Challenge SSTIC 2014

Vincent Fargues <vincent.fargues@thalesgroup.com>

5 mai 2014

Résumé

Le challenge SSTIC 2014 consiste à trouver une adresse email dans une trace USB. Cette trace USB contient un transfert de fichier via ADB. Ce fichier est un binaire ARM64 qui contient une machine virtuelle exécutant du code chiffré avec ChaCha et une clef fixe. Ce code déchiffre une charge utile avec un LFSR simple. En supposant que la charge claire finit par zéro, la clef est retrouvée en inversant le LFSR. La charge utile obtenue est un fichier zip contenant une image de Firmware de micro-contrôleur inconnu, accessible par un oracle en ligne. Ce micro-contrôleur exécute le code fournit dans un mode particulier et une partie du code ROM présente une vulnérabilité permettant d'extraire la RAM sécurisée théoriquement non accessible de celui-ci. C'est dans cette zone mémoire que l'adresse mail objectif du challenge est retrouvée.

Table des matières

1 Trace USB

Le fichier `usbtrace.xz` fourni comme énoncé du challenge est un email contenant une trace textuelle obtenue par `usbmon`.

```
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 1
    f030000 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 <
```

Listing 1: Extrait d'usbtrace.xz

Le protocole USB est un protocole maître/esclave, le contrôleur pilotant des "endpoints". Les échanges de données peuvent être des échanges de signalisation (*Setup*), des échanges de données synchrones (*Isochronous*) ou asynchrones (Bulk).

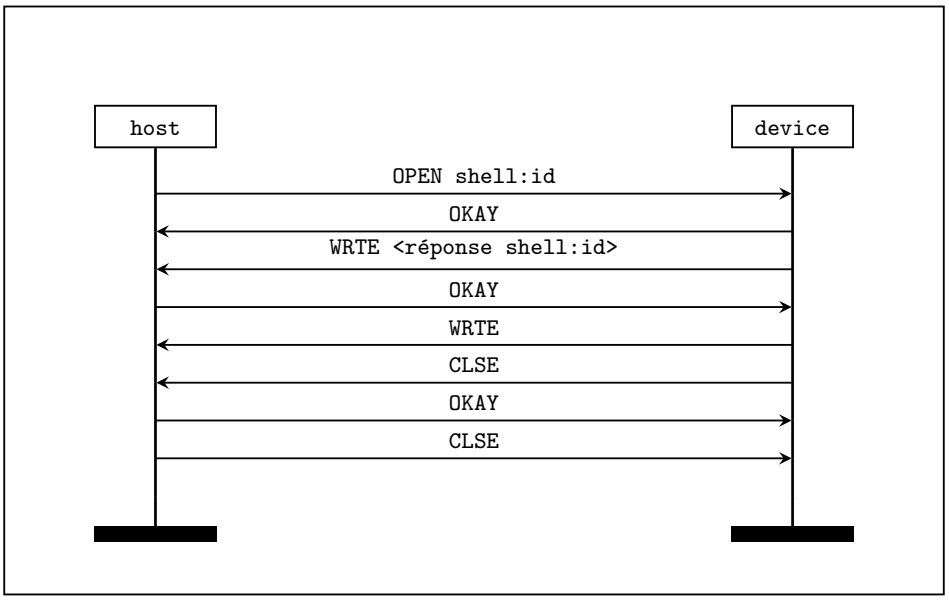
Les paquets intéressants sont les paquets de données *Bulk*, qui servent aux transferts de données en masse et sont les plus fréquents en volume dans cette trace.

1.1 Paquets de données USB

Le flux de données est reconstruit en filtrant les lignes marquées `Bi:2:008:5` (*Endpoint Bulk In*) et `Bo:2:008:3` (*Endpoint Bulk Out*). Les lignes terminant par une suite hexadécimale précédée d'un signe égal sont des transferts de données.

Ensuite, les données sont converties en binaire à partir de leur représentation hexadécimale.

Des séquences telle celle de la figure suivante sont obtenues.



Les différents marqueurs présents dans les paquets permettent d'identifier ceux-ci comme faisant partie d'un échange suivant le protocole ADB (*Android Debug Bridge*).

1.2 Protocole *Android Debug Bridge Protocol*

Le protocole ADB est un protocole utilisé pour le pilotage d'un terminal Android afin de faciliter le travail du développeur. Il peut être porté par une liaison TCP ou USB. Ce protocole basique est décrit dans les sources d'Android et sert de véhicule de base à un certains nombre de services, dont certains sont utilisés au sein de la trace étudiée.

```
> shell:id
< uid=2000(shell) gid=2000(shell) groups=1003(graphics),1004(input),1007(log),1009

> shell:uname -a
< Linux localhost 4.1.0-g4e972ee #1 SMP PREEMPT Mon Feb 24 21:16:40 PST 2015 armv8

> sync LIST /sdcard/
< File '.'
< File '..'
< File 'Samsung'
< File 'Android'
< File '.face'
< File 'Music'
< File 'Podcasts'
< File 'Ringtones'
< File 'Alarms'
< File 'Notifications'
< File 'Pictures'
< File 'Movies'
< File 'Download'
< File 'DCIM'
< File 'Documents'
< File '.SPenSDK30'
< File '.enref'
< File 'Nearby'
< File 'Playlists'
< File '.pla'
< File '.estrongs'
< File 'backups'
< File 'clockworkmod'
< File 'CyanogenMod'
< File 'mmc1'
< DONE
> QUIT

> sync LIST /sdcard/Documents/
< File '.'
< File '..'
< File 'CSW-2014-Hacking-9.11_uncensored.pdf'
< File 'NATO_Cosmic_Top_Secret.gpg'
< DONE
> QUIT

> sync LIST /data/local/tmp
< File '.'
< File '..'
< DONE
> QUIT

> STAT /data/local/tmp/badbios.bin
< STAT <data>

> SEND /data/local/tmp/badbios.bin,33261
[... data ...]
< DONE

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

> sync LIST /data/local/tmp
```

```
< File '.'
< File '..'
< File 'badbios.bin'
< DONE
> QUIT
```

Listing 2: Demandes de services ADB dans la trace

La trace ADB (Listing 2) ne semble pas contenir beaucoup d'informations mis à part une date futuriste et un transfert du fichier `badbios.bin` via le sous-protocole Sync d'ADB. Le code utilisé afin d'extraire le fichier badbios.bin est disponible en Annexe 1 (listing 39).

Le fichier obtenu (`badbios.bin`) est l'énigme de l'étape suivante.

2 Rétro-Ingénierie d'un binaire ARM 64 Bits

L'étape commence avec le fichier badbios.bin récupéré lors de l'étape précédente :

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

Listing 3: Fichier de la seconde étape

2.1 Outils

Cette architecture très peu répandue actuellement nécessite des outils particuliers. Pour l'émulation du processeur ARM-64 Bits, le choix s'est porté vers qemu-aarch64. Le désassemblage a été fait en utilisant IDA Pro. Enfin, une version cross-compilée de gdb a été utilisée pour déboguer le programme et se brancher sur le stub gdb fourni par qemu avec l'option -g.

2.2 Obfuscation

Pour commencer cette étape, la première chose à faire est d'exécuter le programme badbios.bin. Celui-ci demande de rentrer une clef sur stdin. Après plusieurs essais infructueux retournant le message d'erreur "Wrong key format", il apparaît que la clef doit faire 16 caractères et appartenir au charset hexadécimal.

```
qemu-aarch64 bad_bios.bin
:: Please enter the decryption key: AAAAAAAAAAAAAAAA
:: Trying to decrypt payload...
   Invalid padding.
```

Listing 4: Trâce d'exécution de badbios.bin

La recherche des strings affichées sur le binaire original ne donne aucun résultat. Le binaire a donc été obfusqué.

```
$ strings bad_bios.bin | grep Trying
$ strings bad_bios.bin | grep Please
```

Listing 5: Recherche des strings affichées par le binaire

Le code exécuté semble aussi être différent du code présent dans le désassemblage fourni par le logiciel IDA PRO. Une exécution de badbios.bin avec l'option -strace de qemu donne plus d'informations :

```
qemu-aarch64 -strace bad_bios.bin
3555 mmap(0x0000000000400000,12288,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED
    ,0,0) = 0x0000000000400000
3555 mprotect(0x0000000000400000,12288,PROT_EXEC|PROT_READ) = 0
3555 mmap(0x0000000000500000,69632,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED
    ,0,0) = 0x0000000000500000
3555 mprotect(0x0000000000500000,69632,PROT_READ|PROT_WRITE) = 0
3555 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000801000
3555 mmap(NULL,65536,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000802000
3555 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000812000
3555 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000813000
3555 write(1,0x813000,36):: Please enter the decryption key:  = 36
3555 munmap(0x0000004000813000,36) = 0
3555 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000814000
3555 read(0,0x814000,16)AAAAAAAAAAAAAAAA
 = 16
3555 munmap(0x0000004000814000,16) = 0
3555 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000815000
3555 write(1,0x815000,32):: Trying to decrypt payload...
 = 32
3555 munmap(0x0000004000815000,32) = 0
3555 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000816000
3555 write(2,0x816000,20)   Invalid padding.
 = 20
3555 munmap(0x0000004000816000,20) = 0
3555 exit_group(0)
```

Listing 6: Trâce d'exécution de badbios.bin avec l'option -strace

Deux zones mémoires principales semblent être allouées par le premier binaire. Une zone en `0x400000` qui est exécutable et une zone en `0x500000` qui est autorisée en lecture et écriture. Il semble donc que le premier binaire charge en mémoire un second exécutable ainsi que des données. Grâce à gdb, il est possible de mettre un point d'arrêt au moment du syscall "read" de la clef et de copier le contenu de ces zones mémoires. Les zones mémoires sont ensuite chargées dans IDA aux offset correspondant afin de pouvoir continuer la rétro-ingéniérie du programme.

Comme cela sera démontré dans la partie suivante, seul le premier niveau d'obfuscation du binaire a été levé. Cependant il devient alors possible de se concentrer sur le vrai code utile du binaire.

2.3 Code exécuté et données chiffrées

Le code chargé à l'adresse `0x400000` (l'entrypoint est à l'adresse `0x400514`) peut être découpé en plusieurs parties. Tout d'abord, le code effectue toute une partie d'initialisation en enlevant notamment une couche d'obfuscation sur plusieurs parties du code.

2.3.1 Initialisation et désobfuscation

Dans cette phase d'initialisation, plusieurs étapes sont marquantes.

Construction d'une table d'offsets en mémoire La fonction à l'adresse `0x401a08` permet, à travers son code obfusqué d'aller écrire dans une nouvelle zone mémoire précédemment allouée, une table d'offset. Le code obfusqué permettant d'écrire un offset se présente sous la forme suivante :

```
ADRP            X1, #loc_40064C@PAGE
ADD             X1, X1, #loc_40064C@PAGEOFF ; Rd = Op1 + Op2
SUB             X1, X1, #0x20 ; Rd = Op1 - Op2
SUB             X1, X1, #0x30 ; Rd = Op1 - Op2
SUB             X0, X0, #0x9A4 ; Rd = Op1 - Op2
ADD             X0, X0, #0xAF4 ; Rd = Op1 + Op2
ADD             X0, X0, #0x38 ; Rd = Op1 + Op2
ADD             X0, X0, #0xD0 ; Rd = Op1 + Op2
STR             X1, [X0,#0x1E8] ; Store to Memory
```

Listing 7: Code ARM obfusqué permettant d'écrire un offset dans la table d'offset

De la même manière, toute une table va être construite. Chaque offset écrit dans la table correspond à une procédure disponible dans le code. Plus tard, il apparaîtra que chacun de ces offsets représente une instruction d'une machine virtuelle.

```
(gdb) x/40gx 0x4000801360
0x4000801360: 0x0000000000400d9c  0x0000000000400dac
0x4000801370: 0x0000000000401580  0x0000000000401634
0x4000801380: 0x00000000004016e4  0x0000000000401030
0x4000801390: 0x00000000004010ec  0x00000000004011b4
0x40008013a0: 0x0000000000401794  0x0000000000400d58
0x40008013b0: 0x0000000000400c90  0x0000000000400c20
0x40008013c0: 0x0000000000400bd0  0x0000000000400b78
0x40008013d0: 0x0000000000400b04  0x0000000000400a8c
0x40008013e0: 0x0000000000400a08  0x0000000000400978
0x40008013f0: 0x0000000000400918  0x00000000004008c4
0x4000801400: 0x0000000000400864  0x00000000004007ec
0x4000801410: 0x0000000000400d24  0x0000000000400ce0
0x4000801420: 0x0000000000401970  0x00000000004018d0
0x4000801430: 0x000000000040187c  0x00000000004005f4
0x4000801440: 0x00000000004005fc  0x0000000000401490
0x4000801450: 0x000000000040077c  0x0000000000000000
```

Listing 8: Table d'offsets générée

Reconstruction d'une chaîne de caractères La fonction à l'adresse `0x400408` permet, à travers son code obfusqué de générer une chaîne de caractères marquante. En effet elle reconstruit la chaîne : "expand 16-byte k" qui est une constante de l'algorithme de chiffrement par flux Salsa20 [1]. Cette chaîne donne donc des indices sur un des moyens d'obfuscation utilisé. Il sera vu plus tard, comment cet algorithme a été utilisé pour cacher des données.

```
expand_16_byte_string_generation
LDR             W2, [X1] ; Load from Memory
MOV             W5, #'xe' ; Rd = Op2
STR             W2, [X0,#0x10] ; Store to Memory
LDR             W2, [X1,#4] ; Load from Memory
MOV             W4, #'dn' ; Rd = Op2
STR             W2, [X0,#0x14] ; Store to Memory
LDR             W2, [X1,#8] ; Load from Memory
MOV             W3, #'-6' ; Rd = Op2
STR             W2, [X0,#0x18] ; Store to Memory
LDR             W2, [X1,#0xC] ; Load from Memory
MOVK            W5, #'ap',LSL#16 ; Move with Keep
STR             W2, [X0,#0x1C] ; Store to Memory
LDR             W6, [X1] ; Load from Memory
MOV             W2, #'et' ; Rd = Op2
STR             W6, [X0,#0x20] ; Store to Memory
LDR             W6, [X1,#4] ; Load from Memory
MOVK            W4, #'1 ',LSL#16 ; Move with Keep
STR             W6, [X0,#0x24] ; Store to Memory
LDR             W6, [X1,#8] ; Load from Memory
MOVK            W3, #'yb',LSL#16 ; Move with Keep
STR             W6, [X0,#0x28] ; Store to Memory
LDR             W1, [X1,#0xC] ; Load from Memory
MOVK            W2, #'k ',LSL#16 ; Move with Keep
STR             W1, [X0,#0x2C] ; Store to Memory
STR             W5, [X0] ; Store to Memory
STR             W4, [X0,#4] ; Store to Memory
STR             W3, [X0,#8] ; Store to Memory
STR             W2, [X0,#0xC] ; Store to Memory
RET                     ; Return from Subroutine
```

Listing 9: Code ARM obfusqué permettant de générer une chaîne de caractères

Copie de la zone mémoire `0x500000` en `0x4000802000` Le code en `0x402EA0` copie les données mémoires de l'adresse `0x500000` vers l'adresse `0x4000802000`.

2.3.2 Utilisation de l'algorithme chacha pour déchiffrer les données en mémoire

Après plusieurs phases d'initialisation plusieurs zones mémoires ont été initialisées. Il convient de s'intéresser à l'algorithme de chiffrement par flux Salsa qui semble être présent. La fonction correspondante dans le code assembleur se situe en `0x400498`. Après une étude poussé du code assembleur de cette fonction, il apparaît qu'il ne s'agit pas de l'algorithme Salsa20 mais d'une variante appelée Chacha8 [2]. Les différences se situent surtout sur la façon dont sont utilisés la clef de chiffrement, les constantes d'initialisation et le flux à chiffrer dans la matrice de l'algorithme.

Ici, la rétro-ingénierie du code a donc mis en évidence l'utilisation de l'algorithme Chacha8 avec une clef fixée à `0x0badb1050badb1050badb1050badb1050badb1050badb1050badb1050badb105`.

Des implémentations de cet algorithme existent et permettent de valider rapidement les conclusions précédentes en tentant de déchiffrer le bloc de données présent en 0x500000 qui semble être chiffré au vu de son entropie.

L'utilisation de la commande `strings` sur le bloc déchiffré confirme les hypothèses formulées précédemment en révélant des chaînes de caractères présentes dans l'affichage du binaire badbios.bin

```
$ strings -n 10 output
:: Please enter the decryption key:
:: Trying to decrypt payload...
   Wrong key format.
   Invalid padding.
```

1. http ://fr.wikipedia.org/wiki/Salsa20
2. http ://cr.yp.to/streamciphers/timings/estreambench/submissions/salsa20/chacha8/regs/chacha.c

```
   Cannot open file payload.bin.
:: Decrypted payload written to payload.bin.
payload.bin
XXXXXXXXXXXXXXXX
```

Listing 10: Chaînes de caractères présente dans le bloc de données déchiffrées avec l'algorithme Chacha8

2.4 Machine virtuelle

Après la phase de déchiffrement de la mémoire chiffrée, il est intéressant d'analyser le code assembleur exécuté suite à la phase d'initialisation. Ce code contient notamment une boucle qui est appelé un très grand nombre de fois qui va lire une valeur dans la mémoire déchiffrée et la transforme en offset dans la table d'offset mentionnée précédemment. Le processeur saute ensuite sur l'offset sélectionné. Ce fonctionnement laisse penser qu'une machine virtuelle a été implémentée dans le programme ARM et que le code de cette machine virtuelle est écrit dans la mémoire chiffrée avec Chacha.

La première étape pour valider cette hypothèse est de s'intéresser au code présent aux différents offset de la table d'offset.

2.4.1 Identification des différentes instructions de la machine virtuelle

Chaque instruction de la machine virtuelle présente elle-même du code obfusqué afin de compliquer le fonctionnement. Cependant, il est possible d'identifier un fonctionnement général :

Les instructions sont sur 16 ou 32 bits. Le code de l'instruction est sur les 8 bits de poids faible alors que les données sont sur les 8 ou 24 bits de poids forts.

L'exemple de la compréhension de l'instruction ADD (adresse `0x400918`) sera détaillé ci dessous. Les autres instructions ont été comprises par le même processus.

```
1  seg003:0000000000400918 STP X29, X30, [SP,#-0x30+arg_0]! // Prologue
2  seg003:000000000040091C STP X21, X22, [SP,#arg_20] // Prologue
3  seg003:0000000000400920 MOV X29, SP // Prologue
4  seg003:0000000000400924 MOV W21, W1 // Prologue
5  seg003:0000000000400928 STP X19, X20, [SP,#arg_10] // Prologue
6  seg003:000000000040092C UBFM X20, X21, #8, #0xB // recuperation de l'operande 1 instr[8:12] ==
       numero de registre
7  seg003:0000000000400930
8  seg003:0000000000400930 loc_400930                              ; DATA XREF: init_offset_table
       +37C
9  seg003:0000000000400930                                         ; init_offset_table+380
10 seg003:0000000000400930 MOV W1, W20
11 seg003:0000000000400934 ORR X19, X0, X0
12 seg003:0000000000400938 BL fetch_word ; Return fetched value into X0
13 seg003:0000000000400938                                         ; return 0x4000812000 + (w1
       -1) << 2
14 // lecture du contenu du numero de registre lu precedemment
15 seg003:000000000040093C MOV W22, W0
16 seg003:0000000000400940 MVN X0, X19
17 seg003:0000000000400944 UBFM X1, X21, #0xC, #0xF // recuperation de l'operande 2 instr[12:16]
       == numero de registre
18 seg003:0000000000400948 MVN X0, X0
19 seg003:000000000040094C BL fetch_word ; Return fetched value into X0
20 seg003:000000000040094C                                         ; return 0x4000812000 + (w1
       -1) << 2
21 // lecture du contenu du numero de registre lu precedemment
22 seg003:0000000000400950 ADD W2, W0, W22 // addition des deux valeurs
23 seg003:0000000000400954 EOR X0, X0, X19
24 seg003:0000000000400958 MOV W1, W20
25 seg003:000000000040095C EOR X19, X19, X0
26 seg003:0000000000400960 EOR X0, X19, X0
27 seg003:0000000000400964 LDP X21, X22, [SP,#arg_20]
28 seg003:0000000000400968 ORR X19, X0, X0
29 seg003:000000000040096C LDP X19, X20, [SP,#arg_10]
30 seg003:0000000000400970 LDP X29, X30, [SP+arg_0],#0x30
31 seg003:0000000000400974 B epilogue
```

Listing 11: Code de l'instruction ADD de la machine virtuelle commenté

Les opérandes passées à l'instruction sont des numéros de registres. Le code récupère ces numéros de registres et va chercher leur valeur en mémoire. Les deux valeurs sont ensuite additionnées. La fonction epilogue ira ensuite écrire la valeur retournée dans W2 dans le registre mentionné dans W1. Ainsi l'instruction ci dessus représente une addition entre la valeur de deux registres dans un autre registre.

Le tableau ci-dessous décrit le fonctionnement de chaque instruction :

```
Instructions sur 32 bits
data 0  : r[data[0:3]] = 0
data 1  : r[data[0:3]] = data[4:7] | data[20:23]
data 2  : LOADW r[data[0:3]] <- [r[data[4:7]]+data[8:11]]
data 4  : LOADB r[data[0:3]] <- [r[data[4:7]]+data[8:11]]
data 7  : STORE r[data[0:3]] -> [r[data[4:7]]+data[8:11]]
data 8  :
    if data[d:f] == 0
        jmp data[0x10:]
    else if if data[d:f] == 3
        jnz data[9:12] data[0x10:]
    else
        jcc data[9:12] cond_data[d:f] data[0x10:]

Instructions sur 16 bits
data a  : r[data[0:3]] = r[data[0:3]] ^ r[data[4:7]]
data b  : r[data[0:3]] = r[data[0:3]] | r[data[4:7]]
data c  : r[data[0:3]] = r[data[0:3]] & r[data[4:7]]
data d  : r[data[0:3]] = r[data[0:3]] << r[data[4:7]]
data e  : r[data[0:3]] = r[data[0:3]] >> r[data[4:7]]
data 12 : r[data[0:3]] = r[data[0:3]] + r[data[4:7]]
data 13 : r[data[0:3]] = r[data[0:3]] - r[data[4:7]]
data 16 : r[data[0:3]] += 1
data 17 : r[data[0:3]] -= 1
data 1c : EXIT
data 1d : SYSCALL
    if data[0:3] == 0
        SYS_OPEN
    else if data[0:3] == 1
        SYS_READ
    else if data[0:3] == 2
        SYS_WRITE
    else if data[0:3] == 3
        SYS_CLOSE
data 1e : r[data[0:3]] = parity_of_the_number_of_bit_set_to_1(r[data[0:3]])
```

Listing 12: Décodage des instructions de la machine virtuelle

2.4.2 Désassemblage du programme de la machine virtuelle

Maintenant que toutes les instructions sont décodées, il est possible d'écrire un désassembleur (voir Annexes : listing 40) et de désassembler le programme de la machine virtuelle disponible dans le fichier de la mémoire déchiffré à l'offset 0x40 :

```
00000100        0040:   MOV R1, #0
00002101        0044:   MOV R1, 2
00000200        0048:   MOV R2, #0
00001201        004c:   MOV R2, 1
00000300        0050:   MOV R3, #0
0032e301        0054:   MOV R3, 32e         -- please enter key
00000400        0058:   MOV R4, #0
00024401        005c:   MOV R4, 24
    001d        0060:   SYS write
00000100        0062:   MOV R1, #0
00001101        0066:   MOV R1, 1
    220a        006a:   XOR R2 = R2 ^ R2
00000300        006c:   MOV R3, #0
003fc301        0070:   MOV R3, 3fc         -- buffer XXXXXX
00000400        0074:   MOV R4, #0
00010401        0078:   MOV R4, 10
    001d        007c:   SYS read
00000502        007e:   LOADW R5 <- [R0+0]
```

```
00000300        0082:   MOV R3, #0
00010301        0086:   MOV R3, 10
    3513        008a:   SUB R5 = R5 - R3
02b46a08        008c:   JNZ R5 2b4
----------------
00000f00        0090:   MOV R15, #0
00010f01        0094:   MOV R15, 10
00000e00        0098:   MOV R14, #0
003fce01        009c:   MOV R14, 3fc        -- buffer XXXXXX
00000d00        00a0:   MOV R13, #0
00326d01        00a4:   MOV R13, 326        -- buffer a zero
    0d17        00a8:   DEC R13
00000200        00aa:   MOV R2, #0
00030201        00ae:   MOV R2, 30
00000300        00b2:   MOV R3, #0
00039301        00b6:   MOV R3, 39
00000400        00ba:   MOV R4, #0
00041401        00be:   MOV R4, 41
00000500        00c2:   MOV R5, #0
00046501        00c6:   MOV R5, 46
0000ec04        00ca:   LOADB R12 <- [R14+0]
002c0102        00ce:   LOADW R1 <- [R0+2c]
    2113        00d2:   SUB R1 = R1 - R2
02b48208        00d4:   Jcc R1 cond_4 2b4
----------------
002c0102        00d8:   LOADW R1 <- [R0+2c]
    3113        00dc:   SUB R1 = R1 - R3
0106c208        00de:   Jcc R1 cond_6 106
----------------
002c0102        00e2:   LOADW R1 <- [R0+2c]
    4113        00e6:   SUB R1 = R1 - R4
02b48208        00e8:   Jcc R1 cond_4 2b4
----------------
002c0102        00ec:   LOADW R1 <- [R0+2c]
    5113        00f0:   SUB R1 = R1 - R5
02b4a208        00f2:   Jcc R1 cond_5 2b4
----------------
    4c13        00f6:   SUB R12 = R12 - R4
00000100        00f8:   MOV R1, #0
0000a101        00fc:   MOV R1, a
    1c12        0100:   ADD R12 = R12 + R1
01080008        0102:   JUMP 108
----------------
    2c13        0106:   SUB R12 = R12 - R2
00000700        0108:   MOV R7, #0
00010701        010c:   MOV R7, 10
    f713        0110:   SUB R7 = R7 - R15
00000100        0112:   MOV R1, #0
00001101        0116:   MOV R1, 1
    710c        011a:   AND R1 = R1 & R7
012c6208        011c:   JNZ R1 12c
----------------
00000700        0120:   MOV R7, #0
00004701        0124:   MOV R7, 4
    7c0d        0128:   LSR R12 = R12 << R7
    0d16        012a:   INC R13
0000d104        012c:   LOADB R1 <- [R13+0]
    c10b        0130:   OR  R1 = R1 | R12
0000d107        0132:   STORE R1 -> [R13+0]
    0e16        0136:   INC R14
    0f17        0138:   DEC R15
00ca7e08        013a:   JNZ R15 ca
----------------
00000100        013e:   MOV R1, #0
00002101        0142:   MOV R1, 2
00000200        0146:   MOV R2, #0
00001201        014a:   MOV R2, 1
00000300        014e:   MOV R3, #0
00354301        0152:   MOV R3, 354        -- trying to decrypt payload
00000400        0156:   MOV R4, #0
```

```
00020401        015a:   MOV R4, 20
    001d        015e:   SYS write
00000100        0160:   MOV R1, #0
00326101        0164:   MOV R1, 326        -- buffer a zero
00001a02        0168:   LOADW R10 <- [R1+0]
00041b02        016c:   LOADW R11 <- [R1+4]
    110a        0170:   XOR R1 = R1 ^ R1
00000200        0172:   MOV R2, #0
08000201        0176:   MOV R2, 8000
00000300        017a:   MOV R3, #0
00008301        017e:   MOV R3, 8
    440a        0182:   XOR R4 = R4 ^ R4
0b000c00        0184:   MOV R12, #0
00000c01        0188:   MOV R12, 0
00000d00        018c:   MOV R13, #0
00001d01        0190:   MOV R13, 1
00240802       >0194:   LOADW R8 <- [R0+24]
00280902        0198:   LOADW R9 <- [R0+28]
    c80c        019c:   AND R8 = R8 & R12
    d90c        019e:   AND R9 = R9 & R13
    980a        01a0:   XOR R8 = R8 ^ R9
    891e        01a2:   R9 = parity_of_the_number_of_bit_set_to_1(R9)
    790d        01c2:   LSR R9 = R9 << R7
    9a0b        01c4:   OR  R10 = R10 | R9
    0317        01c6:   DEC R3
00280702        01c8:   LOADW R7 <- [R0+28]
    870c        01cc:   AND R7 = R7 & R8
    370d        01ce:   LSR R7 = R7 << R3
    740b        01d0:   OR  R4 = R4 | R7
01f66608        01d2:   JNZ R3 1f6
----------------
00000700        01d6:   MOV R7, #0
08000701        01da:   MOV R7, 8000
    1712        01de:   ADD R7 = R7 + R1
00007804        01e0:   LOADB R8 <- [R7+0]
    480a        01e4:   XOR R8 = R8 ^ R4
00007807        01e6:   STORE R8 -> [R7+0]
00000300        01ea:   MOV R3, #0
00008301        01ee:   MOV R3, 8
    0116        01f2:   INC R1
    440a        01f4:   XOR R4 = R4 ^ R4
00000800        01f6:   MOV R8, #0
02000801        01fa:   MOV R8, 2000
    1813        01fe:   SUB R8 = R8 - R1
0194b008        0200:   Jcc R8 cond_5 194
----------------
00000d00        0204:   MOV R13, #0
08000d01        0208:   MOV R13, 8000
00000c00        020c:   MOV R12, #0
02000c01        0210:   MOV R12, 2000
00000b00        0214:   MOV R11, #0
00080b01        0218:   MOV R11, 80
    aa0a        021c:   XOR R10 = R10 ^ R10
00000900        021e:   MOV R9, #0
00008901        0222:   MOV R9, 8
    0a16        0226:   INC R10
    0c17        0228:   DEC R12
02dad808        022a:   Jcc R12 cond_6 2da - invalid parity
----------------
00300a02        022e:   LOADW R10 <- [R0+30]
    ca12        0232:   ADD R10 = R10 + R12
0000a104        0234:   LOADB R1 <- [R10+0]
```

```
02264208        0238:   Jcc R1 cond_2 226
----------------
    b113        023c:   SUB R1 = R1 - R11
02da6208        023e:   JNZ R1 2da
----------------
    9a13        0242:   SUB R10 = R10 - R9
02dad408        0244:   Jcc R10 cond_6 2da
----------------
    010c        0248:   AND R1 = R1 & R0
00000200        024a:   MOV R2, #0
003f0201        024e:   MOV R2, 3f0        -- payload.bin
00000300        0252:   MOV R3, #0
00241301        0256:   MOV R3, 241
00000400        025a:   MOV R4, #0
001b6401        025e:   MOV R4, 1b6
    001d        0262:   SYS open
02b28208        0264:   Jcc R1 cond_4 2b2
----------------
00000202        0268:   LOADW R2 <- [R0+0]
00000100        026c:   MOV R1, #0
00002101        0270:   MOV R1, 2
00000300        0274:   MOV R3, #0
08000301        0278:   MOV R3, 8000
002c0402        027c:   LOADW R4 <- [R0+2c]
    001d        0280:   SYS write
00000100        0282:   MOV R1, #0
00003101        0286:   MOV R1, 3
    001d        028a:   SYS close
00000100        028c:   MOV R1, #0
00002101        0290:   MOV R1, 2
00000200        0294:   MOV R2, #0
00001201        0298:   MOV R2, 1
00000300        029c:   MOV R3, #0
003c2301        02a0:   MOV R3, 3c2        -- decrypted payload written
00000400        02a4:   MOV R4, #0
0002d401        02a8:   MOV R4, 2d
    001d        02ac:   SYS write
02b20008        02ae:   JUMP 2b2
----------------
    001c        02b2:   EXIT --------------------------------
[0, 2, 1, 962, 45, 70, 0, 32768, 8191, 8, 8231, 128, 8191, 32768, 1021, 15, 692]
00000100        02b4:   MOV R1, #0
00002101        02b8:   MOV R1, 2
00000200        02bc:   MOV R2, #0
00002201        02c0:   MOV R2, 2
00000300        02c4:   MOV R3, #0
00374301        02c8:   MOV R3, 374        -- invalid format
00000400        02cc:   MOV R4, #0
00015401        02d0:   MOV R4, 15
    001d        02d4:   SYS write
02b20008        02d6:   JUMP 2b2
----------------
00000100        02da:   MOV R1, #0
00002101        02de:   MOV R1, 2
00000200        02e2:   MOV R2, #0
00002201        02e6:   MOV R2, 2
00000300        02ea:   MOV R3, #0
0038a301        02ee:   MOV R3, 38a        -- invalid padding
00000400        02f2:   MOV R4, #0
00014401        02f6:   MOV R4, 14
    001d        02fa:   SYS write
02b20008        02fc:   JUMP 2b2
----------------
00000100        0300:   MOV R1, #0
00002101        0304:   MOV R1, 2
00000200        0308:   MOV R2, #0
00002201        030c:   MOV R2, 2
00000300        0310:   MOV R3, #0
003a0301        0314:   MOV R3, 3a0        -- cannot open file
00000400        0318:   MOV R4, #0
```

```
229 00021401         031c:   MOV R4, 21
230     001d         0320:   SYS write
231 02b20008         0322:   JUMP 2b2
232 ----------------
233 00000000         0326:   MOV R0, #0
234 00000000         032a:   MOV R0, #0
```

Listing 13: Désassemblage du programme écrit pour la machine virtuelle

3 Rétro-ingénierie d'un algorithme de déchiffrement

Le binaire à attaquer est un logiciel de déchiffrement de données, demandant une clef sur 8 octets.

3.1 Plusieurs angles d'attaque

Afin de ne pas rater une clef évidente pendant l'analyse plus fine de l'algorithme, deux attaques sont menées en parallèle :

— une approche par force brute en tâche de fond,
— une réflexion plus fine sur le fonctionnement de l'algorithme de chiffrement afin d'identifier un éventuel raccourci.

3.2 Approche par force brute

En tâche de fond, une attaque par force brute a été mise en place. Le programme badbios.bin réalise une vérification de "padding" sur le dernier octet des données déchiffrées, avec des clefs aléatoires cette vérification fonctionne dans environ 0.5% des cas.

Cette attaque par force brute a très peu de chance d'aboutir, mais a permis de générer un nombre important de fichiers de charge utile "payload.bin", qui pourront être analysés par la suite pour mettre en place une méthode plus performante.

3.2.1 Analyse statistique sur les charges utiles générées

L'analyse des charges utiles générées avec des clefs proches permet de constater des similitudes importantes sur les 8 premiers octets, une analyse statistique est donc réalisée pour identifier les relations entre la clef et chacun des 8 premiers octets.

```
$ ./analyse_key.py keys_first_byte_0x8b
1EE1AE90A333C58A 0001111011100001101011101001000010100011001100111100010110001010
2185DF4DA3B1F293 0010000110000101110111111010011011010001110110001111110010100100 11
2D040812A3FBF18F 0010110100000100000010000000100101010001111111011111100011000 1111
335A31EEA28F1C20 0011001101011010001100011110111010100010100011110001110000100000
378C18A0A283538A 0011011110001100000110001010000010100010100000110101001110001010
3BD9CBA4A27B6BEF 0011101111011001110010111010010010100010011110110110101111101111
3FC9351DA35DF7DC 0011111111001001001101010001110110100011010111011111011111011100
40D7A6C5A2AF55E3 0100000011010111101001101100010110100010101011110101010111100011
44174D06A3A37526 0100010000010111010011010000011010100011101000110111010100100110
591F7D23A3930B11 0101100100011111011111010010001110100011100100110000101100010001
[...]
```

Listing 14: Recherche de similitudes entre les clefs ayants générés un payload débutant par l'octet 0x8b

Pour chacun des 8 premiers octets du fichier "payload.bin", les bits déterminants de la clef sont identifiés.

3.2.2 Générateur de clefs pour une en-tête de charge utile choisie

Avec les résultats de l'analyse statistique, des dictionnaires sont constitués, contenant la relation entre les octets de la charge utile générée et les bits marquants de la clef.

Ces dictionnaires sont ensuite utilisés pour générer des clefs pour une en-tête de charge utile choisie.

```
$ ./create_key_for_header.py 7f454c4602010100
DBADB004FD3F4994
```

Listing 15: Génération d'une clef pour une en-tête ELF 64bits

Quelques en-têtes courantes sont testées avec cette méthode sans résultats. Pour les en-têtes de moins de 8 octets, une attaque par force brute est mise en place, également sans résultat.

3.3 Recherche de vulnérabilités de conception

Pendant que les processeurs a disposition tentent de trouver le clef par hasard, il est nécessaire d'étudier le fonctionnement du programme attaqué pour identifier soit une solution simple de déchiffrement sans clef, soit un test d'arrêt moins coûteux pour l'identification d'au moins quelques bits de la clef.

Le code assembleur de la machine virtuelle est ramené au code C du listing 16

```
void vm_decrypt(uint8_t *payload, int32_t hi, uint32_t lo) {
  uint32_t r10 = 0;
  uint32_t r11 = 0;

  r10 = hi;
  r11 = lo;
  uint32_t r2 = 0x8000;
  uint32_t r3 = 8;
  uint32_t r4 = 0;
  uint32_t r12 = 0xb0000000UL;
  uint32_t r13 = 1;

  //-- tempo vars
  uint32_t r8 = 0;
  uint32_t r9 = 0;
  uint32_t r7 = 0;
  uint32_t r6 = 0;
  uint32_t r1 = 0;
  //--

  do {
    // 0194
    r8 = r10;
    r9 = r11;
    r8 &= r12;
    r9 &= r13;
    r8 ^= r9;

    unsigned int v = r8; // DW
    v ^= v >> 1;
    v ^= v >> 2;
    v = (v & 0x11111111U) * 0x11111111U;
    r9= (v >> 28) & 1;

    r8 = 1;
    r7 = 0x1f;
    // 01b4
    r6 = r10;
    r6 &= r8;
    r6 <<= r7;
    r11 >>= r8;
    r11 |= r6;
    r10 >>= r8;
    r9 <<= r7;
    r10 |= r9;
    r3--;
    // 01c8
    r7 = r11;
    r7 &= r8;
    r7 <<= r3;
    r4 |= r7;

    if (r3 == 0) {
      r7 = 0x8000; // @payload
      r7 += r1;
      r8 = payload[r1];
      r8 ^= r4;
      payload[r1] = r8;
      r3 = 8;
      r1 ++;
      r4 = 0;
    }
```

```
    r8 = 0x2000;
    r8 -= r1;

  } while(r8>0);
}
```

Listing 16: Simulation de la VM

En observant le comportement du programme, il est possible de constater qu'il calcule les valeurs d'un registre à décalage à rétroaction linéaire (*Linear Feedback Shift-Register* ou *LFSR*). Il utilise la sortie du LFSR comme un ver chiffrant combiné octet par octet par l'opération XOR avec le message pour le chiffrer ou le déchiffrer.

En effet, le calcul de parité n'est au final qu'un xor entre bits masqués qui est ensuite ramené à gauche. Le programme peut donc être résumé à un XOR octet par octet avec un *LFSR* ayant pour état interne la clef sur 64 bits et comme polynôme `0xb0000001UL`.

En pseudo-code très résumé, cela donne :

```
    key1 = 0xXXXXXXXX
    key2 = 0xXXXXXXXX

    r3 = 8
    i=0
    while(i<0x2000){}
            mask = (key1 ^key2) & 0xb0000001
            parite = par(mask)
            r6 = key1 & 0x1 << 0x1f
            key2 = (key2 >> 1) | r6
            key1 = (key1>>1) | (parite << 0x1f)
            char = char | ((key2 & 0x1) << r3)
            if(r3=0){
                    out[i] = in[i] ^ char
                    r3 = 8
                    i++
                    r4 = 0
            }
    }
```

3.3.1 Fonctionnement d'un LFSR (Registre à décalage à rétroaction linéaire)

Le LFSR peut être schématiquement représenté comme dans la figure 1.

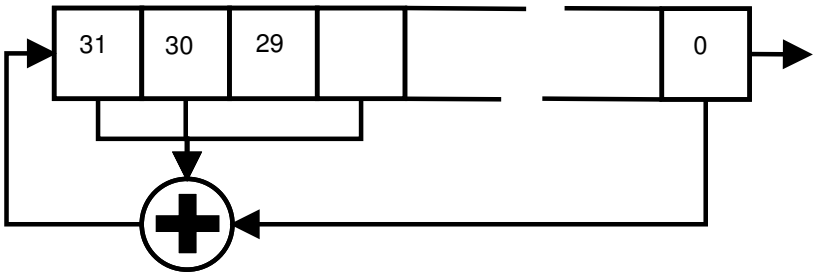


FIGURE 1: Représentation schématique du LFSR

Il s'agit d'un registre à décalage dont l'entrée à chaque tour est un XOR d'une combinaison de bits de l'état interne. La séquence produite par ce LFSR dépend uniquement du polynôme et de l'état initial. Ici, l'état initial est la clef (que l'on ne connait pas) et le polynôme est déduit du code.

Connaissant le polynôme et l'état interne à un instant donné, il est possible de calculer le bit produit suivant. Du fait de l'utilisation d'opération inversibles, il est également possible de calculer l'état interne précédent, le bit ayant été sorti à droite pouvant être calculé par un XOR avec les 3 autres à gauche de l'état interne

3.3.2 Hypothèses d'attaque et inversion

Le test d'arrêt utilisé dans la VM vérifie juste si sur le dernier octet déchiffré le bit suivant du LFSR est à 1. Il est également possible de voir que le clair produit en entrant une clef erronée mais valide produit un binaire bien plus petit que les 8 Kio de chiffré. Partant de là, l'hypothèse que l'état interne du LFSR à la fin du déchiffrement correspond à un XOR avec la fin du chiffré pour obtenir uniquement des zéros entre les octets 1544 à 8191 est prise.

Fort de ces hypothèses, il est possible de calculer l'état interne du LFSR 8192×8 bits avant. La valeur de l'état interne est notée et l'algorithme de la VM est redéroulé. Le résultat obtenu contient effectivement uniquement des zéros après l'octet 1544, le début est un fichier ZIP.

La clef correspondant à l'état interne initial du LFSR pour obtenir ce fichier est `0BADB10515DEAD11` (*Badbios is dead!!*). Le code python du calcul de l'état initial du LFSR se trouve en annexe dans listing 41.

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

L'étude de ce fichier est l'objet de l'étape suivante.

4 Rétro-Ingénierie d'un micro-contrôleur inconnu

4.1 Découverte des fichiers de l'étape

L'étape commence avec l'archive zip récupérée lors de l'étape précédente

```
$ file payload.bin
payload.bin: Zip archive data, at least v2.0 to extract
$ unzip payload.bin
Archive:  payload.bin
  inflating: mcu/upload.py
  inflating: mcu/fw.hex
$ file mcu/*
mcu/fw.hex:     ASCII text
mcu/upload.py: Python script, ASCII text executable
```

Listing 17: Fichiers de la dernière étape

Le fichier `fw.hex` est un firmware au format Intel hex file [3]. Le fichier `upload.py` est un script founi qui permet d'envoyer le firmware vers un serveur distant et d'afficher les données reçues en retour. La première étape consiste donc à analyser le fichier `upload.py` afin d'en apprendre d'avantage :

```
#!/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
```

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

```
47          if not data:
48              break
49          resp += data
50      else:
51          break
52
53  print(resp.decode("utf-8"))
54  s.close()
```

Listing 18: upload.py

Ce fichier apporte plusieurs informations. Tout d'abord, ce script est utilisé pour envoyer le firmware associé vers le serveur distant. De plus, dans les commentaires du fichier, des informations sont fournies sur le micro-contrôleur. Le micro-contrôleur serait d'architecture inconnue, ce qui laisse supposer qu'il ne sera pas possible de désassembler le code du firmware avec un outil connu. Le plan d'adressage mémoire est aussi fourni : il existe deux zones mémoires avec des accès différents : user et privileged. Une des zones mémoires de la zone privilégiées est appelée : Secret Memory Area. Il semble donc que le but de cette étape est d'envoyer un Firmware modifié qui va être capable de lire cette zone mémoire. Enfin, au vu de l'adressage mémoire, le micro-contrôleur présente une architecture 16-bits.

4.2 Identification des instructions du micro-contrôleur

Afin d'essayer de comprendre le fonctionnement du micro-contrôleur, la première étape consiste à envoyer le Firmware fourni et à analyser la réponse, puis à modifier le Firmware afin d'étudier les différences de comportement.

4.2.1 Trace d'exécution avec le Firmware fourni

L'envoi du Firmware original donne la trace suivante :

```
$ 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.
```

Listing 19: trace provoquée par l'envoie de fw.hex avec upload.py

Dans la trace d'exécution, il est possible de retrouver des chaînes de caractères présentes dans le Firmware.

```
$ strings -n 6 fw.hex
YeahRiscIsGood!
Firmware v1.33.7 starting.
Halting.
```

Listing 20: chaînes de caractères du fichier `fw.hex`

Toutefois certaines chaînes ne se retrouvent pas, il est possible qu'elles soient générées par le Rom Code (Kernel) qui est exécuté sur le micro-contrôleur.

Une des chaînes semble contenue dans le firmware mais sous une autre forme, en positionnant cette chaîne avec uniquement des octets null (0x00), une clef pour une opération XOR est obtenue.

```
-------------------------------------------
----- Microcontroller firmware uploader -----
-------------------------------------------
()
:: Serial port connected.
:: Uploading firmware...
done.
```

```
()

53 79 73 74 65 6d 20 72  65 73 65 74 2e 0a 46 69      |System.reset..Fi|
72 6d 77 61 72 65 20 76  31 2e 33 33 2e 37 20 73      |rmware.v1.33.7.s|
74 61 72 74 69 6e 67 2e  0a d1 53 35 0c db 78 d2      |tarting...S5..x.|
70 57 94 bb a5 68 8d e6  6a 2e 8d d1 f4 64 02 c8      |pW...h..j....d..|
4e 82 88 e0 b6 d8 b2 22  f2 5c 85 ac c6 04 6c 52      |N........"...\....lR|
```

Listing 21: Recherche de la clef XOR pour la chaîne "Execution completed in XXXX CPU cycles."

```
key = "\xd1\x53\x35\x0c\xdb\x78\xd2\x70\x57\x94\xbb\xa5\x68\x8d\xe6\x6a\x2e\x8d\xd1\xf4\x64\
    x02\xc8\x4e\x82\x88\xe0\xb6\xd8\xb2\x22\xf2\x5c\x85\xac\xc6\x04\x6c\x52"
msg = "\x94\x2b\x50\x6f\xae\x0c\xbb\x1f\x39\xb4\xd8\xca\x05\xfd\x8a\x0f\x5a\xe8\xb5\xd4\x0d\
    x6c\xe8\x6a\xa6\xac\xc4\x92\xf8\xf1\x72\xa7\x7c\xe6\xd5\xa5\x68\x09\x21"


c = "".join([chr(ord(i)^ord(j)) for i,j in zip(key,msg)])
print c
```

Listing 22: unxor.py

```
$ python unxor.py
Execution completed in $$$$$ CPU cycles
```

Listing 23: Décodage de la chaîne recherchée

4.2.2 Exécution d'un Firmware modifié

Pour obtenir des informations sur le code exécuté, le firmware est modifié afin de remplacer le premier octet par un octet null (00). L'envoie de ce Firmware modifié donne le résultat suivant :

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

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

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

Listing 24: Trace d'exécution avec le premier octet du firmware nul

Suite à l'envoie d'une mauvaise instruction (l'instruction 00 renvoie le message "Invalid instruction"), le micro-controleur fourni une trace de débug avec la valeur de ses registres, les flags S et Z et un mode valant "user" dans le cas de test. Grâce a cette fonctionnalité offerte par le micro-contrôleur il va être possible d'essayer de comprendre le rôle de chaque instruction en étudiant les valeurs des registres.

Le test suivant met en évidence la façon donc la valeur des registres peut permettre de comprendre le role de chaque instruction. Le firmware fourni est remodifié avec le 6ème octet à 00 (les instructions sont sur 16 bits, le dump des registres devrait donc être effectué après l'exécution de deux instructions : 21 00 11 1B).

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

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

System reset.
-- Exception occurred at 0004: Invalid instruction.
   r0:0000     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:0004 fault_addr:0000 [S:0 Z:0] Mode:user
CLOSING: Invalid instruction.
```

Listing 25: Trace d'exécution des deux premières instructions du Firmware

La valeur du pc à 4 confirme la bonne exécution des deux premières instructions. La valeur 1B a été affectée au registre 1. Cette valeur était la seconde partie de la seconde instruction. L'instruction 11 semble donc être une instruction permettant d'affecter une valeur à un registre. Afin de vérifier cette hypothèse, plusieurs tests sont réalisés en envoyant au serveur un Firmware contenant une instruction valide suivie de l'instruction 00. Les résultats suivants sont obtenus.

```
11 1C 00 : r0:0000     r1:001C    r2:0000    r3:0000
12 1C 00 : r0:0000     r1:0000    r2:001C    r3:0000
21 1D 00 : r0:0000     r1:1D00    r2:0000    r3:0000
20 1C 00 : r0:1C00     r1:0000    r2:0000    r3:0000
14 1C 00 : r4:001C     r5:0000    r6:0000    r7:0000
```

Listing 26: Identification de l'instruction 11

Grâce à ces tests il est possible de définir deux instructions :

```
1xyy : r[x] = yy
2xyy : r[x] = yy << 2
```

Listing 27: Identification des instructions 1 et 2

4.2.3 Correspondance des instructions du micro-contrôleur

En procédant de la même façon que l'exemple précédent, il est possible d'identifier toutes les instructions disponibles sur le micro-contrôleur. Le tableau de correspondance suivi est obtenu.

```
1 xyy : r[x] = yy
2 xyy : r[x] = yy << 2
3 xyz : r[x] = r[y] ^ r[z]
4 xyz : r[x] = r[y] | r[z]
5 xyz : r[x] = r[y] & r[z]
6 xyz : r[x] = r[y] + r[z]
7 xyz : r[x] = r[y] - r[z]
8 xyz : r[x] = r[y] * r[z]
9 xyz : r[x] = r[y] / r[z]
a xyz : jz  pc + 2 + offset
b xyz : jmp pc + 2 + offset
c xyz : if x < 8  : branch pc + 2 +offset
c x0y : if x >=8   : syscall y
d 00f : ret
d 800 : privileged instr
e xyz : r[x] = [r[y] + r[z]]
f xyz : [r[y] + r[z]] = r[x]

le calcul de l'offset est fait comme suit :

si x % 4 = 0 offset = 0xYZ
si x % 4 = 1 offset = 0x1YZ
si x % 4 = 2 offset = 0xYZ - 512
si x % 4 = 3 offset = 0xYZ - 256
```

Listing 28: Décodage des instructions du micro-contrôleur

4.2.4 Ecriture d'un désassembleur

Le script disass.py a été écrit afin de désassembler le code du Firmware fourni (voir Annexes : listing 42)

4.3 Rétro-Ingéniérie du Firmware fourni et du Rom Code

4.3.1 Rétro-Ingéniérie du Firmware

Le Firmware fourni dans les fichiers de cette étape ne présente pas un grand intérêt en lui même. Il se contente d'afficher des chaînes de caractères ainsi que de récupérer le nombre de cycles CPU nécessaires à l'exécution du programme. Cependant, certaines zones du code sont intéressantes pour la suite, notamment dans l'optique de comprendre le fonctionnement des syscalls. Le code suivant permet l'affichage, par le biais d'un syscall, d'une chaîne de caractères stockée en mémoire à l'adresse 0x18C de longueur 0x1b. Il permet donc de mettre en évidence la correspondance entre le syscall 02 et un write(1,adresse,longueur).

```
0x0 : 2100 : hr1=0x00
0x2 : 111b : lr1=0x1b
0x4 : 2001 : hr0=0x01
0x6 : 108c : lr0=0x8c
0x8 : c0d2 : jmp 0xdc  // Firmware syscall write
...
0xdc : cc02 : syscall// sys.write(1,r0,r1)
0xde : d00x : ret (jmp sur l'adresse dans r[x]
```

Listing 29: Code permettant l'affichage d'une chaîne de caractères stockée en mémoire

De la même manière, le code suivant permet d'identifier la lecture du nombre d'instruction et son écriture à l'adresse spécifiée en RAM dans r0 par le syscall 03

```
0x26 : 2011 : hr0=0x11
0x28 : 1000 : lr0=0x00
0x2a : c0b4 : jmp 0xe0
...
0xe0 : c803 : syscall
0xe2 : d00f : ret
```

Listing 30: Code permettant la lecture du nombre d'instruction et son écrite à l'adresse spécifiée dans r0 par le syscall 03

Le syscall 02 va donc permettre de lire des données en mémoire et de les afficher. Il est alors possible d'écrire un firmware qui tente d'accèder aux zones mémoires privilégiées.

4.3.2 Tentative de lecture des zones mémoires privilégiées

Dans un premier temps, il est intéressant d'essayer de lire directement la zone "Secret memory area". Le Firmware suivant est destiné à essayer de lire cette zone :

```
0x0 : 21f0 : hr1=0x0b
0x2 : 1100 : lr1=0xff // R1 = 0x0BFF
0x4 : 200b : hr0=0xf0
0x6 : 10ff : lr0=0x00 // R0 = 0xF000
0x8 : c802 : syscall // write(1,0xF000,0x0BFF)
```

Listing 31: Tentative de lecture de la zone "Secret memory area" en utilisant le syscall 02

Cependant, cette tentative échoue car le syscall 02 ne semble pas posséder les droits privilégiés permettant d'accèder à cette zone.

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

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

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

Listing 32: Résultat de la tentative de lecture de la zone "Secret memory area"

Cette tentative ayant échoué, il apparaît qu'il est nécessaire d'essayer de lire le ROM Code et de l'analyser afin de trouver une éventuelle vulnérabilité permettant d'accéder à la zone mémoire protégée. Le Firmware suivant permet de récupérer le ROM Code :

```
0x0 : 21f0 : hr1=0x0b
0x2 : 1100 : lr1=0xff // R1 = 0x0BFF
0x4 : 200b : hr0=0xfd
0x6 : 10ff : lr0=0x00 // R0 = 0xFD00
0x8 : c802 : syscall // write(1,0xFD00,0x0BFF)
```

Listing 33: Lecture du ROM Code en utilisant le syscall 02

Cette commande fonctionne et fournit le ROM Code au format binaire, grâce au désassembleur déjà écrit, il est possible d'analyser ce ROM Code. L'analyse sera faite dans la partie suivante.

4.3.3 Rétro-Ingénierie du ROM Code

Le ROM Code a été désassemblé avec l'outil écrit précédement pour donner le listing suivant commenté manuellement :

```
0xfd00 : 5000 : r0=r0&r0
0xfd02 : a06c : jz 0xfd70
0xfd04 : 2100 : hr1=0x00
0xfd06 : 1103 : lr1=0x03
0xfd08 : 7210 : r2=r1-r0
0xfd0a : a812 : jz 0xfd1e // Is syscall number > 3
0xfd0c : 2200 : hr2=0x00
0xfd0e : 1202 : lr2=0x02
0xfd10 : 8102 : r1=r0*r2
0xfd12 : 7112 : r1=r1-r2
0xfd14 : 20f0 : hr0=0xf0
0xfd16 : 1000 : lr0=0x00
0xfd18 : 6001 : r0=r0+r1
0xfd1a : c094 : call 0xfdb0
0xfd1c : d000 : ret


--------------------------------
//printf(Undefined system call)
--------------------------------
0xfd1e : 2100 : hr1=0x00
0xfd20 : 112b : lr1=0x2b
0xfd22 : 20fe : hr0=0xfe
0xfd24 : 105a : lr0=0x5a // fe5a = [ERROR] Undefined system call. CPU halted.
0xfd26 : c0be : call 0xfde6
--------------------------------

--------------------------------
//syscall 01 : boucle infinie
--------------------------------
0xfd28 : 3000 : r0=r0^r0
0xfd2a : 21fc : hr1=0xfc
0xfd2c : 1110 : lr1=0x10
0xfd2e : 2200 : hr2=0x00
0xfd30 : 1201 : lr2=0x01
0xfd32 : f210 : [r1+r0]=r2
0xfd34 : b3f2 : jmp 0xfd28
--------------------------------

--------------------------------
//Syscall  02 : write
--------------------------------
0xfd36 : 20fc : hr0=0xfc
0xfd38 : 1022 : lr0=0x22
0xfd3a : c074 : call 0xfdb0
0xfd3c : 5500 : r5=r0&r0
0xfd3e : 20fc : hr0=0xfc
0xfd40 : 1020 : lr0=0x20
0xfd42 : c06c : call 0xfdb0
0xfd44 : 5155 : r1=r5&r5
0xfd46 : c09e : call 0xfde6
0xfd48 : d800 : privileged instr
--------------------------------
```

```
54
55 -------------------------------
56 // Syscall 03 : read nbr of instr
57 -------------------------------
58 0xfd4a : 20fc : hr0=0xfc
59 0xfd4c : 1020 : lr0=0x20
60 0xfd4e : c060 : call 0xfdb0
61 0xfd50 : 26fc : hr6=0xfc
62 0xfd52 : 1612 : lr6=0x12
63 0xfd54 : 2100 : hr1=0x00
64 0xfd56 : 1101 : lr1=0x01
65 0xfd58 : 3444 : r4=r4^r4
66 0xfd5a : e561 : r5=[r6+r1]
67 0xfd5c : e264 : r2=[r6+r4]
68 0xfd5e : e364 : r3=[r6+r4]
69 0xfd60 : 7332 : r3=r3-r2
70 0xfd62 : a7f6 : jz 0xfd5a
71 0xfd64 : 2301 : hr3=0x01
72 0xfd66 : 1300 : lr3=0x00
73 0xfd68 : 8223 : r2=r2*r3
74 0xfd6a : 4125 : r1=r2|r5
75 0xfd6c : c056 : call 0xfdc4
76 0xfd6e : d800 : privileged instr
77 -------------------------------
78
79 -------------------------------
80 // Init
81 -------------------------------
82 0xfd70 : 2100 : hr1=0x00
83 0xfd72 : 110e : lr1=0x0e
84 0xfd74 : 20fe : hr0=0xfe
85 0xfd76 : 1086 : lr0=0x86
86 0xfd78 : c06c : call 0xfde6  // Printf System reset
87 0xfd7a : 2400 : hr4=0x00
88 0xfd7c : 1402 : lr4=0x02
89 0xfd7e : 21fd : hr1=0xfd
90 0xfd80 : 1128 : lr1=0x28 // r1 = adress of syscall 01
91 0xfd82 : 20f0 : hr0=0xf0
92 0xfd84 : 1000 : lr0=0x00 // r0 = secret_memory_are
93 0xfd86 : c03c : call 0xfdc4 //Secret_memory_area[0] = adress of syscall 01
94 0xfd88 : 6004 : r0=r0+r4
95 0xfd8a : 21fd : hr1=0xfd
96 0xfd8c : 1136 : lr1=0x36 // r1 = adress of syscall 02
97 0xfd8e : c034 : call 0xfdc4 //Secret_memory_area[2] = adress of syscall 02
98 0xfd90 : 6004 : r0=r0+r4
99 0xfd92 : 21fd : hr1=0xfd
100 0xfd94 : 114a : lr1=0x4a // r1 = adress of syscall 03
101 0xfd96 : c02c : call 0xfdc4 //Secret_memory_area[4] = adress of syscall 03
102 0xfd98 : 20fc : hr0=0xfc
103 0xfd9a : 1020 : lr0=0x20 // r0 = addr of registers array
104 0xfd9c : 3111 : r1=r1^r1 // r1 = 0
105 0xfd9e : 2200 : hr2=0x00
106 0xfda0 : 1236 : lr2=0x36 // r2 = 0x36
107 0xfda2 : c032 : call 0xfdd6 // strncpy(hw_registers,0,0x36) --> flush registers
108 0xfda4 : 20fc : hr0=0xfc
109 0xfda6 : 103a : lr0=0x3a // r13
110 0xfda8 : 21ef : hr1=0xef
111 0xfdaa : 11fe : lr1=0xfe
112 0xfdac : c016 : call 0xfdc4 // set r13 to 0xeffe
113 0xfdae : d800 : privileged instr
114 -------------------------------
115
116 -------------------------------
117 // r0 = [r0]
118 -------------------------------
119 0xfdb0 : 2100 : hr1=0x00
120 0xfdb2 : 1101 : lr1=0x01
121 0xfdb4 : 2201 : hr2=0x01
122 0xfdb6 : 1200 : lr2=0x00
123 0xfdb8 : e301 : r3=[r0+r1]
```

```
0xfdba : 7111 : r1=r1-r1
0xfdbc : e401 : r4=[r0+r1]
0xfdbe : 8442 : r4=r4*r2
0xfdc0 : 4034 : r0=r3|r4
0xfdc2 : d00f : ret
--------------------------------

--------------------------------
// [r0] = r1
--------------------------------
0xfdc4 : 2200 : hr2=0x00
0xfdc6 : 1201 : lr2=0x01
0xfdc8 : 2301 : hr3=0x01
0xfdca : 1300 : lr3=0x00
0xfdcc : f102 : [r0+r2]=r1
0xfdce : 7222 : r2=r2-r2
0xfdd0 : 9113 : r1=r1/r3
0xfdd2 : f102 : [r0+r2]=r1
0xfdd4 : d00f : ret
--------------------------------

--------------------------------
//strncpy(r0,r1,r2)
--------------------------------
0xfdd6 : 2300 : hr3=0x00
0xfdd8 : 1301 : lr3=0x01
0xfdda : 5222 : r2=r2&r2
0xfddc : a006 : jz 0xfde4
0xfdde : 7223 : r2=r2-r3
0xfde0 : f102 : [r0+r2]=r1
0xfde2 : b3f2 : jmp 0xfdd6
0xfde4 : d00f : ret
--------------------------------

--------------------------------
// write(1,r0,r1)
--------------------------------
0xfde6 : 5e00 : re=r0&r0
0xfde8 : 2dfc : hrd=0xfc
0xfdea : 1d00 : lrd=0x00
0xfdec : 2cf0 : hrc=0xf0
0xfdee : 1c00 : lrc=0x00
0xfdf0 : 3888 : r8=r8^r8
0xfdf2 : 5988 : r9=r8&r8
0xfdf4 : 2a00 : hra=0x00
0xfdf6 : 1a01 : lra=0x01
0xfdf8 : 3bbb : rb=rb^rb
0xfdfa : 5111 : r1=r1&r1
0xfdfc : a01a : jz 0xfe18
0xfdfe : 69e8 : r9=re+r8
0xfe00 : 799c : r9=r9-rc
0xfe02 : a808 : jz 0xfe0c
0xfe04 : 69e8 : r9=re+r8
0xfe06 : 799d : r9=r9-rd
0xfe08 : ac02 : jz 0xfe0c
0xfe0a : b00e : jmp 0xfe1a
0xfe0c : 3999 : r9=r9^r9
0xfe0e : e9e8 : r9=[re+r8]
0xfe10 : f9db : [rd+rb]=r9
0xfe12 : 688a : r8=r8+ra
0xfe14 : 711a : r1=r1-ra
0xfe16 : b3e2 : jmp 0xfdfa
0xfe18 : d00f : ret
--------------------------------

--------------------------------
// Error message   : '[ERROR] Printing at unallowed address. CPU halted.\n'
--------------------------------
0xfe1a : 2100 : hr1=0x00
0xfe1c : 1133 : lr1=0x33
```

```
0xfe1e : 20fe : hr0=0xfe
0xfe20 : 1026 : lr0=0x26
0xfe22 : c3c2 : call 0xfde6
------------------------------


0xFE26 : '[ERROR] Printing at unallowed address. CPU halted.\n'
0xFE5A : '\x00[ERROR] Undefined system call. CPU halted.\n'
0xFE86 : '\x00System reset.\n'
```

Listing 34: ROM Code commenté

Après avoir analysé le ROM Code, il est possible de constater la présence des éléments suivants :

- La phase d'initialisation (`0xfd70`) :
  - Écriture de la jumptable des syscall
  - Mise à zéro des registres
  - r13 = 0xeffe
- Des primitives :
  - `0xfdb0` : déréférencement de r0 dans r0
  - `0xfdc3` : écriture de r1 à l'adresse r0
  - `0xfdd6` : strncpy(r0,r1,r2)
  - `0xfde6` : write(1,r0,r1)
- Les syscall :
  - `0xfd28` : syscall 01 boucle infinie
  - `0xfd36` : syscall 02 write
  - `0xfd4a` : syscall 03 read number of instructions
- Le traitement du syscall avec l'affichage d'un message d'erreur ou l'appel d'un des syscall (`0xfd00`)

4.3.4 Recherche de vulnérabilités

Comme mentionné précédemment, le syscall 02 ne permet pas de lire la zone mémoire contenant les secrets. Le syscall 01 est une boucle infinie et ne présente donc pas la possibilité d'exploiter une vulnérabilité. Il reste le syscall 03 qui permet de lire le nombre d'instruction exécutées et de l'écrire à une adresse définie.

Le Firmware suivant tente d'exploiter cette vulnérabilité en écrivant le nombre d'instructions dans le registre r0 (0xfc20) :

```
0x0 : 20fc : hr0=0xfc
0x2 : 1020 : lr0=0x20
0x4 : c803 : syscall
##############################################
----------------------------------------------
----- Microcontroller firmware uploader -----
----------------------------------------------
()
:: Serial port connected.
:: Uploading firmware...
done.
()
System reset.
-- Exception occurred at 0006: Invalid instruction.
   r0:07C0     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:0006 fault_addr:0000 [S:1 Z:1] Mode:user
CLOSING: Invalid instruction.
```

Listing 35: Firmware écrasant r0 par le nombre d'instructions et résultat

Il est donc possible d'utiliser cette vulnérabilité pour écraser une adresse arbitraire avec le nombre d'instructions. Une tentative d'écraser une adresse contenant le ROM Code (et donc read-only) avec cette vulnérabilité donne le résultat suivant :

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

```
------------------------------------------
()
:: Serial port connected.
:: Uploading firmware...
done.
()
System reset.
-- Exception occurred at FDCC: Memory access violation.
   r0:FDDD     r1:07C0    r2:0001    r3:0100
   r4:0000     r5:00C0    r6:FC12    r7:0000
   r8:0000     r9:0000   r10:0000   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:FD6E
   pc:FDCC fault_addr:FDDE [S:0 Z:0] Mode:kernel
CLOSING: Memory access violation.
```

Listing 36: Tentative d'écraser une adresse mémoire read-only

Il n'est pas possible d'écrire cette adresse. Toutefois le code s'exécute avec des droits privilégiés comme mentionné : `Mode:kernel`. Il est donc possible d'exécuter du code avec des droits privilégiés dans le cadre des syscall.

4.3.5 Exploitation de la vulnérabilité

Le chapitre précédent a mis en évidence la possibilité d'écraser une adresse mémoire avec les droits privilégiés. L'idée de l'exploitation de cette vulnérabilité consiste donc à utiliser cette possibilité pour écraser l'adresse d'un autre syscall dans la jumptable afin que celui ci pointe vers du code contrôlé par l'attaquant. Le syscall 01 n'étant pas utilisé, il a été choisi d'écraser ce syscall. La valeur inscrite grâce à la vulnérabilité correspond aux nombre d'instructions exécutées, soit 0x7C0. Cette adresse mémoire correspond a une adresse à l'intérieur du Firmware dans le mapping mémoire. Il reste donc à écrire à cette adresse contrôlée par l'attaquant un code permettant d'aller lire la "Secret memory area" et de l'afficher à l'écran.

Le Firmware ci-dessous exploite la vulnérabilité en écrasant dans un premier temps le syscall 01, puis en appelant ce syscall. La suite du Firmware contient à l'adresse 0x7C0 le code permettant de lire la "Secret memory area" et de la copier en RAM. Ensuite le syscall 02 permettant d'afficher à l'écran est utiliser pour aller lire la RAM et l'afficher.

```
0x0 : 20f0 : hr0=0xf0
0x2 : 1000 : lr0=0x00 // Syscall 01 addr
0x4 : c803 : syscall  // ecraser pointeur syscall 01
0x6 : 11ff : lr1=0xff
0x8 : 21aa : hr1=0xaa
0xa : c801 : syscall // appel au code utilisateur
0xc : 2010 : hr0=0x10
0xe : 1000 : lr0=0x00 // addr a lire en ram
0x10 : 210b : hr1=0x0b
0x12 : 11ff : lr1=0xff
0x14 : c802 : syscall // affichage de l'adresse en ram

...


------------------------------------------------------------
// Bloc permettant la copie d'une addr a une autre
------------------------------------------------------------
0x73c : 220b : hr2=0x0b
0x73e : 12ff : lr2=0xff // longueur a copier
0x740 : 2300 : hr3=0x00
0x742 : 1301 : lr3=0x01
0x744 : 3555 : r5=r5^r5
0x746 : 5222 : r2=r2&r2
0x748 : a00a : jz 0x754
0x74a : 7223 : r2=r2-r3
0x74c : e415 : r4=[r1+r5]
0x74e : f402 : [r0+r2]=r4
0x750 : 6553 : r5=r5+r3
0x752 : b3f2 : jmp 0x746
0x754 : d00f : ret

```

```
...

0x7c0 : 21f0 : hr1=0xf0
0x7c2 : 1100 : lr1=0x00 // Addr a lire
0x7c4 : 2010 : hr0=0x10
0x7c6 : 1000 : lr0=0x00 // Addr ou copier
0x7c8 : c772 : call 0x73c
0x7ca : d800 : privileged instr
```

Listing 37: Firmware permettant de lire la "Secret Memory Area et de l'afficher"

Suite à l'exécution de ce firmware, la commande strings sur les données récupérées donne le résultat :

```
strings resp.out.bin | rev
.teser metsyS
 <66a65dc050ec0c84cf1dd5b3bbb75c8c@challenge.sstic.org>
  MUCH WIN
   SO OPERATIONAL
    VERY CHALLENGING
   SUCH EXPLOIT
  WOW
```

Listing 38: Récupération de l'adresse mail

Il ne reste plus qu'à envoyer un mail à cette adresse et à attendre anxieusement l'affichage du résultat sur la page du Challenge.

5 Conclusion et remerciements

La résolution de ce challenge bien que très prenante et exigeante sur le nombre d'heures de sommeil déduites a été très intéressante. Le fait de se confronter à un assembleur ARM 64 bits qui sera sûrement beaucoup plus utilisée dans le futur était une chose nouvelle et passionnante. La rétro-ingénierie des différentes machines virtuelles obligeant à développer des désassembleurs personnalisés était aussi captivante pour finir en exploitant une vulnérabilité sur un processeur inconnu.

Je tiens à remercier mes collègues Jean-Yves Burlett et David Berard pour leurs coups d'oeil avisés. De manière plus générale, je remercie Thales SecureLab pour m'avoir soutenu et m'avoir laissé du temps sur mes horaires de travail pour résoudre ce challenge. Merci à Axel Souchet Enfin, merci à Guillaume Delugré pour ce challenge original et aux organisateurs du SSTIC.

6 Annexes

6.1 Interprétation de la trace ADB

```
import sys
import struct

#define A_SYNC 0x434e5953
#define A_CNXN 0x4e584e43
#define A_OPEN 0x4e45504f
#define A_OKAY 0x59414b4f
#define A_CLSE 0x45534c43
#define A_WRTE 0x45545257
#define A_AUTH 0x48545541

knownpackets = ["SYNC", "CNXN", "OPEN", "OKAY", "CLSE", "WRTE", "AUTH"]

global nextwritepacket
nextwritepacket = None
global nextwritepacketparams
nextwritepacketparams = None
global outfile
outfile = None


def packetiterator(s):
    while len(s) > 0:
        if not s[:4] in knownpackets:
            s = s[4:]
        else:
            l = struct.unpack("I", s[0xc:0x10])[0]
            payloadlen = 24+l
            p = s[:payloadlen]
            s = s[payloadlen:]
            #print p.encode("hex")
            yield p

def decode_adb_OPEN(s):
    plen = struct.unpack("I", s[0xc:0x10])[0]
    print "OPEN > %s" % (p[0x18:0x18+plen])

def decode_sync_LIST(syncp):
    print "LIST <<<<<<<<<<<<<<<<<<<<"
    return ""

def decode_remain_data(syncp):
    global outfile
    global nextwritepacket
    global nextwritepacketparams

    remaining = nextwritepacketparams[0]
    if remaining < len(syncp):
        d = syncp[:remaining]
        outfile.write(d)
        #outfile.close()
        print ".... all data read"
        nextwritepacket = None
        nextwritepacketparams = None
        return syncp[remaining:]
    else:
        d = syncp
        outfile.write(d)
        remaining -= len(d)
        print ".... 0x%x remaining" % (remaining)
        nextwritepacketparams[0] = remaining
        return ""

def decode_sync_DATA(syncp):
    global nextwritepacket
    global nextwritepacketparams
```

```
67      global outfile
68
69      fnl = struct.unpack("I", syncp[4:8])[0]
70      print "DATA 0x%x bytes" % (fnl)
71      d = syncp[8:]
72      if fnl < len(d):
73          raise RuntimeWarning("pas gere")
74      outfile.write(d)
75      remain = fnl - len(d)
76      print ".... 0x%x remaining" % (remain)
77      if remain > 0:
78          nextwritepacket = decode_remain_data
79          nextwritepacketparams = [fnl-len(d)]
80      else:
81          outfile.close()
82      return ""
83
84  def decode_send_filename(syncp):
85      global nextwritepacket
86      global nextwritepacketparams
87      global outfile
88
89      nextwritepacket = None #decode_data_packet
90      fnl = nextwritepacketparams[0]
91      nextwritepacketparams = None
92      fname = syncp[:fnl]
93      print ">>> sending %s" % (fname)
94      ff = open(fname.replace("/","_"), "wb")
95      outfile = ff
96
97      return syncp[fnl:]
98
99  def decode_sync_SEND(syncp):
100     global nextwritepacket
101     global nextwritepacketparams
102
103     if len(syncp) != 8:
104         raise RuntimeWarning("bad size")
105     fnl = struct.unpack("I", syncp[4:])[0]
106     print "SEND (len=%x)" % (fnl)
107     nextwritepacket = decode_send_filename
108     nextwritepacketparams = [fnl]
109     return ""
110
111 def decode_sync_RECV(syncp):
112     print "RECV"
113     return ""
114
115 def decode_sync_STAT(syncp):
116     print "STAT"
117     return ""
118
119 def decode_sync_DONE(syncp):
120     global outfile
121
122     print "DONE <<<<<<<<<<<<<<<<<<<<"
123     if outfile:
124         outfile.close()
125         outfile = None
126     return ""
127
128 def decode_sync_QUIT(syncp):
129     print "QUIT ----------------------"
130     return ""
131
132 #===============================================================================
133 # The directory entries will be returned in the following form
134 # 1. A four-byte sync response id beeing "DENT"
135 # 2. A four-byte integer representing file mode.
136 # 3. A four-byte integer representing file size.
```

```
# 4. A four-byte integer representing last modified time.
# 5. A four-byte integer representing file name length.
# 6. length number of bytes containing an utf-8 string representing the file
#    name.
#
# When an sync response "DONE" is received the listing is done.
#=============================================================================

def decode_sync_DENT(syncp):
    dent = syncp[:4]
    syncp = syncp[4:]

    fmode = struct.unpack("I", syncp[:4])[0]
    syncp = syncp[4:]

    fsize = struct.unpack("I", syncp[:4])[0]
    syncp = syncp[4:]

    mtime = struct.unpack("I", syncp[:4])[0]
    syncp = syncp[4:]

    nlength = struct.unpack("I", syncp[:4])[0]
    syncp = syncp[4:]

    fname = syncp[:nlength]
    syncp = syncp[nlength:]

    print "File '%s'  (%d)" % (fname, len(syncp))
    return syncp

dictsync = { \
            "LIST": decode_sync_LIST, \
            "SEND": decode_sync_SEND, \
            "RECV": decode_sync_RECV, \
            "DENT": decode_sync_DENT, \
            "DONE": decode_sync_DONE, \
            "QUIT": decode_sync_QUIT, \
            "STAT": decode_sync_STAT, \
            "DATA": decode_sync_DATA, \
            }


def decode_adb_WRTE(s):
    plen = struct.unpack("I", s[0xc:0x10])[0]
    pload = p[0x18:0x18+plen]

    remain = pload
    while len(remain) > 0:
        if nextwritepacket:
            remain = nextwritepacket(remain)
        else:
            subco = remain[:4]
            if subco in dictsync:
                remain = dictsync[subco](remain)
            else:
                print "WRTE > %s" % (remain[:80])
                remain = ""


def decode_adb_CLSE(s):
    plen = struct.unpack("I", s[0xc:0x10])[0]
    if plen != 0:
        raise RuntimeWarning("packet not empty")
    print "CLOSE"

def decode_adb_OKAY(s):
    plen = struct.unpack("I", s[0xc:0x10])[0]
    if plen != 0:
        raise RuntimeWarning("packet not empty")
    #print "OKAY"
```

```
207
208
209 dicto = { \
210     "OPEN": decode_adb_OPEN, \
211     "WRTE": decode_adb_WRTE, \
212     "CLSE": decode_adb_CLSE, \
213     "OKAY": decode_adb_OKAY, \
214          }
215
216
217
218 def decode_adb(s):
219     command = s[:4]
220     #check xor
221     xco = struct.pack("I", struct.unpack("I", s[0x14:0x18])[0] ^ 0xffffffff)
222     if xco != command:
223         raise RuntimeWarning("invalid crc")
224
225     if command in dicto:
226         dicto[command](s)
227     else:
228         print "unknown command %s" % (command)
229
230 if __name__ == "__main__":
231     trace = ""
232     print sys.argv[1]
233     with open(sys.argv[1], "rb") as f:
234         trace = f.readlines()
235         bulkonly = filter(lambda x: ("Bi" in x) or ("Bo" in x), trace)
236         dataonly = filter(lambda x: " = " in x, bulkonly)
237         datapart = map(lambda x: x[x.index(" = ")+3:], dataonly)
238         datapart = map(str.strip, datapart)
239         l = "".join(datapart)
240         l = l.replace(" ", "")
241         trace = l.decode("hex")
242
243     for p in packetiterator(trace):
244         decode_adb(p)
```

Listing 39: Extracteur du binaire badbios de la trace

6.2 Code du désassembleur de la machine virtuelle présente dans le code ARM

```
import struct
import sys

regs = list()
instr = str()

def opcode_nop(val, regs):
    print "KKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKK %s" % (val.encode("hex"))
    return regs

def opcode_unknown(val, regs):
    print "UNK"
    print regs
    return regs

def opcode_exit(val, regs):
    print "EXIT " + "-"*32
    print regs
    return regs

def opcode_compare(val, regs):
    rval = struct.unpack("H", val)[0]
    rnum = (rval >> 8) & 0xf
    rin = rval >> 12
    print "SUB R%d = R%d - R%d" % (rnum, rnum, rin)
    regs[rnum] -= regs[rin]
    return regs

def opcode_add(val, regs):
    rval = struct.unpack("H", val)[0]
    rnum = (rval >> 8) & 0xf
    rin = rval >> 12
    print "ADD R%d = R%d + R%d" % (rnum, rnum, rin)
    regs[rnum] += regs[rin]
    return regs

def opcode_and(val, regs):
    rval = struct.unpack("H", val)[0]
    rnum = (rval >> 8) & 0xf
    rin = rval >> 12
    print "AND R%d = R%d & R%d" % (rnum, rnum, rin)
    regs[rnum] = regs[rnum] & regs[rin]
    return regs

def opcode_xor(val, regs):
    rval = struct.unpack("H", val)[0]
    rnum = (rval >> 8) & 0xf
    rin = rval >> 12
    print "XOR R%d = R%d ^ R%d" % (rnum, rnum, rin)
    regs[rnum] = regs[rnum] ^ regs[rin]
    return regs

def opcode_or(val, regs):
    rval = struct.unpack("H", val)[0]
    rnum = (rval >> 8) & 0xf
    rin = rval >> 12
    print "OR  R%d = R%d | R%d" % (rnum, rnum, rin)
    regs[rnum] = regs[rnum] | regs[rin]
    return regs

def opcode_lshift(val, regs):
    rval = struct.unpack("H", val)[0]
    rnum = (rval >> 8) & 0xf
    rin = rval >> 12
    print "LSR R%d = R%d << R%d" % (rnum, rnum, rin)
    regs[rnum] = regs[rnum] << regs[rin]
    return regs

```

```
69 def opcode_rshift(val, regs):
70     rval = struct.unpack("H", val)[0]
71     rnum = (rval >> 8) & 0xf
72     rin = rval >> 12
73     print "RSR R%d = R%d >> R%d" % (rnum, rnum, rin)
74     regs[rnum] = regs[rnum] >> regs[rin]
75     return regs
76 
77 def opcode_sign(val, regs):
78     rval = struct.unpack("H", val)[0]
79     rnum = (rval >> 8) & 0xf
80     rin = rval >> 12
81     print "PARITY R%d = parity(R%d)" % (rnum, rnum)
82     #regs[rnum] = regs[rnum] >> regs[rin]
83     return regs
84 
85 def opcode_mov(val, regs):
86     # potentiellement ormov
87     rval = struct.unpack("I", val)[0]
88     rnum = (rval >> 8) & 0xf
89     imm = rval >> 12
90     orm = rval >> 0x1c
91     if orm != 0:
92         raise RuntimeError("OR de marde")
93     s=""
94     if imm == 0x32e:
95         s=" "*8+"-- please enter key"
96     elif imm == 0x3fc:
97         s=" "*8+"-- buffer XXXXXX"
98     elif imm == 0x326:
99         s=" "*8+"-- buffer a zero"
100    elif imm == 0x354:
101        s=" "*8+"-- trying to decrypt payload"
102    elif imm == 0x38a:
103        s=" "*8+"-- invalid padding"
104    elif imm == 0x3f0:
105        s=" "*8+"-- payload.bin"
106    elif imm == 0x3c2:
107        s=" "*8+"-- decrypted payload written"
108    elif imm == 0x3a0:
109        s=" "*8+"-- cannot open file"
110    elif imm == 0x374:
111        s=" "*8+"-- invalid format"
112    print "MOV R%d, %x%s" % (rnum, imm, s)
113    regs[rnum] = imm
114    return regs
115 
116 def opcode_loadw(val, regs):
117     # potentiellement ormov
118     rval = struct.unpack("I", val)[0]
119     rnum = (rval >> 8) & 0xf
120     rin = (rval >> 12) & 0xf
121     imm = rval >> 16
122     print "LOADW R%d <- [R%d+%x]" % (rnum, rin, imm)
123     regs[rnum] = imm
124     return regs
125 
126 def opcode_store(val, regs):
127     # potentiellement ormov
128     rval = struct.unpack("I", val)[0]
129     rnum = (rval >> 8) & 0xf
130     rin = (rval >> 12) & 0xf
131     imm = rval >> 16
132     print "STORE R%d -> [R%d+%x]" % (rnum, rin, imm)
133     return regs
134 
135 def opcode_jcc(val, regs):
136     rval = struct.unpack("I", val)[0]
137     op1 = (rval >> 9) & 0b1111
138     op2 = (rval >> 0xd) & 0b111
```

```
139     op3 = (rval >> 0x10)
140     if op2==0:
141         print "JUMP %x" % (op3)
142     elif op2==3:
143         print "JNZ R%d %x" % (op1, op3)
144     else:
145         print "Jcc R%d cond_%x %x" % (op1, op2, op3)
146     print "-"*16
147     return regs
148 
149 def opcode_clearreg(val, regs):
150     rval = struct.unpack("I", val)[0]
151     rnum = (rval >> 8) & 0xf
152     print "MOV R%d, #0" % (rnum)
153     regs[rnum] = 0
154     return regs
155 
156 def opcode_inc(val, regs):
157     rval = struct.unpack("H", val)[0]
158     rnum = (rval >> 8) & 0xf
159     print "INC R%d" % (rnum)
160     regs[rnum] += 1
161     return regs
162 
163 def opcode_dec(val, regs):
164     rval = struct.unpack("H", val)[0]
165     rnum = (rval >> 8) & 0xf
166     print "DEC R%d" % (rnum)
167     regs[rnum] -= 1
168     return regs
169 
170 def opcode_syscall_read(val,regs):
171     print "SYS read"
172     return regs
173 
174 def opcode_syscall_open(val,regs):
175     print "SYS open"
176     return regs
177 
178 def opcode_syscall_write(val,regs):
179     print "SYS write"
180     return regs
181 
182 def opcode_syscall_close(val,regs):
183     print "SYS close"
184     return regs
185 
186 syscalls = { \
187     0: opcode_syscall_open,\
188     1: opcode_syscall_read,\
189     2: opcode_syscall_write,\
190     3: opcode_syscall_close,\
191         }
192 
193 def opcode_syscall(val, regs):
194     rval = struct.unpack("H", val)[0]
195     rnum = (rval >> 8) & 0xf
196     regs = syscalls[regs[rnum+1]](val, regs)
197     return regs
198 
199 def opcode_loadb(val, regs):
200     #raise RuntimeError("been there")
201     rval = struct.unpack("I", val)[0]
202     rnum = (rval >> 8) & 0xf
203     rin = (rval >> 12) & 0xf
204     imm = rval >> 16
205 
206     print "LOADB R%d <- [R%d]+%x" % (rnum, rin, imm)
207     return regs
208 
```

```
209
210 simu = { \
211     0: (4, opcode_clearreg),\
212     1: (4, opcode_mov),\
213     2: (4, opcode_loadw),\
214     4: (4, opcode_loadb),\
215     7: (4, opcode_store),\
216     8: (4, opcode_jcc),\
217     0xa: (2, opcode_xor),\
218     0xb: (2, opcode_or),\
219     0xc: (2, opcode_and),\
220     0xd: (2, opcode_lshift),\
221     0xe: (2, opcode_rshift),\
222     0x12: (2, opcode_add),\
223     0x13: (2, opcode_compare),\
224     0x16: (2, opcode_inc),\
225     0x17: (2, opcode_dec),\
226     0x1c: (2, opcode_exit),\
227     0x1d: (2, opcode_syscall),\
228     0x1e: (2, opcode_par),\
229     }
230
231 if __name__ == "__main__":
232     with open("output","rb") as f:
233         instr = f.read()
234         regs = [0]+list(struct.unpack("I"*16, instr[:64]))
235
236     while regs[16] < 0x232e:
237         pc = regs[16]
238         opcode = instr[pc]
239         sz = simu[ord(opcode)][0]
240         regs[16] += sz
241         sys.stdout.write("%08s\t" % (instr[pc:pc+sz][::-1].encode("hex")))
242         sys.stdout.write("%04x:\t" % (pc))
243         regs = simu[ord(opcode)][1](instr[pc:pc+sz], regs)
```

Listing 40: Code du désassembleur de la machine virtuelle présente dans le code ARM

6.3 Code de l'inverseur de LFSR

```
# -*- coding: utf-8 -*-
import struct
import bitarray

def lfsr_bit(state):
    nbit = bitarray.bitarray(1)
    nbit[0] = (((state[0] ^ state[2]) ^ state[3]) ^ state[63])
    nstate = nbit + state[:63]
    obit = nstate[63]
    return (obit,nstate)

def reverse_lfsr_1bit(state):
    nbit = bitarray.bitarray(1)
    nbit[0] = (((state[0] ^ state[1]) ^ state[3]) ^ state[4])
    nstate = state[1:]+nbit
    return nstate

def lfsr_unbit(ibit, state):
    nbit = bitarray.bitarray(1)
    nbit[0] = ibit
    state = state[1:] + nbit
    return state

def grab_byte(lfsr):
    v = 0
    s = lfsr
    for i in range(8):
        o, s = lfsr_bit(s)
        v <<= 1
        v |= o
    return v, s

if __name__ == "__main__":
    s = bitarray.bitarray()

    lo = struct.pack("<I", 0xca8f5382L^0x80)
    hi = struct.pack("<I", 0x6ab654c3L)
    neu=bitarray.bitarray(64)
    for c in lo+hi:
        by = ord(c)
        for bi in range(8):
            neu = lfsr_unbit(by & 1, neu)
            by >>= 1
    neu = lfsr_unbit(True, neu)

    print "hyp pre-final state %s" % (neu.tobytes().encode("hex"))
    for i in range(8):
        b,neu=grab_byte(neu)
        print "> %02x" % (b)
    print "hyp final state %s" % (neu.tobytes().encode("hex"))

    KK=neu
    for i in range(8):
        KK=reverse_lfsr_1bit(KK)
    print "hyp final state 1 byte less %s" % (KK.tobytes().encode("HEX").upper())

    for i in range(0x2000*8):
        neu=reverse_lfsr_1bit(neu)
    print "potential init state %s" % (neu.tobytes().encode("HEX").upper())
    g = map(ord, open("black-payload","rb").read())
    oo=open("output-neu","wb")
    for i in g:
        b,neu=grab_byte(neu)
        oo.write(chr(b^i))
    print "final state %s" % (neu.tobytes().encode("hex"))
```

Listing 41: Code de l'inversion du LFSR

6.4 Code du désassembleur de la machine virtuelle du micro-contrôleur inconnu

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


def decode_instr(i,offset):

  ret = "unk"
  if i[0] == "1":
    ret = "lr%s=0x%s" % (i[1],i[2:4])
  if i[0] == "2":
    ret = "hr%s=0x%s" % (i[1],i[2:4])
  if i[0] == "3":
    ret = "r%s=r%s^r%s" % (i[1],i[2],i[3])
  if i[0] == "4":
    ret = "r%s=r%s|r%s" % (i[1],i[2],i[3])
  if i[0] == "5":
    ret = "r%s=r%s&r%s" % (i[1],i[2],i[3])
  if i[0] == "6":
    ret = "r%s=r%s+r%s" % (i[1],i[2],i[3])
  if i[0] == "7":
    ret = "r%s=r%s-r%s" % (i[1],i[2],i[3])
  if i[0] == "8":
    ret = "r%s=r%s*r%s" % (i[1],i[2],i[3])
  if i[0] == "9":
    ret = "r%s=r%s/r%s" % (i[1],i[2],i[3])
  if i[0] == "a":
    if (int(i[1],16)%4 == 0) or (int(i[1],16)%4 == 1):
      ret = "jz 0x%x " % (int(i[2:4],16)+2+offset)
    if int(i[1],16)%4 == 2:
      ret = "jz 0x%x " % (int(i[2:4],16)+2-512+offset)
    if int(i[1],16)%4 == 3:
      ret = "jz 0x%x " % (int(i[2:4],16)+2-256+offset)

  if i[0] == "b":
    if (int(i[1],16)%4 == 0) or (int(i[1],16)%4 == 1):
      ret = "jmp 0x%x " % (int(i[1:4],16)+2+offset)
    if int(i[1],16)%4 == 2:
      ret = "jmp 0x%x " % (int(i[2:4],16)+2-512+offset)
    if int(i[1],16)%4 == 3:
      ret = "jmp 0x%x " % (int(i[2:4],16)+2-256+offset)
  if i[0] == "c":
    if(int(i[1],16)<8):
      if (int(i[1],16)%4 == 0) or (int(i[1],16)%4 == 1):
        ret = "call 0x%x " % (int(i[1:4],16)+2+offset)
      if int(i[1],16)%4 == 2:
        ret = "call 0x%x " % (int(i[2:4],16)+2-512+offset)
      if int(i[1],16)%4 == 3:
        ret = "call 0x%x " % (int(i[2:4],16)+2-256+offset)
    else :
      ret ="syscall time"

  if i[0]=="d":
    if(int(i[1],16)<8):
      if i=="d00f":
        ret = "ret"
    else:
      ret = "privileged instr"
  if i[0] == "e":
    ret = "r%s=[r%s+r%s]" % (i[1],i[2],i[3])
  if i[0] == "f":
    ret = "[r%s+r%s]=r%s" % (i[2],i[3],i[1])




  return ret

```

```
69 f=open("fw.bin","r")
70 offset=0
71 for l in f.readlines() :
72   for instr in [l[i:i+2] for i in range(0,len(l),2)]:
73     print str(hex(offset))+" : "+instr.encode("hex") + " : "+decode_instr(instr.encode("hex"),
           offset)
74     offset+=2
75
76
77
78 f.close()
```

Listing 42: Code du désassembleur de la machine virtuelle du micro-contrôleur inconnu