Solution Challenge SSTIC 2014

Martin Balch <martin@balch.fr>

Plan

- Le défi
- L'environnement de travail
- Analyse trace USB
- Analyse de badbios.bin
- Cryptanalyse de badbios.bin
- Analyse de mcu.zip
- Exploitation de la VM
- Conclusion

Le défi

Le défi consiste à analyser une trace USB. L'objectif est d'y retrouver une adresse e-mail (...@challenge.sstic.org).
Pour participer au concours, il faut, une fois cette adresse trouvée :

- envoyer aussitôt un courrier électronique à cette adresse (pour le classement rapidité);
- **PUIS** envoyer dans les dix jours (à partir de la date de réception du message précédent par le serveur mail du SSTIC) une réponse qui décrit la démarche suivie (pour le classement qualité).

La trace est disponible ici : http://static.sstic.org/challenge2014/usbtrace.xz
MD5: 3783cd32d09bda669c189f3f874794bf - usbtrace.xz

L’environnement de travail

Logiciels utilisés:

- Windows 7 64
- Virtual box https://www.virtualbox.org/
- Linux Debian ‘Jessie’ http://www.debian.org/devel/debian-installer/
- Notepad++ http://notepad-plus-plus.org/
- IDA pro https://www.hex-rays.com/products/ida/
- Python 2.7 https://www.python.org/
- qemu http://wiki.qemu.org/

usbtrace.xz

- On commence par télécharger le fichier et on utilise la commande ‘file’ afin de déterminer le type de fichier:

```
# wget http://static.sstic.org/challenge2014/usbtrace.xz
# file usbtrace.xz
usbtrace.xz: XZ compressed data
```

- On a a priori affaire à un type d’archive de type ‘XZ’, un format de compression sans perte basé sur LZMA2 (http://en.wikipedia.org/wiki/Xz)
- On décompresse alors la trace USB grâce aux commandes suivantes:

```
# apt-get install xz-utils
# xz -d usbtrace.xz
# file usbtrace
usbtrace: UTF-8 Unicode text, with very long lines
```

- Maintenant analysons la trace USB...

usbtrace

- On ouvre le fichier dans notre éditeur de texte préféré et on voit:

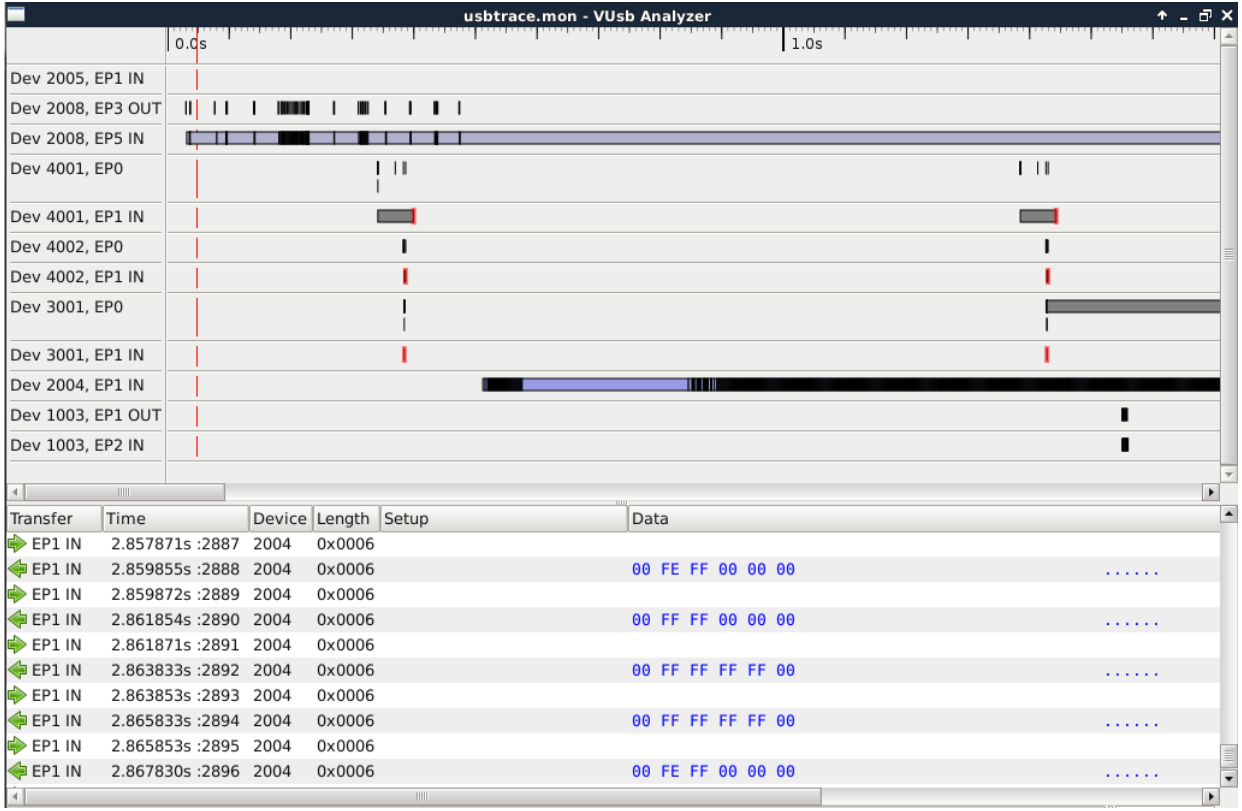
```
Date: Thu, 17 Apr 2015 00:40:34 +0200
To: <challenge2014@sstic.org>
Subject: Trace USB

Bonjour,

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

--

ffff8804ff109d80 1765779215 C Ii:2:005:1 0:8 8 = 00000000 00000000
ffff8804ff109d80 1765779244 S Ii:2:005:1 -115:8 8 <
ffff88043ac600c0 1765809097 S Bo:2:008:3 -115 24 = 4f50454e fd010000 00000000 09000000 1f030000
b0afbab1
ffff88043ac600c0 1765809154 C Bo:2:008:3 0 24 >
[...]
```

- Après les 10 lignes d’e-mail, on trouve environ 3000 lignes de log USB au format
  - “Linux usbmon log, raw ASCII format (*.mon)”

usbtrace.mon

- On commence par changer l’extension du fichier en .mon et on enlève les 10 premières lignes.
- N’étant pas familier avec ce format de fichier, on part en quête d’un logiciel permettant de nous donner un peu plus d’information sur la trace.
- Après quelques requêtes Google, on trouve un logiciel qui convient parfaitement, vusb-analyzer disponible ici: http://vusb-analyzer.sourceforge.net/download.html
- Ce dernier plante malheureusement à la ligne 1097, celle-ci étant mal formée, on supprime tout simplement cette ligne en croisant les doigts qu’elle ne soit pas nécessaire à la résolution du challenge, et on relance le tout.
- Voici ce que l’on voit ensuite:

usbtrace.mon



ADB protocol

- L’email nous donne un indice précieux pour gagner du temps: il s’agirait d’une communication entre un téléphone de type Android et un PC.
- Le protocole de communication utilisé pour communiquer, installer, débugger ou rooter un appareil Android est appelé ADB (Android Debug Bridge)
  - Voici quelques références utiles afin de comprendre ce protocole:
    - https://android.googlesource.com/platform/system/core/+/master/adb/protocol.txt
    - http://blogs.kgsoft.co.uk/2013_03_15_prg.htm
    - http://powerdroid.googlecode.com/svn-history/r3/trunk/src/droid/adb.py

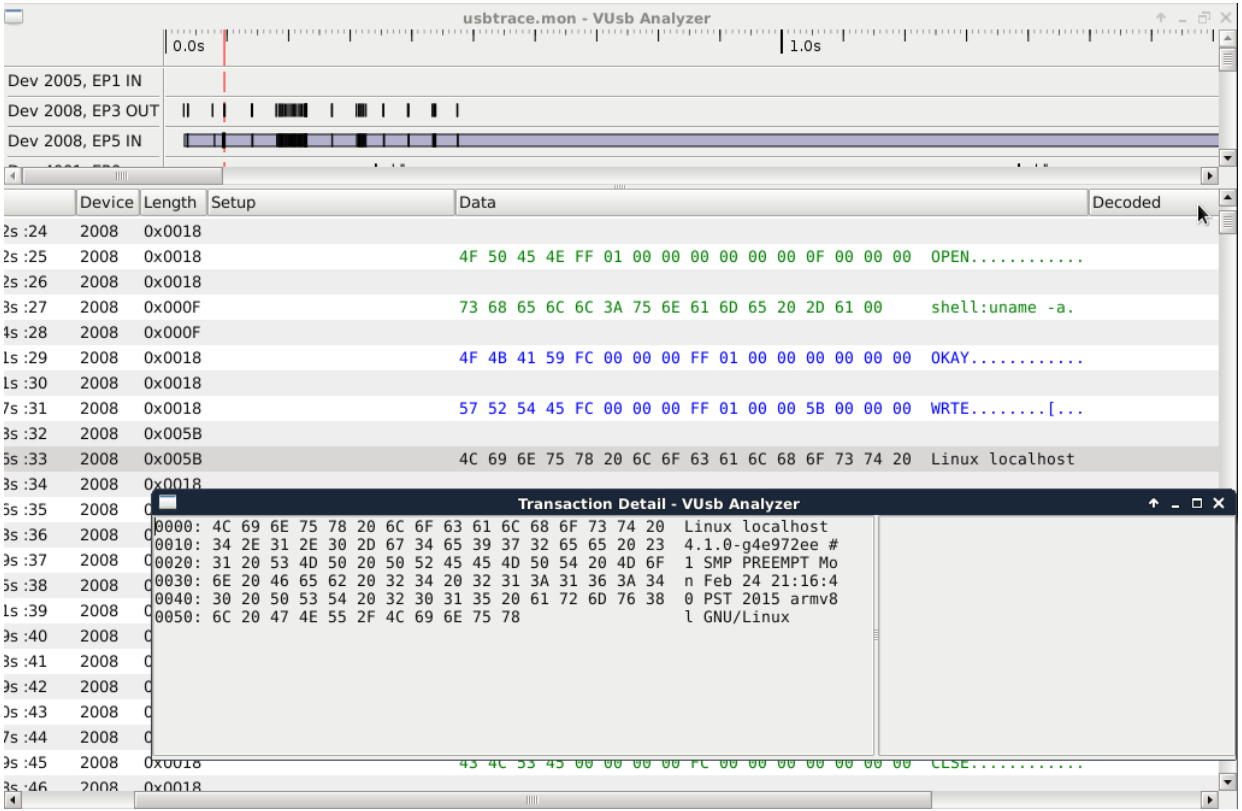
- Ce protocole permet notamment d'exécuter des commandes et de transférer des fichiers depuis et vers l’appareil.

- En continuant de regarder la trace usbtrace.mon avec vusb-analyzer, on confirme effectivement que le protocole ADB est utilisé car on voit les mots clés: SYNC, OPEN, WRTE et OKAY très régulièrement dans le fichier.

- Assez tôt dans la trace, on peux apercevoir la commande suivante:

```
uname -a
Linux localhost4.1.0-g4e972ee #1 SMP PREEMPT Mon Feb 24 21:16:40 PST 2015 armv8l GNU/Linux
```

ADB protocol



ADB protocol

- La commande uname -a nous informe du type de processeur qui équipe le téléphone de la victime, c'est un "ARMv8" aussi connu sous le nom "Aarch 64".
- Ce genre de processeur ARM 64 bits, très récent, se trouve notamment dans les iPhone 5S dernière génération.
- On trouve également dans les commandes ADB une liste de fichiers et dossiers présents sur le téléphone:
  - Samsung
  - Android
  - clockworkmod
  - CyanogenMod
  - CSW-2014-Hacking-9.11_uncensored.pdf
  - NATO_Cosmic_Top_Secret.gpg
  - …
- Mais on constate surtout quelque chose de très intéressant: un transfert du fichier "badbios.bin"

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

Transfert de badbios.bin

- Le transfert du fichier ‘badbios.bin’ est constitué de 19 blocs de 4096 octets et d’un bloc de 233 octets.
- Le premier bloc commence par le chemin complet du fichier
  - il est donc stocké dans /data/local/tmp/
- Son mask unix
  - 33261 decimal = 100755 octal = -rwxr-xr-x
- Le mot clé DATA
- La longueur des données au format uin32_t little endian
  - 0x010000 = 65536 octets
- Et enfin le bloc de données

- Après ces 65536 octets on trouve un autre bloc DATA d’une longueur de 0x30b0 (12464) octets

- Le fichier badbios.bin est donc un exécutable d’une longueur de 78000 octets

Analyse de badbios.bin

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

- Comme on s'en doutait en voyant les premiers octets du fichier (`7f 45 4c 46 - signature ELF`), on a affaire à un fichier exécutable linux ARM 64 bits.

- Il va falloir mettre en place un système de test qui va nous permettre d'émuler cette architecture + os afin de pouvoir lancer & debugger le programme.

- Le projet debian a une page très détaillée permettant de mettre en place de genre de système grâce à qemu:
  - https://wiki.debian.org/Arm64Qemu
  - http://people.debian.org/~wookey/bootstrap/rootfs/debian-unstable-arm64.tar.gz

- Pour résumer en quelques mots la procédure: on installe une version beta de qemu, on créé un chroot contenant un système debian ARM64 minimaliste et on configure "binfmts-misc" pour exécuter automatiquement, via qemu les programes ARM64/

Exécution de badbios.bin

```
# ./badbios.bin
:: Please enter the decryption key: hello world!
   Wrong key format.
```

- Le programme nous demande une clé de déchifrement et, après avoir rentré “hello world!”, on est informé que le format de clé n’est pas valide
- On retente notre chance en utilisant uniquement des caractères hexadécimaux:

```
# ./badbios.bin
:: Please enter the decryption key: 00112233445566778899AABBCCDDEEFF
:: Trying to decrypt payload...
   Invalid padding.
# 8899AABBCCDDEEFF
-bash: 8899AABBCCDDEEFF: command not found
```

- Cette fois ci notre clé est acceptée et le programme essaye de déchiffrer, sans succès, sa charge utile.
- On constate de plus que uniquement les 16 premiers caractères ont été lus par badbios.bin
- Ce qui correspond donc à une clé de 64 bits, impossible à brute-forcer (avec nos moyens) dans un temps raisonnable.

- Il va falloir plonger plus profond dans ses entrailles.

Structure de badbios.bin

- On commence avec objdump:

```
(chroot-arm64)root@jessie:~# objdump -x /tmp/badbios.bin

/tmp/badbios.bin:     file format elf64-littleaarch64
/tmp/badbios.bin
architecture: aarch64, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x00000000000102cc

Program Header:
    LOAD off    0x0000000000000000 vaddr 0x0000000000010000 paddr 0x0000000000010000 align 2**16
         filesz 0x00000000000005d8 memsz 0x00000000000005d8 flags r-x
    LOAD off    0x0000000000001000 vaddr 0x0000000000021000 paddr 0x0000000000021000 align 2**16
         filesz 0x0000000000011f50 memsz 0x0000000000011f50 flags rw-
    NOTE off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**3
         filesz 0x0000000000000000 memsz 0x0000000000000000 flags r--
private flags = 0:

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         0000048c  000000000001010c  000000000001010c  0000010c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000040  0000000000010598  0000000000010598  00000598  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00011f50  0000000000021000  0000000000021000  00001000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
SYMBOL TABLE:
no symbols
```

Désassemblage de badbios.bin

- A première vue, le segment .text semble petit (0x048c) donc pas beaucoup de code, et le segment .data lui est plus conséquent (`0x11f50`).
- Le fichier est aussi linké statiquement et n'importe aucun symbole.
- On est très probablement en présence d'un binaire 'packé'.

- On arrive vite à la même conclusion en analysant le programme avec IDA car:
  - Il y a en tout et pour tout 3 fonctions dans le segment .text
  - Le segment data contient des données semblant aléatoires.

- A cette étape, on se pose la question suivante: A-t-on vraiment envie de reverse engineerer le packer?

- Étant paresseux, ne connaissant pas bien (du tout?) l'assembleur ARM64, on se dit que l'approche dynamique ne serait pas si mal que ça :)

Setup d’analyse dynamique

- On installe gdb dans notre chroot
  - apt-get install gdb

- En dehors de notre chroot:

```
# qemu-arm64 -g 12345 -strace ./badbios.bin
```

- Dans le chroot:

```
# gdb ./badbios.bin
(gdb) target remote :12345
```

- De cette manière on peut faire du “step-by-step” debugging en arm64 avec GDB
- Il peut aussi être très pratique de créer un fichier .gdbinit, et d’écrire quelques fonctions pour afficher automatiquement l’état de tous les registres, de la pile ou de la mémoire après chaque instruction exécutée.
  - voir https://github.com/gdbinit/Gdbinit pour un exemple x86 / arm

Appels systèmes - strace

- Une exécution du programme dans notre environnement d'analyse dynamique nous donne les informations suivantes:

```
# echo 0011223344556677 | qemu-arm64 -g 12345 -strace ./badbios.bin
5129 mmap(0x0000000000400000,12288,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED,0,0) = 0x0000000000400000
5129 mprotect(0x0000000000400000,12288,PROT_EXEC|PROT_READ) = 0
5129 mmap(0x0000000000500000,69632,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP_FIXED,0,0) = 0x0000000000500000
5129 mprotect(0x0000000000500000,69632,PROT_READ|PROT_WRITE) = 0
5129 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000801000
5129 mmap(NULL,65536,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000802000
5129 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000812000
5129 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000813000
5129 write(1,0x813000,36):: Please enter the decryption key:  = 36
5129 munmap(0x0000004000813000,36) = 0
5129 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000814000
5129 read(0,0x814000,16) = 16
5129 munmap(0x0000004000814000,16) = 0
5129 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000815000
5129 write(1,0x815000,32):: Trying to decrypt payload...
 = 32
5129 munmap(0x0000004000815000,32) = 0
5129 mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) = 0x0000004000816000
5129 write(2,0x816000,20)   Invalid padding.
 = 20
5129 munmap(0x0000004000816000,20) = 0
5129 exit_group(0)
```

Appels systèmes - strace

- La trace des appels systèmes est utile et nous donne les informations suivantes:

- Le programme alloue une plage mémoire de 0x3000 octets à une adresse fixe (0x00400000)
  - La zone mémoire est ensuite rendue exécutable
  - Probablement pour y stocker le code décrypté
- Le programme alloue une plage mémoire de 0x11000 (69632) octets à une adresse fixe
  - Probablement pour y stocker des données décryptées

- Avant chaque opération IO, le programme alloue une page mémoire de 0x1000 (4096) octets
  - Cette page mémoire est utilisée pour l'opération (read / write)
  - La page mémoire est libérée après l'opération
  - On ne risque donc pas de trouver grand chose d'intéressant à ces endroits.

- On décide alors de dumper les deux plages mémoire allouées à adresses fixes car on a de grandes chances d'y trouver le code et les données de badbios.bin en clair

Analyse dynamique de badbios.bin

- Juste avant de récupérer ces zones mémoires, on essaye de confirmer notre intuition en lançant le programme sous gdb et lorsque le programme nous demande de rentrer la clé de chiffrement, on va à la place appuyer sur CTRL-C afin de stopper l'exécution du programme et regarder l'adresse de l'instruction en cours.

```
(gdb) c
Continuing.

Program received signal SIGINT, Interrupt.
0x0000000000400ee8 in ?? ()
(gdb) i r $pc
pc             0x400ee8 0x400ee8
```

- On constate que PC (le registre contenant l'adresse de la prochaine instruction) vaut 0x400ee8
  - Une adresse contenue dans la zone de mémoire exécutable allouée dynamiquement!

- Notre intuition était donc bonne, le packer a déchiffré le code, et la placé à l'adresse 0x400000
- Afin de passer la protection du packer, il suffit d'attendre le bon moment et de dumper la mémoire!
- Pas besoin de perdre du temps à analyser statiquement le binaire!
  - Pour info, le transfert de contrôle au code unpacké se fait à l'adresse `0x0102C0 (blr x2)`

Analyse dynamique de badbios.bin

- On récupère la mémoire avec la fonction python suivante:

```
def dmem(pid, addr, size):
        f = open('/proc/%d/mem' % pid)
        f.seek(addr)
        d = f.read(size)
        f.close()
        f = open('0x%x.bin' % addr, 'wb+')
        f.write(d)
        f.close()
```

- Afin d'être sur que toute la mémoire ait été décryptée, et soit accessible, on utilise la même technique que précédemment, on interrompt l'exécution avec CTRL-C quand le programme nous demande d'entrer la clé de déchiffrement.
- On charge ensuite le segment dans IDA (Load File - Additional Binary File) et on spécifie l'adresse `0x400000`
- On doit ensuite lancer le désassemblage au bon endroit (`0x04000B0`) que l'on identifie soit en parsant le header ELF se trouvant au début du segment, soit en tâtonnant, comme je l'ai fait, en essayant le 1er octet non nul après une longue série (padding de fin d'en-tête) à l'adresse `0x4000B0`

- On peut enfin voir le code en clair du programme!!!

Deception!

- En fait non!
- On a passé une couche de protection, mais ce n’était pas la seule
- Le code est obfusqué:
  - Instructions décomposées

```
MOV          X8, #8
EXTR         X8, X8, X8, #0x38
EXTR         X8, X8, X8, #0x3E
EXTR         X8, X8, X8, #7     ; tout ca pour dire “mov x8, 64”
SVC          0                  ; syscall 64 = WRITE
```

  - Des branchements qui nous baladent un peu dans tous les sens
  - On ne trouve toujours pas les chaînes de texte du programme en clair
- Par contre on trouve quelques nouvelles chaînes en clair:
  - “No error.”, “Bad instruction pointer.”, “Invalid instruction.”, “Memory fault.”, “Internal error.”, etc.
  - Mais ces dernières n’ont jamais l’air d’être utilisées durant l’exécution normale du programme.

- Bref, on n’est pas au bout de nos peine. Tout cela ressemble bien à un autre packer, d’un genre plus coriace que le précédent:
  - Un virtualiseur de code!

Virtualisation de code

- Cette étape du challenge est à mon avis la plus difficile de toutes.
- Pour des raisons de temps et de place, je ne vais pas pouvoir décrire toutes mes tentatives, frustrations et échecs.
- C'était cependant très gratifiant à la fois:
  - De découvrir de manière approfondie l'assembleur arm64
  - De reverse engineerer pour la première fois un virtualiseur de code
  - Et de finalement en venir à bout!
- Mon approche a été la suivante:
  - Privilégier l'analyse dynamique autant que possible
    - Pour comprendre "sur le tas" plutôt que de passer des heures dans les manuels de spécifications ARM 64 (http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0802a/a64_general_alpha.html)
    - Scripter GDB au maximum afin de faciliter et compléter mon analyse statique
  - Identifier d'abord tous les syscalls (SVC 0)
  - Identifier la fonction de "dispatch" de la machine virtuelle (VM)
  - Identifier le nombre et les fonctions de chaque instruction virtuelle
  - Identifier le type de VM utilisée
  - Trouver et extraire le code virtualisé
  - Écrire un désassembleur pour cette machine virtuelle

Fausses pistes...

Voici quelques une des approches non concluantes que j'ai poursuivi:

- L'allocation dynamique des pages mémoires utilisées pour les IO mentionnées précédemment
  - J'ai essayé de stopper l'exécution du programme lors de l'accès en écriture ou lecture à ces adresses
  - qemu ne supportant pas les hardware breakpoints, j'ai essayé de changer manuellement la protection des pages pour générer des violations d'accès
  - Mais cette approche n'a pas aboutit car c'est qemu qui plantait du coup, et l'erreur n'était pas récupérable sous gdb
- Run-traces
  - Ayant un petit faible pour ollydbg (sous windows) et ses run traces, j'ai souhaité en générer pour analyser statiquement et séquentiellement l'exécution du programme (grep ftw).
  - Avec toutes les couches de virtualisation et d'émulation successives, les performances n'étaient vraiment pas au rendez vous!
    - gdb dans qemu dans linux dans virtualbox dans windows - inception!
  - Je n'ai jamais réussi à en finir une seule, même en plus de 24h!
- Salsa20 / chacha !
  - On verra ça dans le prochain slide…
- M'entêter à ne pas vouloir analyser statiquement et méthodiquement le fonctionnement de la VM sous IDA
  - Que j'ai quand même du faire en fin de compte.

Chacha / Salsa20

- En analysant la mémoire du programme, notamment aux plages allouées dynamiquement (0x400080XXXX), on tombe vite sur:

```
0x4000801010:   0x61707865      0x3120646e      0x79622d36      0x6b206574 ; expand 16-byte k
0x4000801020:   0x05b1ad0b      0x05b1ad0b      0x05b1ad0b      0x05b1ad0b ; 0badb105 = badbios
0x4000801030:   0x05b1ad0b      0x05b1ad0b      0x05b1ad0b      0x05b1ad0b
0x4000801040:   0x0000000e      0x00000000      0x00000000      0x00000000
```

- Une recherche rapide sur google nous indique que “expand 16-byte k” est une constante utilisée dans les algorithmes de chiffrement chacha et salsa20 de D. J. Bernstein
  - http://cr.yp.to/snuffle/salsa20/merged/salsa20.c
  - http://cr.yp.to/streamciphers/timings/estreambench/submissions/salsa20/chacha8/ref/chacha.c
- On en déduit naturellement que le programme badbios utilise soit chacha, soit salsa20 et que la clé demandée par le programme va être mélangée à cette constante et aux “IVs” 0xbadbios
  - … et non
  - la clé demandée par le programme n’a absolument rien a voir avec chacha, salsa20, “expand 16byte k” ou 0x0badb105
- Mais ce n’est pas une fausse piste non plus, et l’étude du code salsa20.c n’a pas été en vain.

Crypto-Code-Virtualization-Packer

- Durant nos analyses dynamiques et tentatives de run-traces avortées, on arrive quand même à identifier une des fonction les plus utilisée du programme:
  - sub_4000E8
  - que l'on nomme "salsa_encrypt" car elle ressemble comme 2 gouttes d'eau au code C de salsa20 ECRYPT_encrypt_bytes
- Cette dernière est appelée (via 1 ou 2 indirections) par une autre fonction également très très utilisée par le programme:
  - sub_4020C4
  - que l'on nomme "protect_unprotect_page"
  - Cette fonction accède à la mémoire décrite précédemment (expand 16 byte k + 0xbadb105)
  - Voici un extrait de log des appels de cette dernière:

```
crypt: IN 0x802000      OUT 0x812000    LEN 64     IV0 0x0                  IV1 0x0 IV2 0x0 IV3 0x0
crypt: IN 0x812000      OUT 0x802000    LEN 64     IV0 0x0                  IV1 0x0 IV2 0x0 IV3 0x0
crypt: IN 0x80a4c0      OUT 0x812000    LEN 64     IV0 0x213           IV1 0x0 IV2 0x0 IV3 0x0
crypt: IN 0x802000      OUT 0x812000    LEN 64     IV0 0x0                  IV1 0x0 IV2 0x0 IV3 0x0
crypt: IN 0x812000      OUT 0x802000    LEN 64     IV0 0x0                  IV1 0x0 IV2 0x0 IV3 0x0
```

- Les arguments de cette fonction sont:
  - x0 = pointeur vers l'état de la VM (quasiment toutes les fonctions prennent cet argument en x0)
  - x1 = mémoire source
  - x2 = mémoire destination
  - x3 = longueur
- IVn correspondent à la valeur de la mémoire à l'adresse x0 + 0x30 + n

Crypto-Code-Virtualization-Packer

- Grâce au log du slide précédent, on arrive à comprendre que cette fonction est utilisée pour décrypter une partie de la mémoire du programme et que juste un peu plus tard, la mémoire décryptée est re-chiffrée.
- Le byte code virtuel n'est donc jamais présent en entier dans la mémoire!
- On comprends aussi que l'adresse `0x812000` qui est en fait `0x4000812000` (troncation 32bits) est la zone ou les données sont déchiffrées.
- La valeur des IVs change et dépend de l'adresse mémoire source (addr_src - `0x4000802000`) / 64
- On va alors instrumentaliser cette fonction afin de décrypter tout le byte code virtualisé avec la fonction gdb suivante:

```
define decrypt
                b *0x00000000004000e8                                   # salsa_encrypt_bytes
                c
                set $sa = 0x4000802000                                  # début du byte cote virtualisé crypté
                set $ea = $sa + 0x10000                                 # longueur à décrypter
                set $a = $sa
                while ($a < $ea)
                                set $x0 = 0x4000801010                                  # ptr vm_state
                                set $x1 = $a                                            # addr src
                                set $x2 = 0x4000812000                                  # addr dst
                                set $x4 = 0x40                                          # len = 64 b
                                set $pc = 0x00000000004000e8                            # set PC to addr of
salsa_encrypt_bytes
                                set *(unsigned int*)($x0 + 0x30) = ($a - $sa) / 0x40    # set correct IV
                                u *0x0000000000400400                                   # wait til RET of salsa_encrypt_bytes
                                printf "0x%x\n", $a                                     # print decrypted address
                                x/64uxb 0x4000812000                                    # dump decrypted bytes
                                printf "=========================\n"
                                set $a = $a + 0x40                                      # increment address
                end
end
```

Machine Virtuelle

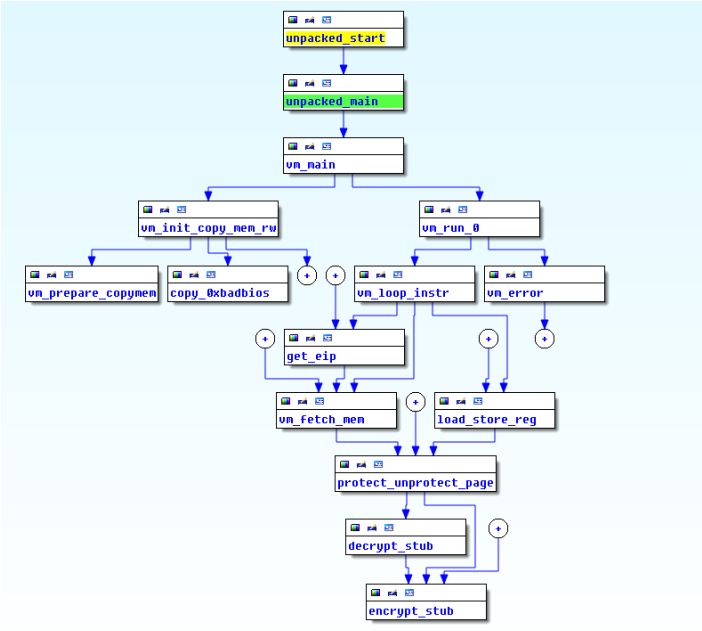
- La routine précédente nous à permis de récupérer le bytecode du programme virtualisé.
  - On a récupéré les 0x400 octets qui composent le programme virtualisé
  - Et on voit enfin les strings “Please enter the decryption key”, “Invalid padding”, etc. en clair!

- Il faut maintenant comprendre cette dernière et pour cela:
  - Identifier la séquence d’initialisation de la VM
    - Pour cela on suit en pas à pas l’exécution du programme après l’allocation de mémoire exécutable jusqu’au premier saut vers cette dernière
  - Identifier la boucle qui exécute les instructions virtuelles
    - On recherche une instruction de type BR (branch register) qui puisse être utilisée pour appeler une routine dynamiquement en fonction d’un opcode.
    - Pour limiter l’étendue de la recherche, on peut partir d’un point d'arrêt lorsque la VM affiche “Trying to decrypt payload”, faire du pas à pas et compter les sorties de fonctions (“ret” ou “br x30”), on devrait alors retomber sur la boucle d’exécution d’instructions virtuelles assez vite.
  - Identifier et comprendre les différentes opcodes supportés par la VM
    - Une fois la routine de “dispatch” identifiée, il faut mettre un breakpoint à l’adresse du saut vers registre (`0x040285C BLR X2`) et enregistrer les valeurs possible de X2

- Bref, un jeu d’enfant ;)

Machine Virtuelle

- Voici un schéma partiel de la machine virtuelle:



- Et l'adresse des fonctions importantes:

0x00400514	unpacked_start
0x004004D8	unpacked_main
0x004000B0	vm_main
0x00402960	vm_init
0x00402914	vm_run_0
0x00402754	vm_loop_instr
0x00402070	vm_error
0x004020C4	protect_unprotect_page
0x004000E8	salsa_encrypt_bytes
0x0040285C	vm_dispatch (pas une fonction)

Notes sur la VM

- Au fur et à mesure que l'on analyse et debug le code virtualisé on arrive à déduire que:
    - Les registres sont stockés à l'adresse 0x4000812000 et sont au nombre de 16 (avec un registre r0 special qui vaut toujours 0, et r16 = PC)
    - La mémoire que le programme essaye de déchiffrer se trouve à l'adresse 0x4000801000 et a une longueur de 0x8000 octets
        - On utilise la même technique que celle qu'on a utilisée pour extraire le code afin de la récupérer.

Architecture de la VM

- On arrive finalement à identifier 31 instructions différentes dont environ 10 ne sont pas utilisées par le code virtualisé.
- Ce nombre 31 m'a interpellé et j'ai lancé des recherches sur internet pour trouver si une architecture connue pouvait correspondre.
- Et il en existe une!
- Berkeley RISC 1!
  - whoa, c'est old school! ça date de 1981! Je n'était même pas né!
  - http://www.eecs.berkeley.edu/Pubs/TechRpts/1982/CSD-82-106.pdf
  - http://web.cecs.pdx.edu/~alaa/courses/ece587/spring2011/papers/patterson_isca_1981.pdf

- **Sympa, le tout dernier processeur ARM émule son plus vieux ancêtre :)**

- On lit toute la doc, surtout les annexes expliquant la liste et le format des instructions, et on repart sur IDA pour essayer de coller des noms à ces 31 instructions.
- Et après on commence à écrire un désassembleur RISC1.

RISC 1 debugging

- Pour être sûre de bien désassembler le code et afin de nous aider à identifier toutes les instructions, on écrit une fonction gdb nous permettant de tracer les exécutions du programme virtualisé en pas à pas:

```
define tracevi
                d breakpoint                            # suppression des breakpoints existants
                b *0x04027C0                            # break au retour de get_vm_eip

                c
                set $_eip = $w0                         # sauvegarde de l'eip

                d breakpoint
                b *0x0402850                            # break après la récupération de l'opcode
                c
                set $_op = $w0                          # sauvegarde opcode
                set $_instr = $w1                       # sauvegarde instruction complète
                printf "eip: %x   opcode: %x   instr: %.4x\n", $_eip, $_op, $_instr       # affichage
                x/16xw 0x4000812000                     # dump des registres

                d breakpoint
                b *0x0402860                            # break au retour de l'exécution de l'instruction
                c
                x/16xw 0x4000812000                     # re-dump des registres
end
```

- Il peut aussi être très utile de modifier cette fonction pour boucler tant qu'un type spécifique d'instruction n'est pas atteint - cela permet facilement d'étudier les différentes instructions, et voir le contenu des registres évoluer.

Format des instructions RISC1

- Voila l'intégralité de la description du format des instructions que l'on peut dénicher sur internet:

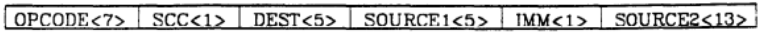


Figure 6. RISC I Basic Instruction Format

- Précis, non?
- Bon et en plus, la VM ne respecte pas exactement les spécifications… notamment sur la taille
- On a des instructions sur 32 bits qui ressemblent (la plupart du temps) à ca:

opcode	opt1 (imm_hi / rs1)	Rd	opt2 (imm_md)	opt3 (imm_lo)
8 bits	4 bits	4 bits	8 bits	8 bits

- ou à ça pour les jumps

opcode	cond	imm_hi	imm_lo
8 bits	8 bits	8 bits	8 bits

Format des instructions RISC1

- Et on a aussi des instructions sur 16 bits qui ressemblent à ça:

opcode	rd	rs
8 bits	4 bits	4 bits

- ou à ça:

opcode	imm
8 bits	8 bits

Liste des opcodes

Voici les instructions les plus importantes, accompagnées de leur opcode:

opcode	Instruction	opcode	Instruction	opcode	Instruction
0	mov rd, rs + imm	11	or rd, rs	23	dec rd
1	add rd, imm	12	and rd, rs	27	jmp reg / ret
2	ldr rd, [rs + imm]	13	lsl rd, rs	28	halt
4	ldrb rd, [rs + imm]	14	lsr rd, rs	29	int imm
7	strb rd, [rs + imm]	18	add rd, rs	30	??? mystère!
8	jxx imm_rel	19	sub rd, rs		
10	xor rd, rs	22	inc rd		

Désassemblage

Extraits du programme désassemblé:

```
0040>   00010000        movr    r1      0
0044>   01210000        addi    r1      2
0048>   00020000        movr    r2      0
004c>   01120000        addi    r2      1
0050>   00030000        movr    r3      0
0054>   01e33200        addi    r3      32e     //:: Please enter the decryption key:
0058>   00040000        movr    r4      0
005c>   01440200        addi    r4      24
0060>   1d00            int     0
```

```
007e>   02050000        ldr     r5      [r0]
0082>   00030000        movr    r3      0
0086>   01030100        addi    r3      10
008a>   1335            sub     r5      r3
008c>   086ab402        jmp.r5.nz       2b4
```

```
0190>   011d0000        addi    r13     1
0194>   02082400        ldr     r8      [r0 + 24]
0198>   02092800        ldr     r9      [r0 + 28]
019c>   0cc8            and     r8      r12
019e>   0cd9            and     r9      r13
01a0>   0a98            xor     r8      r9
```

Structure du programme

Maintenant désassemblé, le programme est assez simple à comprendre:

- Affichage de la chaîne “:: Please enter the decryption key:”
- Lecture de 16 octets à l’adresse 0x3fc
- Vérification du format de la clé
  - Si pas ok, affichage de “ Wrong key format.” puis halt
- Conversion des 16 octets hexadécimaux ascii en binaire et stockage à l’adresse 0x326
- Routine de déchiffrement
- Vérification du padding (doit finir par des zéros, le dernier caractère non nul doit être 0x80)
  - Si pas ok, affichage de “ Invalid padding.” puis halt
- Ouverture de payload.bin
  - Si pas ok, affichage de “ Cannot open file payload.bin.” puis halt
- Écriture du message décrypté dans payload.bin
- Fermeture du fichier
- Affichage du message “:: Decrypted payload written to payload.bin.”

Il nous reste maintenant à analyser la routine de déchiffrement.

Crypto!

L’algorithme de déchiffrement est relativement simple, une fois qu’on le couche sur papier.
Son fonctionnement est le suivant:

- Stocker la clé de chiffrement dans 2 registres de 32 bits (K1 et K2)
- Pour chaque octet d’input à chiffrer:
  - clé = 0
  - pour chaque bit de 7 à 0:
    - calculer la valeur du prochain bit de clé avec l’opération mystère “opcode 30”
      - bit = 1 si (K1 & 0xb0000000) ^ (K2 & 0x1) a un nombre de bits set impair
    - décalage de 1 bit vers la droite de K1
    - décalage de 1 bit vars la droite de K2
    - mettre le bit ‘perdu’ de K1 tout à gauche de K2
    - ajouter le bit le plus a droite de K2 à la clé
  - déchiffrer un octer du cipher text avec XOR clé

Crypto!

En python:

```
def xxx(key1, key2):
                lolz = (key1 & 0xb0000000) ^ (key2 & 0x1)
                bits = 0
                for i in range(32):
                                if lolz & (1 << i):
                                                bits += 1
                return bits % 2

def cipher(_key1, _key2, msg):
                key1, key2 = _key1, _key2
                dec = ''
                for i in range(len(msg)):
                                k = 0
                                for j in reversed(xrange(8)):
                                                b = xxx(key1, key2)

                                                key2 = (key2 >> 1) | ((key1 & 0x1) << 0x1f)
                                                key1 = (key1 >> 1) | (b << 0x1f)

                                                kb = key2 & 0x1

                                                k = k | ( kb << j)
                                dec += chr(ord(msg[i]) ^ k)
                return dec
```

La vulnérabilité

La combinaison de plusieurs éléments rend la crypto attaquable:

- Le chiffrement reviens à un xor avec une clé qui est dérivée progressivement
- La dérivation de la clé est bijective
- On connaît une partie du clear text (le padding + le dernier octet non nul)

Il est donc possible, si on suppose que les 8 derniers octets du clear text sont nuls, de connaître l'intégralité des bits de la clé de chiffrement 8 octets avant la fin.

Étant donné que la dérivation est bijective, on peut ensuite remonter à la clé valide d'origine.

Calcul de la clé de chiffrement

Les 8 derniers octets du cipher text sont:

```
6a b6 54 c3 ca 8f 53 02
```

Si on inverse les bits on obtient:

```
56 6d 2a c3 53 f1 ca 40
```

Ce qui nous donne, _après_ le déchiffrement valide du cipher text: `K1 = 0x40caf153 et K2 = 0xc32a6d56`
La fonction suivante nous permet de dériver les primitives de K1 et K2:

```
def goback1(k1, k2):
                b0k1 = (k1 & 0x80000000) >> 0x1f
                b0k2 = (k2 & 0x80000000) >> 0x1f

                k2 = (k2 << 1) & 0xffffffff
                k1 = (k1 << 1) & 0xffffffff
                k1 |= b0k2

                if xxx(k1, k2) != b0k1:
                                k2 |= 0x1

                return k1, k2
```

Victoire !! O_o ?

- Une fois l'algorithme vérifié, débuggé, et que la clé dérivé arrive à déchiffrer la fin du cipher text et nous donne 8 zéros, on dérive la clé 262080 fois `((0x8000 - 8) * 8)` et on obtient les valeurs suivantes:

```
orig k1: 0x05b1ad0b
orig k2: 0x11adde15
final key: 0BADB10515DEAD11
```

- On lance ensuite le programme avec ces valeurs (ou juste notre script python ^^) et magie, le fichier finit bien par plein de zéros, et le dernier octet non nul vaut bien 0x80! Bad bios is dead!!
- Victoire! célébration! on a fini le challenge!
- … Et non :)
- Le fichier n'est pas une image ou un texte nous congratulant mais un zip, avec encore une nouvelle étape

mcu.zip

- Le fichier décrypté payload.bin est en fait un zip, reconnaissable par l'entête 'PK'. On l'extrait et on trouve 2 fichiers:

```
# file payload.bin
payload.bin: Zip archive data, at least v2.0 to extract
# mv payload.bin payload.zip && unzip payload.zip
Archive:  payload.zip
  inflating: mcu/upload.py
  inflating: mcu/fw.hex
```

- Le fichier upload.py est un script python qui se connecte à l'adresse IP 178.33.105.197 sur le port 10101, et transfère le fichier fw.hex
- Ce fichier contient le commentaire suivant:

```
# 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)              /
```

- La zone "Secret memory area" attire immédiatement notre attention et on se doute que la clé du challenge s'y trouve!

Exécution de upload.py

- Voici la sortie générée par l'exécution du script upload.py (pour que le script fonctionne avec python 2 supprimer ",end='')" à la ligne 32):

```
# ./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.
```

- On déduit assez naturellement que le service auquel on se connecte est un émulateur de micro-controlleur et qu'on lui envoie le programme fw.hex.
- Ce dernier à l'air d'être assez simple, un genre de "hello world" qui affiche "Firmware v1.33.7 starting".
- On se pose aussi la question de savoir si le programme ou le service est responsable pour afficher les autres lignes comme le compte d'instructions et la ligne "Halting."
- Pour en avoir le coeur net, il va falloir décortiquer dans le fichier fw.hex

fw.hex

- Le fichier fw.hex est composé de 31 lignes ressemblant à:

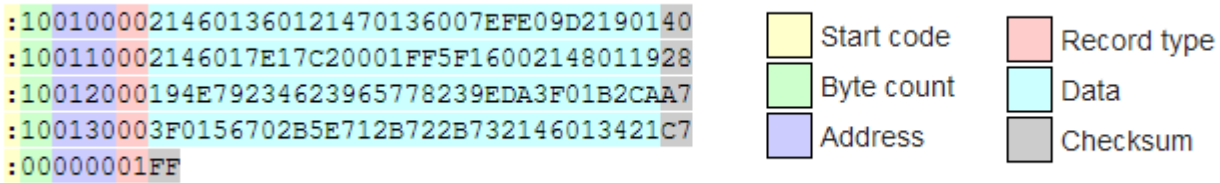
```
:100000002100111B2001108CC0D2201010002101F2
:10001000117C2200120FC03C20101000210111B2EF
```

- Les 2 dernières ligne sont un peu courtes:

```
:0C01D00072A77CE6D5A5680921D4410087
:00000001FF
```

- Ce format de fichier est appelé “intel hex” et est souvent utilisé pour programmer des microcontrolleurs

- Voici un extrait de la page wikipedia qui illustre bien ce format de fichier:
  - http://en.wikipedia.org/wiki/Intel_HEX



Extraction du binaire

- A l'aide de la doc wikipedia, on écrit vite fait bien fait un script python permettant de convertir un fichier intel hex en binaire et vice-versa.
- On obtient un ficher binaire de 476 octets, et un éditeur hexadécimal nous révèle les chaînes de caractères:

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

- **Après avoir apprécié la référence au meilleur film de tous les temps on lance prodigy-onelove.mp3 et on se lance!**
- Notre but est d'écrire un programme pour le micro-controlleur qui va nous afficher le contenu de la zone mémoire secrète (`0xf000 - 0xf800`)
- Mais pour ça, on a besoin de comprendre l'architecture et le format de ses instructions.
- On va encore écrire un désassembleur!
- Mais comment faire sans aucune documentation?

La bidouille

- On commence par regarder l'adresse de la string "Firmware starting" - 0x18c et on essaye de trouver un octet ayant la valeur 0x8c
- On en trouve un très vite à l'adresse 0x07 et on voit aussi un 0x01 à l'adresse 0x05
- C'est prometteur, ça sent la ou les instruction(s) pour mettre l'adresse 0x18c dans un registre afin d'afficher le message.

- A ce moment on peut faire une supposition éclairée, si on modifie l'octet 0x8c à l'offset 0x07 pour avoir la valeur 0x7c, on pourrait modifier le texte d'initialisation par YeahRiscIsGood! car cette chaîne est placée 16 octets plus tôt dans le fichier.

- Je n'ai en fin de compte jamais fait ce test, car une erreur de manipulation de mon éditeur hexadécimal (réécriture du premier octet par 00) a révélé une piste irrésistible:

```
-- 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.
```

  - Le système nous donne l'état des registres, ça va grandement nous aider!

Reverse engineering des instructions

- Étant donné l'indice RiscIsGood, les instructions ont probablement (on espère) une taille fixe de 16 ou 32 bits.
- On commence par nullifier tous les octets à partir de l'adresse 0x2
  - Si le programme plante à l'adresse 0x2, alors les instructions font 16 bits.
  - Si le programme plante à l'adresse 0x4, alors les instructions font 32 bits.
  - 16 bits! et les registres sont de 16 bits aussi, ca fait sens.
- Avec notre méthode de step by step, les 8 premiers octets du programme nous permettent d'identifier très vite deux instructions:
  - movr.hi qui met une valeur immédiate de 8 bits dans les 8 bits de poid fort du registre
  - movr.lo qui met une valeur immédiate de 8 bits dans les 8 bits de poid faible du registre

Hex	Résultat	Instruction		Hex	Résultat	Instruction
21 00	r1 = 0000	movhi r1, 0		20 01	r0 = 0100	movhi r0, 1
11 1b	r1 = 001b	movlo r1, 1b		10 8c	r0 = 018c	movlo r0, 8c

- On comprend aussi une partie du format des instructions:
  - Le nibble de poid fort du premier octet d'une instruction est l'opcode
  - Le nibble de poid faible du premier octet est le registre de destination
  - Le 2eme octet est utilisé comme valeur immédiate 8 bits

Reverse engineering des instructions

- Maintenant que l'on contrôle les valeurs de registres avec les opcodes 0x1 et 0x2, on continue nos déductions progressivement pour identifier les opérations arithmétiques standard comme add, sub, mul, div et les opérations binaires or, and, xor (valeurs 0x3 à 0x9).

- Le format de ces dernières, qui n'utilisent que des registres et aucune valeur immédiates est différent:

opcode	Rd	Rs1	Rs2
4 bits	4 bits	4 bits	4 bits

- Une fois ces instructions identifiées, on essaye de trouver les sauts, ce qui s'avère assez facile en regardant si la valeur de PC bouge ou pas.

- Au fur et à mesure que l'on arrive à décoder de plus en plus d'instructions on complète notre désassembleur et on le relance sur le programme hello world fw.hex.

- Cette approche itérative nous permet en fin de compte de décoder 16 types d'instructions

Liste des instructions

opcode	Instruction
1	mov.lo rd, imm8
2	mov.hi rd, imm8
3	add rd, rs1, rs2
4	xor rd, rs1, rs2
5	and rd, rs1, rs2
6	or rd, rs1, rs2
7	sub rd, rs1, rs2
8	mul rd, rs1, rs2
9	div rd, rs1, rs2

opcode	Instruction	Notes
a	j.cond imm8	a0 = jz a8 = jg ? a7 = jl ?
b	jmp imm8	imm8 positif si byte[0] = 0 imm8 negatif si byte[0] & 0x1
c0 c8	call imm8 syscall imm8	jump + set r15 to PC + 2 comme jump pour négatif/positif
d	j rs2	utilisé pour RET
e	ldrb rd, [rs1 + rs2]	rd = [rs1 + rs2]
f	strb rd, [rs1 + rs2]	[rs1 + rs2] = rd
0	invalid instr	

Secret Memory Area #1

- Suite à ces efforts, on est en mesure de comprendre le programme hello world entièrement et du coup d'identifier les syscalls
  - 1 = exit
  - 2 = print
  - 3 = get perf counter (compte d'instructions exécutées depuis le début du programme)
- On construit ensuite un programme simple ayant pour but de dumper le contenu de la zone de mémoire secrète.

`10 00`	`20 f0`	`11 00`	`21 08`	`c8 02`	`00 00`
`r0 = 0000`	`r0 = f000`	`r1 = 0000`	`r1 = 0800`	`call print`	`invalid instr`

- Output:

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

- Hé non :) pas si vite!

On continue

- La doc indiquait bien une séparation entre la zone “userland” et la zone “privilégiée” ou “kernel”.
- Notre objectif est donc de trouver un moyen de changer notre contexte d’exécution - ou élever nos droits - afin de pouvoir accéder à cette satanée mémoire protégée.

- Il doit y avoir dans le kernel (rom) une faille de sécurité nous permettant d’y parvenir, si seulement on avait accès au code du kernel…
  - Essayons notre script précédent, mais en spécifiant l’adresse 0xfd00 et la longueur 0x300
  - Ca marche! on récupère plein de binaire!
  - On peut maintenant désassembler et analyser le code du kernel.

- Tant qu’on y est, on va aussi dumper la plage 0xfc00 - 0xfcff nommée “HW registers”
  - On y trouve les registres et flags de notre programme.

- Voyons voir ce que le kernel a dans le ventre maintenant.

Analyse du kernel

- Nous lançons notre script de désassemblage sur le code du kernel fuité et analysons le tout:
  - Le début du code du kernel vérifie d'abord si r0 est égal à 0.
    - Si c'est le cas, la chaine "System reset" est affichée, et la VM est redémarrée.
  - r0 est ensuite comparé à 3
    - Si r0 est supérieur, alors afficher "unidentified syscall" et stopper l'exécution
  - Le kernel met ensuite dans r0 la valeur de 0xf000 + (syscall_index -1) * 2
  - La fonction 0xb0 est ensuite appelée
    - Il s'agit d'une fonction de récupération de la mémoire, qui retourne dans r0 la valeur contenue à l'adresse de r0
  - Ensuite le kernel exécute l'instruction "**d0 00**" qui correspond à **JMP r0**

- On a trouvé le code de dispatch des appels systèmes, et la "syscall table" est située tout au début de la RAM kernel à l'adresse 0xF000.

- Si on arrive à modifier le contenu de la mémoire de cette table avec un adresse pointant vers notre code et que l'on déclenche ensuite l'appel système correspondant, on va pouvoir faire exécuter notre code au kernel!

Write-what-where

- On part à la recherche d'une primitive de type "write what where" qui nous permettrait d'écrire une valeur choisie à une adresse donnée.
- La surface d'attaque est limitée, il n'existe que 4 syscalls valide:
  - le numéro 1 correspond à EXIT
  - le numéro 2 est PRINT
  - le numéro 0 correspond à RESET
    - Notre programme est redémarré, mais le contenu de la RAM reste inchangé
    - Un effet secondaire qui pique notre curiosité est le fait que si on aboutit à une boucle infinie, le programme va planter en mode kernel après un certain temps.
    - J'ai perdu un peu de temps en pensant que mon code était ré-exécuté avec les privilèges.
- Il ne reste donc plus qu'un seul syscall, le numéro 3, celui qui permet de demander au kernel combien d'instructions ont été exécutées depuis le début du programme.
  - On lui fournit une adresse mémoire et il la remplit..
  - Serait-il possible qu'il n'y ait aucune vérification sur l'adresse fourni?
    - OUI!
  - Nous pouvons de plus contrôler - dans certaines limites - la valeur qui est écrite!
- On a maintenant toutes les cartes en main pour subvertir le système!

Ecriture arbitraire de la mémoire

- Afin de pouvoir écrire exactement ce que l'on veut dans la mémoire protégée, il est nécessaire d'étudier d'un peu plus près les limites de notre primitive write-what-where.

- On se rend compte assez vite que le nombre de cycles cpu est borné entre 0x07c0 et 0x4FFF.

- Il ne va pas être possible d'écrire directement une valeur n'étant pas incluse dans ces bornes.
  - Cela dit, on contrôle entièrement le dernier octet, de 00 à FF

- Afin de pouvoir écrire 16 bits de n'importe quelle valeur à une adresse donnée A, on va décomposer notre écriture en 2 fois, en controllant uniquement le dernier octet, et en décrémentant l'adresse de 1 pour la 2eme écriture.
  - exemple: on veut écrire 0xDEAD à l'adresse 0x1234
  - écrire 0xNNAD à 0x1234
  - écrire 0XNNDE à 0x1233
- Afin d'avoir un retour sur nos écritures mémoire et de perfectionner notre attaque, on cible les registres HW.
  - ex: Le registre R7 est situé à l'adresse 0xFC2E.
- On a maintenant tout ce qui est nécessaire pour exploiter le service!

Algorithme d’attaque finale

```
0000: print [0x1000 - 0x1C00]                    ; affiche 0x0C00 zéros à la première exécution - mémoire secrète la 2e

0010: while( instr_count & 0xff != 0x00)         ; faire en sorte que le dernier octet de instr_counter soit null
    nop
0030: syscall_table[sys_exit] = instr_count      ; écrire le compteur d’instruction à l’adresse du pointeur de sys_exit

0050: while( instr_count & 0xff != 0x01)         ; faire en sorte que le dernier octer de instr_counter soit 0x1
    nop
0070: syscall_table[sys_exit - 1] = instr_count  ; syscall_table[sys_exit] == 0x100

0080: call syscall_exit                          ; déclencher sys_exit, ce qui appelle notre code à 0x0100 en mode kernel

00b0: while(r2 > 0)                              ; implémentation de memcpy
    [r0++] = [r1++]
    r2 = r2 - 1

0100: call 00b0 (0x1000, 0xF000, 0x0C00)         ; copie de la mémoire secrète dans la RAM userland
      syscall 0                                  ; reset de la VM, notre programme est rappelé, mais la RAM contient la
                                                   mémoire secrète
```

Exploit

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	10	00	20	20	11	00	21	0c	c8	02	10	00	10	00	10	00	.. ..!.È.....
00000010	10	50	20	05	11	01	21	00	70	01	a0	14	b3	fa	00	00	.P ...!.p...³ú..
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	10	00	20	f0	10	00	c8	03	b0	16	00	00	00	00	00	00	.. ð..È.°.....
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000050	10	3f	20	00	11	01	21	00	70	01	a0	14	b3	fa	00	00	.? ...!.p...³ú..
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000070	10	00	10	00	10	00	20	ef	10	ff	c8	03	b0	12	00	00	..... ï.ÿÈ.°...
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000090	10	a0	20	00	11	00	21	00	c8	01	00	00	00	00	00	00	.. ...!.È.....
000000a0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000b0	13	00	23	00	14	01	24	00	e5	13	f5	03	63	34	72	24	..#...$.å.õ.c4r$
000000c0	a0	02	b3	f4	d0	0f	00	00	00	00	00	00	00	00	00	00	..³ôÐ.....
000000d0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000e0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000f0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000100	10	00	20	20	11	00	21	f0	12	00	22	0c	c3	a2	10	00	.. ..!ð.."Ã¢..
00000110	11	00	21	0c	c8	00	00	00	00	00	00	00	00	00	00	00	..!.È.....

Output: WOW!

ÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÇÔûêÔûêÔûÇÔûÆÔûî WOW

ÔöÇÔöÇÔûÉÔûÆÔûÆÔûÆÔûäÔûäÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÇÔûäÔûÆÔûÆÔûî

ÔöÇÔöÇÔûîÔûÆÔûÆÔûÉÔûäÔûêÔûÇÔûÆÔûÆÔûÆÔûÆÔûäÔûÇÔûêÔûäÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûêÔûÆÔûÉ SUCH EXPLOIT

ÔöÇÔûÉÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûîÔûêÔûêÔûÇÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÇÔûäÔûî

ÔöÇÔûîÔûÆÔûÇÔûäÔûêÔûêÔûäÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûæÔûæÔûæÔûÆÔûÆÔûÆÔûÆÔûî VERY CHALLENGING

ÔöÇÔûîÔûÇÔûÉÔûäÔûêÔûäÔûêÔûîÔûäÔûÆÔûÇÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûæÔûæÔûæÔûæÔûæÔûÆÔûÆÔûÆÔûÉ

ÔûÉÔûÆÔûÇÔûÉÔûÇÔûÉÔûÇÔûÆÔûÆÔûäÔûäÔûÆÔûäÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûæÔûæÔûæÔûæÔûæÔûæÔûÆÔûÆÔûÆÔûÆÔûî SO OPERATIONAL

ÔûÉÔûÆÔûÆÔûÆÔûÇÔûÇÔûäÔûäÔûÆÔûÆÔûÆÔûäÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûæÔûæÔûæÔûæÔûæÔûæÔûÆÔûÆÔûÆÔûÉ

ÔöÇÔûîÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÇÔûÇÔûÇÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûæÔûæÔûæÔûæÔûÆÔûÆÔûÆÔûî MUCH WIN

ÔöÇÔûÉÔûÆÔûÉ

ÔöÇÔöÇÔûÇÔûäÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûäÔûÆÔûÆÔûÆÔûÆÔûî <66a65dc050ec0c84cf1dd5b3bbb75c8c@challenge.sstic.org>

ÔöÇÔöÇÔöÇÔöÇÔûÇÔûäÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûÆÔûäÔûäÔûÇÔûÇÔûÆÔûÆÔûÆÔûäÔûÇ

ÔöÇÔöÇÔöÇÔûÉÔûÇÔ

Conclusion

On a réussi!

L’adresse e-mail est 66a65dc050ec0c84cf1dd5b3bbb75c8c@challenge.sstic.org

Un grand merci pour l’aventure, c’était très fun de reverser un packer crypto-code-virtualizer, de découvrir l’arm64, d’écrire 2 désassembleurs RISCs et de finir en beauté en écrivant un remote exploit directement en hexadécimal!

A l’année prochaine,

- Martin Balc’h

martin@balch.fr