_r0
```

```
      e4 : 21 00   movhi      r1, 0
      e6 : 11 01   mov        r1, 1
      e8 : 22 01   movhi      r2, 100
      ea : 12 00   mov        r2, 0
      ec : E3 01   ld         r3, [r0+r1]
      ee : 71 11   sub        r1, r1, r1
      f0 : E4 01   ld         r4, [r0+r1]
      f2 : 84 42   mul        r4, r4, r2
      f4 : 40 34   or         r0, r3, r4
      f6 : D0 0F   ret

sub_f8:        r0: buffer, r1 size? strchr with ending char r1
      f8 : 32 22   xor        r2, r2, r2
      fa : 23 00   movhi      r3, 0
      fc : 13 01   mov        r3, 1
loc_fe:
      fe : 34 44   xor        r4, r4, r4
     100 : E4 02   ld         r4, [r0+r2]
     102 : 54 44   and        r4, r4, r4
     104 : A0 08   bz          10e : exit_zero
     106 : 74 41   sub        r4, r4, r1
     108 : A0 06   bz          110 ; exit ptr
     10a : 60 03   add        r0, r0, r3
     10c : B3 F0   jmp        fe                    ; loc_fe
     10e : 30 00   xor        r0, r0, r0
     110 : D0 0F   ret


     112 : 24 10   movhi      r4, 1000
     114 : 14 00   mov        r4, 0
     116 : 25 00   movhi      r5, 0
     118 : 15 0F   mov        r5, f
     11a : 26 00   movhi      r6, 0
     11c : 16 0A   mov        r6, a
     11e : 27 00   movhi      r7, 0
     120 : 17 01   mov        r7, 1
loc_122:
     122 : 92 14   div        r2, r1, r4
     124 : 52 25   and        r2, r2, r5
     126 : 73 26   sub        r3, r2, r6
     128 : A8 06   bnz         130
     12a : 23 00   movhi      r3, 0
     12c : 13 37   mov        r3, 37
     12e : B0 04   jmp       134                    ; loc_134
loc_130:
     130 : 23 00   movhi      r3, 0
     132 : 13 30   mov        r3, 30
loc_134:
     134 : 62 32   add        r2, r3, r2
     136 : 33 33   xor        r3, r3, r3
     138 : F2 03   str        r2, [r0+r3]
     13a : 60 07   add        r0, r0, r7
     13c : 72 47   sub        r2, r4, r7
     13e : A0 06   bz          146                          ; loc_146
     140 : 63 57   add        r3, r5, r7
     142 : 94 43   div        r4, r4, r3
     144 : B3 DC   jmp       122                    ; loc_122
loc_146 :
     146 : D0 0F   ret

sub_148:      r0: buffer, r1: valnum          convert_word_to_string
     148 : 24 27   movhi      r4, 2700
     14a : 14 10   mov        r4, 10                       ; 2710h ie 10000
     14c : 25 00   movhi      r5, 0
     14e : 15 0A   mov        r5, a                        ; 10
     150 : 36 66   xor        r6, r6, r6
     152 : 27 00   movhi      r7, 0
     154 : 17 01   mov        r7, 1
```

```
     156 : 70 07   sub        r0, r0, r7
loc_158:
     158 : 60 07   add        r0, r0, r7
     15a : 92 14   div        r2, r1, r4
     15c : 83 24   mul        r3, r2, r4
     15e : 71 13   sub        r1, r1, r3
     160 : 94 45   div        r4, r4, r5                  ; divide by 10
     162 : 28 00   movhi      r8, 0
     164 : 18 20   mov        r8, 20
     166 : 33 33   xor        r3, r3, r3
     168 : F8 03   str        r8, [r0+r3]
     16a : 46 62   or         r6, r6, r2
     16c : A3 EA   bz          158                        ; loc_158
     16e : 28 00   movhi      r8, 0
     170 : 18 30   mov        r8, 30
     172 : 68 82   add        r8, r8, r2
     174 : F8 03   str        r8, [r0+r3]
     176 : 54 44   and        r4, r4, r4
     178 : A7 DE   bz          158                        ; loc_158
     17a : D0 0F   ret
/dev$
```

La présence de l'instruction load (ld) nous incite à tenter une lecture dans la "Secret memory area" (F000-FBFF) mais sans surprise nous obtenons le message suivant:

```
user@debian:~/Downloads/intelhex-1.5/scripts$ hexdump -C tryreadsecret.bin
00000000  20 f0 10 00 21 00 11 00  e2 10 00 00 00 00 00 00  |
...!...........|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
*
000001dc
user@debian:~/Downloads/intelhex-1.5/scripts$ python bin2hex.py
tryreadsecret.bin fw.hex && python3 upload.py
--------------------------------------------
----- Microcontroller firmware uploader -----
--------------------------------------------

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

System reset.
-- Exception occurred at 0008: Memory access violation.
   r0:F000     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:0008 fault_addr:F000 [S:0 Z:1] Mode:user
CLOSING: Memory access violation.

user@debian:~/Downloads/intelhex-1.5/scripts$
```

Analyse, exploitation et coup de grâce

La présence de l'instruction syscall laisse penser que leur code est exécuté en mode kernel. L'idée est donc de détourner le code d'un ou plusieurs syscalls afin d'effectuer une élévation de privilége qui nous permettrait de lire le contenu de la mémoire secrète. En jouant avec les syscalls on en conclut les points suivants:
- l'appel système getCPUcycles (numéro 3) ne filtre pas beaucoup l'adresse du buffer destination donnée en paramétre puisqu'il est possible d'écrire le nombre de cycles CPU dans toute le plan mémoire sauf dans la zone du noyau entre FD00 et FFFF, ce qui est logique vu le nom donné à cette zone : ROM (Read Only Memory).
- l'appel système print est lui aussi plutôt laxiste puisqu'il permet la lecture de la zone du noyau contrairement à l'instruction load pour ces mêmes adresses.

La stratégie sera donc la suivante:
1. Récupération du code du noyau, en tentant d'afficher la totalité de la zone du noyau. Le microcontrôleur enverra dans sa réponse des caractères non imprimables, il conviendra d'écrire la réponse brute dans un fichier en modifiant le fichier Python qui télécharger le micrologiciel.

```
user@debian:~/Downloads/intelhex-1.5/scripts$ tail upload_binResp.py
    else:
        break

#print(resp.decode("utf-8"))

f = open('fw_response.bin', 'wb')
f.write(resp)
f.close()
s.close()

user@debian:~/Downloads/intelhex-1.5/scripts$
```

2. Désassemblage du code du noyau pour analyse. On réutilise notre désassembleur écrit précédemment.

```
/dev$ cat os_disas.txt
syscall_select:
    fd00 : 50 00   and        r0, r0, r0
    fd02 : A0 6C   bz         fd70                  ; loc_fd70 main

    fd04 : 21 00   movhi      r1, 0
    fd06 : 11 03   movi       r1, 3
    fd08 : 72 10   sub        r2, r1, r0
    fd0a : A8 12   bnz        fd1e                  ; loc_fd1e
print_error_undef_syscall

    fd0c : 22 00   movhi      r2, 0
    fd0e : 12 02   movi       r2, 2
    fd10 : 81 02   mul        r1, r0, r2
    fd12 : 71 12   sub        r1, r1, r2
    fd14 : 20 F0   movhi      r0, f000
    fd16 : 10 00   movi       r0, 0                 ; f000
    fd18 : 60 01   add        r0, r0, r1
    fd1a : C0 94   call       fdb0                  ; sub_fdb0 GetWordAtR0
    fd1c : D0 00   br         r0              ; branch to syscall

loc_fd1e:    print_error_undef_syscall
    fd1e : 21 00   movhi      r1, 0
    fd20 : 11 2B   movi       r1, 2b
```

```
    fd22 : 20 FE   movhi      r0, fe00
    fd24 : 10 5A   movi       r0, 5a    ; addr of "[ERROR] Undefined system call. CPU
halted."
    fd26 : C0 BE   call       fde6                  ; sub_fde6 printf

loc_fd28:    syscall_reset: sets 1 into register_MachineDisabled
    fd28 : 30 00   xor        r0, r0, r0
    fd2a : 21 FC   movhi      r1, fc00
    fd2c : 11 10   movi       r1, 10          ; fc10 : register_MachineDisabled
    fd2e : 22 00   movhi      r2, 0
    fd30 : 12 01   movi       r2, 1
    fd32 : F2 10   str        r2, [r1+r0]
    fd34 : B3 F2   jmp        fd28                  ; loc_fd28

loc_fd36: syscall_print
    fd36 : 20 FC   movhi      r0, fc00
    fd38 : 10 22   movi       r0, 22                ; offset of r1
    fd3a : C0 74   call       fdb0                  ; sub_fdb0 GetWordAtR0

    fd3c : 55 00   and        r5, r0, r0
    fd3e : 20 FC   movhi      r0, fc00
    fd40 : 10 20   movi       r0, 20                ; offset of r0
    fd42 : C0 6C   call       fdb0                  ; sub_fdb0 GetWordAtR0
    fd44 : 51 55   and        r1, r5, r5
    fd46 : C0 9E   call       fde6                  ; sub_fde6 printf
    fd48 : D8 00   retf             ; privileged

sub_fda4:    syscall_3_get_cpu_cycles
    fd4a : 20 FC   movhi      r0, fc00
    fd4c : 10 20   movi       r0, 20
    fd4e : C0 60   call       fdb0                  ; sub_fdb0 GetWordAtR0
    fd50 : 26 FC   movhi      r6, fc00
    fd52 : 16 12   movi       r6, 12                ; fc12 addr of cpuCycles register
    fd54 : 21 00   movhi      r1, 0
    fd56 : 11 01   movi       r1, 1
    fd58 : 34 44   xor        r4, r4, r4

loc_fd5a:
    fd5a : E5 61   ld         r5, [r6+r1]           ; get byte at fc13
    fd5c : E2 64   ld         r2, [r6+r4]           ; get byte at fc12
    fd5e : E3 64   ld         r3, [r6+r4]           ; get byte at fc12
    fd60 : 73 32   sub        r3, r3, r2
    fd62 : A7 F6   bz         fd5a            ; loc_fd5a loop until cpucycles changes
    fd64 : 23 01   movhi      r3, 100
    fd66 : 13 00   movi       r3, 0
    fd68 : 82 23   mul        r2, r2, r3
    fd6a : 41 25   or         r1, r2, r5            ; r1 was stored at fc12
    fd6c : C0 56   call       fdc4                  ; sub_fdc4 storeR1AtR0
    fd6e : D8 00   retf             ; privileged

loc_fd70:    main : print "System reset.", sets syscalls
    fd70 : 21 00   movhi      r1, 0
    fd72 : 11 0E   movi       r1, e
    fd74 : 20 FE   movhi      r0, fe00
    fd76 : 10 86   movi       r0, 86                ; addr of "System reset."
    fd78 : C0 6C   call       fde6                  ; sub_fde6 printf
    fd7a : 24 00   movhi      r4, 0
    fd7c : 14 02   movi       r4, 2
    fd7e : 21 FD   movhi      r1, fd00
    fd80 : 11 28   movi       r1, 28                ; fd28 syscall_reset
    fd82 : 20 F0   movhi      r0, f000
    fd84 : 10 00   movi       r0, 0                 ; f000
    fd86 : C0 3C   call       fdc4                  ; sub_fdc4 storeR1AtR0
    fd88 : 60 04   add        r0, r0, r4
    fd8a : 21 FD   movhi      r1, fd00
    fd8c : 11 36   movi       r1, 36                ; fd36 in f002
    fd8e : C0 34   call       fdc4                  ; sub_fdc4 storeR1AtR0
    fd90 : 60 04   add        r0, r0, r4
```

```
    fd92 : 21 FD   movhi      r1, fd00
    fd94 : 11 4A   movi       r1, 4a                ; fd4a in f004
    fd96 : C0 2C   call       fdc4                  ; sub_fdc4 storeR1AtR0
    fd98 : 20 FC   movhi      r0, fc00
    fd9a : 10 20   movi       r0, 20                ; fc20 register r0
    fd9c : 31 11   xor        r1, r1, r1
    fd9e : 22 00   movhi      r2, 0
    fda0 : 12 36   movi       r2, 36
    fda2 : C0 32   call       fdd6     ; sub_fdd6 memset resets all user registers
    fda4 : 20 FC   movhi      r0, fc00
    fda6 : 10 3A   movi       r0, 3a                ; fc3a register     r13
    fda8 : 21 EF   movhi      r1, ef00
    fdaa : 11 FE   movi       r1, fe
    fdac : C0 16   call       fdc4     ; sub_fdc4 storeR1AtR0 : sets user r13 to EFFE
    fdae : D8 00   retf           ; privileged : transfer execution to user at 0000

sub_fdb0: GetWordAtR0
    fdb0 : 21 00   movhi      r1, 0
    fdb2 : 11 01   movi       r1, 1
    fdb4 : 22 01   movhi      r2, 100
    fdb6 : 12 00   movi       r2, 0
    fdb8 : E3 01   ld         r3, [r0+r1]
    fdba : 71 11   sub        r1, r1, r1
    fdbc : E4 01   ld         r4, [r0+r1]
    fdbe : 84 42   mul        r4, r4, r2
    fdc0 : 40 34   or         r0, r3, r4
    fdc2 : D0 0F   ret

sub_fdc4: storeR1AtR0
    fdc4 : 22 00   movhi      r2, 0
    fdc6 : 12 01   movi       r2, 1
    fdc8 : 23 01   movhi      r3, 100
    fdca : 13 00   movi       r3, 0
    fdcc : F1 02   str        r1, [r0+r2]
    fdce : 72 22   sub        r2, r2, r2
    fdd0 : 91 13   div        r1, r1, r3
    fdd2 : F1 02   str        r1, [r0+r2]
    fdd4 : D0 0F   ret

sub_fdd6:    loc_fdd6                         ; memset r0 with r1 size r2
    fdd6 : 23 00   movhi      r3, 0
    fdd8 : 13 01   movi       r3, 1
    fdda : 52 22   and        r2, r2, r2
    fddc : A0 06   bz         fde4     ; loc_fde4          exit if r2 = 0
    fdde : 72 23   sub        r2, r2, r3
    fde0 : F1 02   str        r1, [r0+r2]
    fde2 : B3 F2   jmp        fdd6                  ; loc_fdd6          start
loc_fde4:
    fde4 : D0 0F   ret

sub_fde6:    syscall_printr0: buf, r1: size
    fde6 : 5E 00   and        r14, r0, r0
    fde8 : 2D FC   movhi      r13, fc00
    fdea : 1D 00   movi       r13, 0                ; fc00 register_printChar
    fdec : 2C F0   movhi      r12, f000
    fdee : 1C 00   movi       r12, 0
    fdf0 : 38 88   xor        r8, r8, r8
    fdf2 : 59 88   and        r9, r8, r8
    fdf4 : 2A 00   movhi      r10, 0
    fdf6 : 1A 01   movi       r10, 1
    fdf8 : 3B BB   xor        r11, r11, r11

loc_fdfa:
    fdfa : 51 11   and        r1, r1, r1
    fdfc : A0 1A   bz         fe18                  ; loc_fe18 exit_syscall_print
    fdfe : 69 E8   add        r9, r14, r8
    fe00 : 79 9C   sub        r9, r9, r12           ; cmp to f000
    fe02 : A8 08   bnz        fe0c                  ; loc_fe0c do_printf
```

```
    fe04 : 69 E8   add        r9, r14, r8
    fe06 : 79 9D   sub        r9, r9, r13           ; cmp to fc00
    fe08 : AC 02   bnz        fe0c                  ; loc_fe0c do_printf
    fe0a : B0 0E   jmp        fe1a     ; loc_fe1a error_print_unallowed_address

loc_fe0c:    ; do_printf
    fe0c : 39 99   xor        r9, r9, r9
    fe0e : E9 E8   ld         r9, [r14+r8]
    fe10 : F9 DB   str        r9, [r13+r11]
    fe12 : 68 8A   add        r8, r8, r10
    fe14 : 71 1A   sub        r1, r1, r10
    fe16 : B3 E2   jmp        fdfa                  ; loc_fdfa

loc_fe18:    ; exit_syscall_print
    fe18 : D0 0F   ret

loc_fe1a:    error_print_unallowed_address
    fe1a : 21 00   movhi      r1, 0
    fe1c : 11 33   movi       r1, 33
    fe1e : 20 FE   movhi      r0, fe00
    fe20 : 10 26   movi       r0, 26 ; address of "[ERROR] Printing at unallowed
address. CPU halted."
    fe22 : C3 C2   call       fde6                  ; sub_fde6 printf
    fe24 : B3 02   jmp        fd28                  ; loc_fd28 syscall_reset
/dev$
```

3. On remarque que les adresses des appels systèmes sont initialisés par le noyau dans la mémoire secrète (aux adresses 0xf000, 0xf002 et 0xf004, cf code situé à l'adresse 0xfd7a). On va donc pouvoir utiliser l'appel système getCPUcycles pour modifier l'appel système Reset afin de remplacer son adresse située dans le noyau par la nôtre présente dans la zone du firmware. Le dernier défi reste à définir un algorithme pour obtenir un nombre de cycles CPU égal à l'adresse de notre fonction qui pourra alors récupérer le contenu complet de la mémoire secrète. Une simple boucle fera l'affaire.

4. On exécute notre firmware d'exploitation qui récupère le contenu de la zone secrète et on peut finalement l'afficher de façon un peu plus sympa après toutes ces épreuves:

```
/dev$ ./mcudis dumper.bin 0 13a 0
Loading code (size 13ah) from offset 0h in file dumper.bin loaded at address
0h

       0 : B0 22   jmp        24                ; loc_24

sub_2:
       2 : 20 F0   movhi      r0, f000
       4 : 10 00   movi       r0, 0
       6 : C8 03   Syscall    3                      ; sets new_syscall_1
       8 : C8 01   Syscall    1
       a : D0 0F   ret

loc_24:
      24 : 28 7C   movhi      r8, 7c00 ; with value 7c6c here,
      26 : 18 6C   movi       r8, 6c   ; we get 00A0 CPU cycles!
      28 : 29 00   movhi      r9, 0
      2a : 19 01   movi       r9, 1
loc_2c:                ; loop to obtain good cpu cycles value!
      2c : 78 89   sub        r8, r8, r9
      2e : AF FC   bnz        2c                ; loc_2c ---------------^
      30 : C3 D0   call       2                      ; sub_2
      32 : C8 01   Syscall    1
```

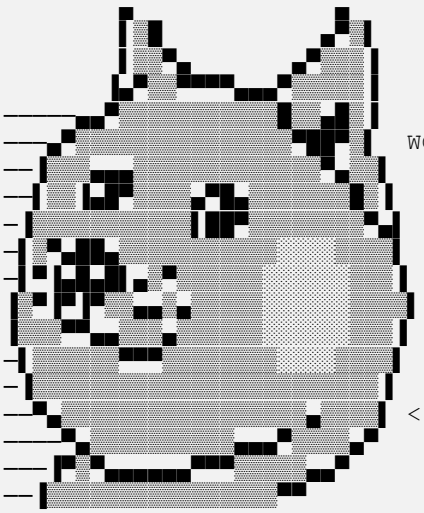
```
new_syscall_1:
      a0 : B0 3C   jmp        de ; loc_de do the print without restriction!

loc_de:     ; inspired from original print syscall
      de : 20 F0   movhi      r0, f000
      e0 : 10 00   movi       r0, 0
      e2 : 21 0C   movhi      r1, c00
      e4 : 11 00   movi       r1, 0
      e6 : 5E 00   and        r14, r0, r0
      e8 : 2D FC   movhi      r13, fc00
      ea : 1D 00   movi       r13, 0
      ec : 2C F0   movhi      r12, f000
      ee : 1C 00   movi       r12, 0
      f0 : 38 88   xor        r8, r8, r8
      f2 : 59 88   and        r9, r8, r8
      f4 : 2A 00   movhi      r10, 0
      f6 : 1A 01   movi       r10, 1
      f8 : 3B BB   xor        r11, r11, r11
loc_fa:
      fa : 51 11   and        r1, r1, r1
      fc : A0 1A   bz         118              ; loc_118
      fe : 69 E8   add        r9, r14, r8
     100 : 79 9C   sub        r9, r9, r12
     102 : B0 08   jmp        10c              ; loc_10c
     104 : 69 E8   add        r9, r14, r8
     106 : 79 9D   sub        r9, r9, r13
     108 : AC 02   bnz        10c              ; loc_10c
     10a : B0 0E   jmp        11a              ; loc_11a
loc_10c:
     10c : 39 99   xor        r9, r9, r9
     10e : E9 E8   ld         r9, [r14+r8]
     110 : F9 DB   str        r9, [r13+r11]
     112 : 68 8A   add        r8, r8, r10
     114 : 71 1A   sub        r1, r1, r10
     116 : B3 E2   jmp        fa               ; loc_fa
loc_118:
     118 : 24 00   movhi      r4, 0
loc_11a:
     11a : 14 02   movi       r4, 2
     11c : 21 FD   movhi      r1, fd00
     11e : 11 28   movi       r1, 28
     120 : 20 F0   movhi      r0, f000
     122 : 10 00   movi       r0, 0
     124 : C0 02   call       128              ; sub_128
     126 : D8 00   retf                        ; privileged

sub_128:
     128 : 22 00   movhi      r2, 0
     12a : 12 01   movi       r2, 1
     12c : 23 01   movhi      r3, 100
     12e : 13 00   movi       r3, 0
     130 : F1 02   str        r1, [r0+r2]
     132 : 72 22   sub        r2, r2, r2
     134 : 91 13   div        r1, r1, r3
     136 : F1 02   str        r1, [r0+r2]
     138 : D0 0F   ret
/dev$
```

On peut alors afficher une partie de la réponse de façon à faire apparaître le sésame tant attendu !

```
>>> print binascii.unhexlify(str_dump_reponse)
```



```
WOW

  SUCH EXPLOIT

    VERY CHALLENGING

    SO OPERATIONAL

  MUCH WIN

<66a65dc050ec0c84cf1dd5b3bbb75c8c@challenge.sstic.org>


>>>
```

Conclusion

Voilà le challenge SSTIC 2014 est terminé. J'ai vraiment manqué de temps pour finaliser l'écriture de cette solution mais c'est la première fois que je pousse le jeu jusqu'au bout et ce fut un vraiment un challenge très intéressant! Merci aux concepteurs de l'avoir réalisé. J'ai découvert l'architecture ARM 64 bits, j'ai enfin utilisé QEMU, j'ai progressé en Python... Et enfin grand MERCI à Julien Dusser pour son expertise QEMU/grep/sed/graphviz qui m'ont également bien aidés! Vivement 2015!

Annexes

Le code du désassembleur de la machine virtuelle :

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

// types
typedef unsigned short        WORD;
typedef unsigned char BYTE;
typedef unsigned int  DWORD;
typedef int           BOOL;

// constants
#define MAX_DATASTRING_SIZE                 2048
#define MAX_LEN                                     256
#define TRUE                                1
#define FALSE                               0


#define VM_ADDIS_OP           0x0
#define VM_ADDI_OP            0x1
#define VM_MOVIND_OP          0x2
#define VM_3_OP                       0x3
#define VM_LDB_OP             0x4
#define VM_5_OP                       0x5
#define VM_6_OP                       0x6
#define VM_STB_OP             0x7
#define VM_Bcond_OP           0x8
#define VM_9_OP                       0x9
#define VM_XOR_OP             0xa
#define VM_OR_OP              0xb
#define VM_AND_OP             0xc
#define VM_SHL_OP             0xd
#define VM_SHR_OP             0xe
#define VM_f_OP                       0xf
#define VM_10_OP              0x10
#define VM_11_OP              0x11
#define VM_ADD_OP             0x12
#define VM_SUB_OP             0x13
#define VM_14_OP              0x14
#define VM_15_OP              0x15
#define VM_INC_OP             0x16
#define VM_DEC_OP             0x17
#define VM_18_OP              0x18
#define VM_19_OP              0x19
#define VM_1a_OP              0x1a
#define VM_1b_OP              0x1b
#define VM_HLT_OP             0x1c
#define VM_SC_OP              0x1d
#define VM_TBC_OP             0x1e
#define VM_1f_OP              0x1f

const char* opCodes[32] =             {       "addis", "addi", "mov", "3",
                                              "ld", "5", "6", "sb",
                                              "B", "9", "xor", "or",
                                              "and", "shl", "shr", "f",
                                              "10", "11", "add", "sub",
                                              "14", "15", "inc", "dec",
                                              "18", "19", "1a", "1b",
                                              "hlt", "syscall", "TBitCnt", "1f"};

const char* flags [8] = { "ranch",".1","Z","NZ","Neg","sGTZ","LE",".7"};

////////////////////////////////////////////////////////////////////////////
DWORD GetCondReg0(DWORD inst)
{
        return ((inst >> 9) & 0xf);
```

```
}

////////////////////////////////////////////////////////////////////////////
DWORD GetCondFlag(DWORD inst)
{
	return ((inst >> 13) & 0x7);
}

////////////////////////////////////////////////////////////////////////////
DWORD GetCondSense(DWORD inst)
{
	return ((inst >> 8) & 0x1);
}


////////////////////////////////////////////////////////////////////////////
DWORD GetReg0(DWORD inst)
{
	return ((inst >> 8) & 0xf);
}

////////////////////////////////////////////////////////////////////////////
DWORD GetReg1(DWORD inst)
{
	return ((inst >> 12) & 0xf);
}

////////////////////////////////////////////////////////////////////////////
DWORD GetImm0(DWORD inst)
{
	return ((inst >> 12) & 0xffff);
}

////////////////////////////////////////////////////////////////////////////
DWORD GetImm1(DWORD inst)
{
	return ((inst >> 16) & 0xffff);
}


////////////////////////////////////////////////////////////////////////////
BOOL VMDis(BYTE* Code, DWORD szCode, DWORD codeAddress)
{

	DWORD		pc;
	DWORD		offsetpc=0;
	DWORD		szInst = 0;
	DWORD		inst;
	BYTE*		pByteInst;
	BYTE		opCode = 0;
	BYTE		Branch[MAX_LEN];
	BYTE		Condition[MAX_LEN];

	pByteInst = (BYTE*)&inst;
	pc = codeAddress;

	while(offsetpc < szCode)
	{

		// read opcode to get inst size
		opCode = Code[offsetpc];
		if(opCode >= 0x20)
		{
			printf("Invalid Opcode %02x at pc %Xh!\n",opCode,pc);
			return FALSE;
		}

		switch(opCode)
		{
		case VM_SC_OP:
		case VM_XOR_OP:
		case VM_SUB_OP:
		case VM_DEC_OP:
		case VM_AND_OP:
		case VM_SHL_OP:
		case VM_SHR_OP:
		case VM_INC_OP:
```

```
case VM_ADD_OP:
case VM_OR_OP:
case VM_TBC_OP:
case VM_HLT_OP:
	szInst = 2;
	inst =  ((WORD*)&Code[offsetpc])[0];
	break;
default:
	szInst = 4;
	inst = ((DWORD*)&Code[offsetpc])[0];
	break;
}



// disas
switch(opCode)
{

// 32 bits instructions
case VM_ADDIS_OP:
	printf("%8x : %02X %02X %02X %02X %-10s r%02d, %x\n",
		offsetpc+pc,pByteInst[3],pByteInst[2],pByteInst[1],pByteInst[0],
		opCodes[opCode], GetReg0(inst), GetImm0(inst));
	break;

case VM_ADDI_OP:
	printf("%8x : %02X %02X %02X %02X %-10s r%02d, %x\n",
		offsetpc+pc,pByteInst[3],pByteInst[2],pByteInst[1],pByteInst[0],
		opCodes[opCode], GetReg0(inst), GetImm0(inst));
	break;

case VM_MOVIND_OP:
	if(GetReg1(inst) == 0)
	{
		printf("%8x : %02X %02X %02X %02X %-10s r%02d, r%02d\n",
		offsetpc+pc,pByteInst[3],pByteInst[2],pByteInst[1],pByteInst[0],
		opCodes[opCode], GetReg0(inst), 1+GetImm1(inst)/4);
	}
	else
	{
		printf("%8x : %02X %02X %02X %02X %-10s r%02d, [r%02d + %x]\n",
		offsetpc+pc,pByteInst[3],pByteInst[2],pByteInst[1],pByteInst[0],
		opCodes[opCode], GetReg0(inst), GetReg1(inst), GetImm1(inst));
	}
	break;

case VM_Bcond_OP:
	sprintf((char*)Branch, "%s%s", opCodes[opCode], flags[GetCondFlag(inst)]);
	if(GetCondFlag(inst))
	{
		sprintf((char*)Condition, " r%02d, [F%x, S%x]",
		GetCondReg0(inst), GetCondFlag(inst), GetCondSense(inst));
	}
	else
	{
		sprintf((char*)Condition, " ");
	}
	printf("%8x : %02X %02X %02X %02X %-10s %x %s\n",
		offsetpc+pc,pByteInst[3],pByteInst[2],pByteInst[1],pByteInst[0],
		Branch, GetImm1(inst), Condition);
	break;

case VM_LDB_OP:
case VM_STB_OP:
	printf("%8x : %02X %02X %02X %02X %-10s r%02d, [r%02d + %x]\n",
		offsetpc+pc,pByteInst[3],pByteInst[2],pByteInst[1],pByteInst[0],
		opCodes[opCode], GetReg0(inst), GetReg1(inst), GetImm1(inst));
	break;

// 16 bits instructions
case VM_XOR_OP:
case VM_ADD_OP:
case VM_SUB_OP:
case VM_AND_OP:
case VM_SHL_OP:
case VM_OR_OP:
```

```
		case VM_TBC_OP:
		case VM_SHR_OP:
			printf("%8x : %02X %02X	%-10s r%02d, r%02d\n",
				offsetpc+pc,pByteInst[1],pByteInst[0], opCodes[opCode],
GetReg0(inst), GetReg1(inst));
			break;

		case VM_DEC_OP:
		case VM_INC_OP:
			printf("%8x : %02X %02X	%-10s r%02d\n",
				offsetpc+pc,pByteInst[1],pByteInst[0], opCodes[opCode],
GetReg0(inst));
			break;

		case VM_SC_OP:
		case VM_HLT_OP:
			printf("%8x : %02X %02X	%-10s\n",
				offsetpc+pc,pByteInst[1],pByteInst[0], opCodes[opCode]);
			break;


		default:
			if(szInst == 2)
				printf("Not resolved yet : %08x : op %02X, full inst %04x\n",
				offsetpc+pc, opCode, ((WORD*)&Code[pc])[0]);
			else
				printf("Not resolved yet : %08x : op %02X, full inst %08x\n",
				offsetpc+pc, opCode, ((DWORD*)&Code[pc])[0]);
			break;
		}

		// inc pc
		offsetpc = offsetpc +szInst;
	}

	return TRUE;
}



////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
int main(int argc, char** argv)
{
	FILE*			fp;
	DWORD			szCode;
	BYTE*			pCode;
	DWORD			codeOffset;
	DWORD			codeLoadAddress;

	if(argc != 5)
	{
		printf("Usage : %s <fw file> <code file offset> <code size> <code load
address>\n",argv[0]);
		printf("All numeric arguments must be given in hex format!\n\n");
		return (0);
	}


	// get args
	codeOffset = strtoul(argv[2],NULL,16);
	szCode = strtoul(argv[3],NULL,16);
	codeLoadAddress = strtoul(argv[4],NULL,16);
	printf("Loading code (size %xh) from offset %xh in file %s loaded at address %xh\n",
		szCode, codeOffset, argv[1], codeLoadAddress);

	fp = fopen(argv[1],"r");
	if(fp == NULL)
	{
		perror("Open file to disassemble failed");
		return(errno);
	}
```

```
	// read code
	pCode = malloc(szCode);
	if(pCode == NULL)
	{
		perror("malloc failed");
		return(errno);
	}

	if(fseek(fp, codeOffset, SEEK_SET) == -1)
	{
		perror("Cannot get to code offset");
		free(pCode);
		return(errno);
	}

	if(fread(pCode, 1, szCode, fp) != szCode)
	{
		perror("Reading input file failed");
		free(pCode);
		return(errno);
	}

	VMDis(pCode, szCode, codeLoadAddress);

	// free stuff
	fclose(fp);
	free(pCode);

	return (0);
}
```

Le code du désassembleur du microcontrôleur :

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

// types
typedef unsigned short	WORD;
typedef unsigned char BYTE;
typedef unsigned int  DWORD;
typedef int		BOOL;

// constants
#define MAX_DATASTRING_SIZE   2048
#define MAX_LEN			256
#define TRUE			1
#define FALSE			0

// Macros for endianess
#define WEndianGet(x,bBigEndian)	(GetWord(x,bBigEndian))
#define WEndianConv(x,bBigEndian)	x = GetWord(x,bBigEndian)
#define WbGet(x)			(GetWord(x,1))
#define WbConv(x)			x = GetWord(x,1)

////////////////////////////////////////////////////////////////////////////
// GetWord() : converts val to low endian if input is big endian
// IN:  val
//		bBigEndian true if input is from big endian
// OUT: converted val
////////////////////////////////////////////////////////////////////////////
WORD GetWord(WORD val, BOOL bBigEndian)
{
	if(bBigEndian)
		return( ((val>>8) & 0xff) | ((val<<8) & 0xff00) );
	else
		return(val);
}

#define MCU_ADDI		0x1
#define MCU_ADDIS		0x2
#define MCU_XOR			0x3
```

```
#define MCU_OR                0x4
#define MCU_AND                       0x5
#define MCU_ADD                       0x6
#define MCU_SUB                       0x7
#define MCU_MUL                       0x8
#define MCU_DIV                       0x9
#define MCU_BZ                0xA
#define MCU_JREL              0xB
#define MCU_CREL              0xC
#define MCU_RET                       0xD
#define MCU_LDR                       0xE
#define MCU_STR                       0xF

const char* opCodes[16] = {
        "0", "addi", "addis","xor",
        "or", "and", "add", "sub",
        "mul", "div", "b", "jmp",
        "call", "br", "ld", "str",
        };

////////////////////////////////////////////////////////////////////////////////
DWORD GetReg0(DWORD inst)
{
        return ((inst >> 8) & 0xf);
}

////////////////////////////////////////////////////////////////////////////////
DWORD GetReg1(DWORD inst)
{
        return ((inst >> 4) & 0xf);
}

////////////////////////////////////////////////////////////////////////////////
DWORD GetReg2(DWORD inst)
{
        return ((inst >> 0) & 0xf);
}
////////////////////////////////////////////////////////////////////////////////
DWORD GetImm0(DWORD inst)
{
        return ((inst >> 0) & 0xff);
}

////////////////////////////////////////////////////////////////////////////////
WORD GetOffset(DWORD inst)
{
        WORD    offset;
        offset = (inst & 0x3ff);
        if(offset & 0x200)
        {
                offset = 0xFC00 | offset;
                //PDBG("ii %x offset %x\n",inst,offset);
        }
        return (offset);
}

////////////////////////////////////////////////////////////////////////////////
BOOL McuDis(BYTE* Code, DWORD szCode, DWORD codeAddress)
{

        DWORD           pc;
        DWORD           offsetpc=0;
        DWORD           szInst = 0;
        DWORD           inst;
        BYTE*           pByteInst;
        BYTE            opCode = 0;
        DWORD           notflag;

        pByteInst = (BYTE*)&inst;
        pc = codeAddress;
        szInst = 2;

        while(offsetpc < szCode)
        {
                // read opcode and instruction
                opCode = Code[offsetpc] >>4;
                inst = WbGet(((WORD*)&Code[offsetpc])[0]);
```

```
		// disas
		switch(opCode)
		{

		case MCU_ADDIS:
			printf("%8x : %02X %02X   %-10s r%d, %x\n",
				offsetpc+pc,pByteInst[1],pByteInst[0],
				opCodes[opCode], GetReg0(inst), GetImm0(inst)<<8);
			break;
		case MCU_ADDI:
			printf("%8x : %02X %02X   %-10s r%d, %x\n",
				offsetpc+pc,pByteInst[1],pByteInst[0],
				opCodes[opCode], GetReg0(inst), GetImm0(inst));
			break;

		case MCU_XOR:
		case MCU_SUB:
		case MCU_OR:
		case MCU_ADD:
		case MCU_MUL:
		case MCU_DIV:
		case MCU_AND:
			printf("%8x : %02X %02X   %-10s r%d, r%d, r%d\n",
				offsetpc+pc,pByteInst[1],pByteInst[0], opCodes[opCode],
				GetReg0(inst), GetReg1(inst),GetReg2(inst));
			break;

		case MCU_JREL:
			printf("%8x : %02X %02X   %-10s %hx\t\t\t; loc_%hx\n",
				offsetpc+pc,pByteInst[1],pByteInst[0], opCodes[opCode],
				(short)(
(short)(offsetpc+pc)+(short)szInst+(short)GetOffset(inst)),
				(short)(
(short)(offsetpc+pc)+(short)szInst+(short)GetOffset(inst))
				);
			break;

		case MCU_BZ:
			if(inst & 0x800) notflag = 1;
			else notflag = 0;

			printf("%8x : %02X %02X   %1s%-9s %hx\t\t\t; loc_%hx\n",
				(offsetpc+pc),pByteInst[1],pByteInst[0],
opCodes[opCode],notflag?"nz":"z",
				(short)(
(short)(offsetpc+pc)+(short)szInst+(short)GetOffset(inst)),
				(short)(
(short)(offsetpc+pc)+(short)szInst+(short)GetOffset(inst))
				);
			break;

		case MCU_CREL:
			if((inst & 0x800) == 0)
			{
				printf("%8x : %02X %02X   %-10s %hx\t\t\t; sub_%hx\n",
					(offsetpc+pc),pByteInst[1],pByteInst[0], opCodes[opCode],
					(short)(offsetpc+pc)+(short)szInst+(short)GetOffset(inst),
					(short)(offsetpc+pc)+(short)szInst+(short)GetOffset(inst)
					);
			}
			else
			{
				printf("%8x : %02X %02X   %-10s %x\n",
					(offsetpc+pc),pByteInst[1],pByteInst[0],
"Syscall",GetImm0(inst));
			}
			break;

		case MCU_RET:
			if(GetReg0(inst) == 8)
			{
				printf("%8x : %02X %02X   %-10s\n",
					(offsetpc+pc),pByteInst[1],pByteInst[0],
					"retf\t\t\t\t\t; privileged");
			}
			else
```

```
                        {
                                if(GetOffset(inst))
                                {
                                        printf("%8x : %02X %02X   %-10s\n",
                                                (offsetpc+pc),pBytеInst[1],pByteInst[0], "ret");
                                }
                                else
                                {
                                        printf("%8x : %02X %02X   %-10s r%d\n",
                                                (offsetpc+pc),pByteInst[1],pByteInst[0],
                                                opCodes[opCode],GetOffset(inst)&0xf);
                                }
                        }
                        break;

                case MCU_LDR:
                case MCU_STR:
                        printf("%8x : %02X %02X   %-10s r%d, [r%d+r%d]\n",
                                (offsetpc+pc),pByteInst[1],pByteInst[0],
                                opCodes[opCode], GetReg0(inst), GetReg1(inst),GetReg2(inst));
                        break;

                default:
                        printf("Not resolved yet : %04x : op %02X, full inst %04x\n",
                                (offsetpc+pc), opCode, inst);
                        break;
                }

                // increment offsetpc
                offsetpc = offsetpc +szInst;
        }

        return TRUE;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
int main(int argc, char** argv)
{
        FILE*                           fp;
        DWORD                           szCode;
        BYTE*                           pCode;
        DWORD                           codeOffset;
        DWORD                           codeLoadAddress;

        if(argc != 5)
        {
                printf("Usage : %s <fw file> <code file offset> <code size> <code load
address>\n",
                        argv[0]);
                printf("All numeric arguments must be given in hex format!\n\n");
                return (0);
        }


        // get args
        codeOffset = strtoul(argv[2],NULL,16);
        szCode = strtoul(argv[3],NULL,16);
        codeLoadAddress = strtoul(argv[4],NULL,16);
        printf("Loading code (size %xh) from offset %xh in file %s loaded at address %xh\n",
                szCode, codeOffset, argv[1], codeLoadAddress);

        fp = fopen(argv[1],"r");
        if(fp == NULL)
        {
                perror("Open file to disassemble failed");
                return(errno);
        }

        // read code
        pCode = malloc(szCode);
        if(pCode == NULL)
```

```
        {
                perror("malloc failed");
                return(errno);
        }

        if(fseek(fp, codeOffset, SEEK_SET) == -1)
        {
                perror("Cannot get to code offset");
                free(pCode);
                return(errno);
        }

        if(fread(pCode, 1, szCode, fp) != szCode)
        {
                perror("Reading input file failed");
                free(pCode);
                return(errno);
        }

        McuDis(pCode, szCode, codeLoadAddress);

        // free stuff
        fclose(fp);
        free(pCode);

        return (0);
}
```