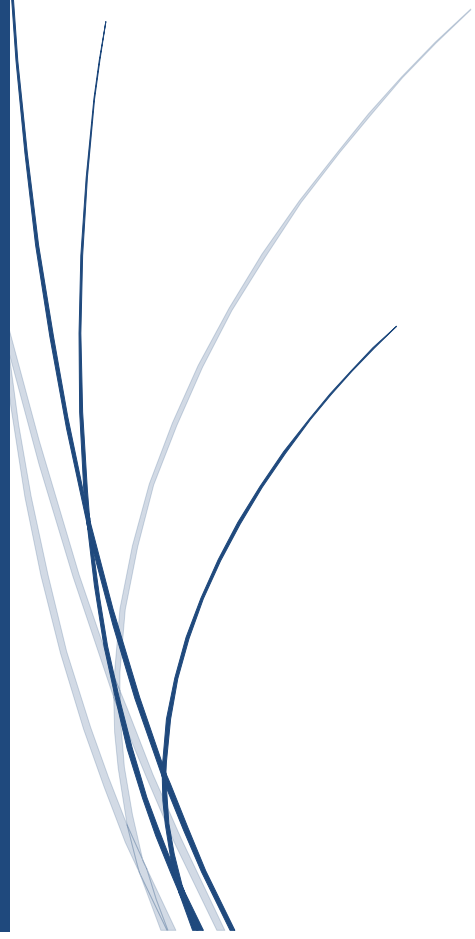
29/04/2014

# Challenge SSTIC 2014



Jean Sigwald

# Table des matières

# Introduction

Le challenge commence avec un fichier texte accompagné d'une trace de communication usbmon[1] au format "texte".

```
$ file usbtrace.xz
usbtrace.xz: XZ compressed data

$ xz -d usbtrace.xz

$ less usbtrace

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

Bonjour,

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

--

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
...
```

Il ne semble pas exister d'outil permettant de convertir une trace usbmon "texte" au format binaire supporté par Wireshark. On peut décoder simplement la trace en ne gardant que les données hexadécimales sur chaque ligne après le signe "=". Le script 01_extract_badbios.py stocke le résultat de cette opération dans le fichier usbtrace.bin.

```
$ strings usbtrace.bin
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:s0OKAY
shell:uname -a
Linux localhost 4.1.0-g4e972ee #1 SMP PREEMPT Mon Feb 24 21:16:40 PST 2015
armv8l GNU/LinuxOKAY
/data/local/tmp/badbios.binOKAY
/data/local/tmp/badbios.bin,33261DATA
shell:chmod 777 /data/local/tmp/badbios.bin
```

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

La trace correspond donc à l'exécution de différentes commandes shell sur un terminal Android via le protocole ADB (Android Debug Bridge). Un exécutable "badbios.bin" est également transféré. Le résultat de la commande *uname* nous indique qu'il s'agit d'un terminal (fictif?) ARMv8 (ARM 64 bits).

Le format des messages ADB est détaillé dans les sources d'Android[2] [3]. Le script 01_extract_badbios.py décapsule tous les messages de type WRTE, puis extrait les 2 réponses DATA correspondant à la récupération du fichier badbios.bin. On obtient un binaire ELF64[4].

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

# Binaire ARM64 badbios.bin

## Émulation ARM64

Pour étudier le binaire il est souhaitable de pouvoir le lancer et le déboguer. Dans un premier temps, j'ai suivi les instructions du wiki Debian[5] pour installer une version de Qemu qui permet de lancer le binaire :

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

La version actuelle de Qemu ne fait que de l'émulation user-land (traduction du code binaire et interception des syscalls) et ne supporte pas l'émulation d'un système ARM64 complet. Dans notre cas, cela nous empêche de pouvoir déboguer et tracer le programme avec gdb.

Cependant, il existe un émulateur de système complet : "ARMv8 Foundation Platform", disponible en s'inscrivant gratuitement sur le site d'ARM[6]. La documentation sur le wiki d'Ubuntu[7] permet d'installer rapidement un système Linux ARM64 complet avec cet émulateur.

Le programme badbios.bin demande une clé de chiffrement de 64 bits sous forme de 16 caractères hexadécimaux (en majuscules) et tente de déchiffrer un "payload". Il va donc falloir

---

[2] https://android.googlesource.com/platform/system/core/+/master/adb/protocol.txt

[3] https://android.googlesource.com/platform/system/core/+/master/adb/SYNC.TXT

[4] http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf

[5] https://wiki.debian.org/Arm64Qemu

[6] http://www.arm.com/products/tools/models/fast-models/foundation-model.php

[7] https://wiki.ubuntu.com/ARM64/FoundationModel

reverser le binaire pour trouver la bonne clé : j'ai utilisé IDA 6.5, gdb sous Ubuntu ARM64, et la documentation de référence ARMv8[8].

## "Unpacking" du binaire

Bien qu'il ne permette pas de déboguer, Qemu possède l'option -strace qui permet d'afficher tous les appels système effectués :

```
$ qemu-arm64-static -strace badbios.bin
mmap(0x0000000000400000,12288,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP
_FIXED,0,0) = 0x0000000000400000
mprotect(0x0000000000400000,12288,PROT_EXEC|PROT_READ) = 0
mmap(0x0000000000500000,69632,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP
_FIXED,0,0) = 0x0000000000500000
mprotect(0x0000000000500000,69632,PROT_READ|PROT_WRITE) = 0
mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000801000
mmap(NULL,65536,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000802000
mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000812000
mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000813000
write(1,0x813000,36):: Please enter the decryption key:  = 36
munmap(0x0000004000813000,36) = 0
mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000814000
read(0,0x814000,16)AAAA

munmap(0x0000004000814000,16) = 0
mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0) =
0x0000004000815000
write(2,0x815000,21)   Wrong key format.

munmap(0x0000004000815000,21) = 0
exit_group(0)
```

En cherchant les instructions *SVC 0* (Supervisor Call) dans le binaire, on peut identifier l'endroit où sont effectués les différents syscalls : le numéro du syscall est passé dans le registre X8. Le binaire badbios.bin ne contient que des appels aux syscalls mmap (0xDE), mprotect (0xE2) et exit (0x5E). Le code qui nous intéresse, l'affichage des messages et la lecture de la clé, n'est pas directement identifiable.

La trace nous permet de voir que le programme commence par mapper 2 zones mémoires :

- 0x400000 : taille 0x3000, protection PROT_EXEC|PROT_READ
- 0x500000 : taille 0x11000, protection PROT_READ|PROT_WRITE

En fait, badbios.bin charge les 2 segments d'un second binaire ELF embarqué dans son segment de données puis continue l'exécution à l'adresse 0x400514.

---

[8] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.b/index.html

Le segment de code chargé en 0x400000 est décompressé par la fonction sub_10304. Cette fonction n'a pas été étudiée. Le script 02_dump_badbios2.sh utilise gdb pour dumper les 2 segments en mémoire avant le saut vers l'adresse 0x400514 (l'entry point du binaire décompressé).

On obtient un binaire ELF avec tous les champs du Program Header à 0. Le script 03_fix_badbios2_pheader.py remplit les champs du Program Header de manière à pouvoir lancer le binaire, le charger dans IDA et gdb :

```
$ ./03_fix_badbios2_pheader.py
$ readelf -l badbios2.bin

Elf file type is EXEC (Executable file)
Entry point 0x400514
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000000000
                 0x0000000000003000 0x0000000000003000  R E    10000
  LOAD           0x0000000000003000 0x0000000000500000 0x0000000000000000
                 0x0000000000011000 0x0000000000011000  RW     10000
```

## Machine virtuelle

Les seules chaînes de caractères présentes dans le binaire badbios2.bin sont :

```
No error.
Bad instruction pointer.
Invalid instruction.
Memory fault.
Internal error.
Invalid argument.
Out of memory.
```

Ce qui laisse penser qu'il s'agit d'une machine virtuelle. Les messages affichés par le programme (par exemple "Please enter the decryption key") ne sont pas présents en clair dans le binaire.

En étudiant les fonctions sub_0400408 et sub_004000E8, on identifie la chaîne de caractères "expand 16-byte k". En cherchant sur Google on tombe sur l'algorithme de chiffrement par flot salsa20[9] de D. J. Bernstein. Cependant, on voit que la fonction sub_004000E8 fait 8 tours de "xors", et que les opérations ne correspondent pas à celles de salsa20. Il s'agit en fait de l'algorithme ChaCha[10], une variante de salsa20.

En posant un breakpoint sur la fonction sub_004000E8, on observe que le programme déchiffre les données situées à l'adresse 0x500000 par blocs de 64 octets. Il s'agit en fait de la

[9] http://cr.yp.to/snuffle.html

[10] http://cr.yp.to/chacha.html

"RAM" de la machine virtuelle qui est déchiffrée à la volée par la "MMU" qui gère des "pages" de 64 octets (sub_0004020C4).

La clé utilisée est en clair à l'adresse 0x510000 :

```
LOAD:0510000 key_chacha DCB 0xB, 0xAD, 0xB1, 5, 0xB, 0xAD, 0xB1, 5, 0xB, 0xAD
LOAD:0510000                DCB 0xB1, 5, 0xB, 0xAD, 0xB1, 5
```

Le script 04_decrypt_vm_memory.py utilise une implémentation Python de l'algorithme ChaCha[11] pour déchiffrer la mémoire RAM de la VM. On y retrouve bien les chaînes de caractères affichées par le programme.

Il faut à présent identifier les différentes instructions de la machine virtuelle et écrire un désassembleur pour comprendre comment est utilisée la clé saisie.

La structure représentant l'état de la machine virtuelle est la suivante (05_badbios.h) :

```
typedef struct
{
    uint32_t flags;
    uint32_t error_code;
    uint64_t field_unk;

    chacha_state chacha_ctx;//+0x10

    //+0x50
    uint8_t* mmaped_encrypted_ram;//sizeof=0x10000

    //+0x58
    page_table_entry page_table[0x20];//0x20*0x18=0x300

    //+0x358
    uint8_t* decrypted_ram_view;//size=0x1000 => 0x0000007fb7fed000

    //+0x360
    void* instruction_handlers[0x1F];//31 instructions

    //+0x458
} badbios_context;
```

Le tableau à l'offset +0x360 contient les adresses des gestionnaires pour les 31 instructions. Cette table est construite de manière obfusquée par la fonction sub_401A08. Le script 06_dump_vm_context.sh sauvegarde la structure badbios_context après son initialisation, ce qui permet d'identifier les gestionnaires des différentes instructions dans IDA (script 07_rename_vm_handlers_ida.py).

[11] http://www.seanet.com/~bugbee/crypto/chacha/chacha.py

Les principales fonctions du binaire badbios2.bin sont les suivantes :

| Adresse | Fonction |
|---|---|
| 04000B0 | main |
| 0402960 | init_vm |
| 0400408 | init_chacha_ctx |
| 0401A08 | init_vm_handlers |
| 0402754 | main_vm_loop |
| 04022AC | read_memory |
| 04025F4 | read_register |
| 04027BC | get_pc |
| 04025F4 | write_memory |
| 04025F4 | write_register |
| 04025F4 | read_register |
| 040243C | set_ctx_error |
| 04020C4 | memory_page_in |
| 04000E8 | decrypt_chacha |

La fonction main_vm_loop contient la boucle principale de décodage et d'exécution des instructions de la machine virtuelle. Les instructions sont codées sur 2 ou 4 octets selon la valeur de l'opcode (premier octet).

Les registres de 32 bits sont mappés en mémoire à l'adresse 0x0. Le compteur d'instruction (r15) est situé à l'adresse 15*4=60 (0x3C).

Un désassembleur minimal est implémenté dans le script 08_vm_disas.py.

Le pseudo code du programme exécuté par la machine virtuelle est le suivant :

```
syscall(write, stdout, ":: Please enter the decryption key: ")
if(syscall(read, stdin, &buf_0x03fc) != 0x10)
    return syscall(write, stdout, "Wrong key format.")
convert hex string to binary
syscall(write, stdout, ":: Trying to decrypt payload...")
decrypt payload
check padding
if padding_ok
    fd = open(payload.bin)
    write(fd, decrypted_payload)
else
    syscall(write, stdout, "bad padding")
```

Le code à partir l'adresse 0x0194 correspond au déchiffrement d'un payload de 0x2000 octets situé à l'adresse 0x8000 dans la mémoire de la VM.

```
$ ./08_vm_disas.py
...
    0x0152 : mov          00354301    or r3, 0x0354   :: Trying to decrypt
payload...
    0x0156 : op_00        00000400    mov r4, 0x0
    0x015a : mov          00020401    or r4, 0x0020
    0x015e : syscall      0000001d
    0x0160 : op_00        00000100    mov r1, 0x0
    0x0164 : mov          00326101    or r1, 0x0326   ; hex2bin result buffer
    0x0168 : mov w[reg]   00001a02    mov r10, w[r1 + 0x0]  ;r10 = key[0:4]
    0x016c : mov w[reg]   00041b02    mov r11, w[r1 + 0x4]  ;r11 = key[4:8]
    0x0170 : xor          0000110a    xor r1, r1
    0x0172 : op_00        00000200    mov r2, 0x0
    0x0176 : mov          08000201    or r2, 0x8000   ; encrypted payload
    0x017a : op_00        00000300    mov r3, 0x0
    0x017e : mov          00008301    or r3, 0x0008
    0x0182 : xor          0000440a    xor r4, r4
    0x0184 : op_00        0b000c00    mov r12, 0x0
    0x0188 : mov          00000c01    or r12, 0x0000
    0x018c : op_00        00000d00    mov r13, 0x0
    0x0190 : mov          00001d01    or r13, 0x0001
decrypt_loop:
    0x0194 : mov w[reg]   00240802    mov r8, r10
    0x0198 : mov w[reg]   00280902    mov r9, r11
    0x019c : and          0000c80c    and r8, r12
    0x019e : and          0000d90c    and r9, r13
    0x01a0 : xor          0000980a    xor r8, r9
    0x01a2 : xor283930    0000891e    lfsr r9, r8
    0x01a4 : op_00        00000800    mov r8, 0x0
    0x01a8 : mov          00001801    or r8, 0x0001
    0x01ac : op_00        00000700    mov r7, 0x0
    0x01b0 : mov          0001f701    or r7, 0x001f
    0x01b4 : mov w[reg]   00240602    mov r6, r10
    0x01b8 : and          0000860c    and r6, r8
    0x01ba : lsl          0000760d    lsl r6, r7
    0x01bc : lsr          00008b0e    lsr r11, r8
    0x01be : or           00006b0b    or r11, r6
    0x01c0 : lsr          00008a0e    lsr r10, r8
    0x01c2 : lsl          0000790d    lsl r9, r7
    0x01c4 : or           00009a0b    or r10, r9
    0x01c6 : dec          00000317    dec r3
    0x01c8 : mov w[reg]   00280702    mov r7, r11
    0x01cc : and          0000870c    and r7, r8
    0x01ce : lsl          0000370d    lsl r7, r3
    0x01d0 : or           0000740b    or r4, r7
    0x01d2 : jmp          01f66608    3 3 decrypt_loop2
    0x01d6 : op_00        00000700    mov r7, 0x0
    0x01da : mov          08000701    or r7, 0x8000
    0x01de : add          00001712    add r7, r1
    0x01e0 : mov b[reg]   00007804    mov r8, byte[r7]
    0x01e4 : xor          0000480a    xor r8, r4
```

```
    0x01e6 : mov_mem      00007807    mov b[r7 + 0x0], r8
    0x01ea : op_00        00000300    mov r3, 0x0
    0x01ee : mov          00008301    or r3, 0x0008
    0x01f2 : inc          00000116    inc r1
    0x01f4 : xor          0000440a    xor r4, r4
decrypt_loop2:
    0x01f6 : op_00        00000800    mov r8, 0x0
    0x01fa : mov          02000801    or r8, 0x2000
    0x01fe : cmp          00001813    cmp r8, r1
    0x0200 : jmp          0194b008    0 1 decrypt_loop
end_decrypt_loop:
```

La clé entrée par l'utilisateur est en fait utilisée pour initialiser l'état d'un registre à décalage (LFSR[12]) de 64 bits, stocké dans les registres r10 et r11. L'opcode 0x1E calcule le bit de sortie du registre en xorant les bits 60, 61 et 63 (+ le bit 0 qui est récupéré par l'instruction *and r9, r13*).

```
LOAD:040077C instr_1e_xor_bits
...
LOAD:04007B0    BL          read_register
LOAD:04007B4    EOR         W0, W0, W0,LSR#1
LOAD:04007B8    EOR         W1, W0, W0,LSR#2
LOAD:04007BC    AND         W2, W1, #0x11111111
LOAD:04007C0    MOV         W0, #0x11111111
LOAD:04007C4    MADD        W2, W2, W0, WZR   ;w2 = w2 + w0*0 (nop)
LOAD:04007C8    UBFM        X1, X19, #8, #11
LOAD:04007CC    ORR         X0, X20, X20
LOAD:04007D0
LOAD:04007D0 loc_4007D0                       ; DATA XREF: init_vm_state+3EC
LOAD:04007D0                                  ; init_vm_state+3F0
LOAD:04007D0    CSEL        X2, X2, X30, AL   ;conditional select ALways => x2=x2
(nop)
LOAD:04007D4    LDP         X19, X20, [SP,#-0x10+arg_20]
LOAD:04007D8    NOP
LOAD:04007DC    LDP         X29, X30, [SP-0x10+arg_10],#0x20
LOAD:04007E0    UBFM        X2, X2, #28, #28
LOAD:04007E4    ADD         X4, X4, #0
LOAD:04007E8    B           write_register
```

Les bits de sortie de ce registre à décalage sont xorés avec le payload chiffré.

[12] http://en.wikipedia.org/wiki/Linear_feedback_shift_register

## Calcul de la clé de chiffrement

Pour que le payload déchiffré soit considéré correct il faut que les 8 derniers octets soit nuls. Il faut donc que l'état du LFSR à cet endroit soit égal aux 8 derniers octets chiffrés. Comme la fonction de transition du LFSR est connue, il suffit de l'inverser et de "remonter" à l'état initial (qui correspond à la clé entrée) en partant de l'état final souhaité.

```
def bit(r, b): return (r >> b) & 1

def push_bit_right(r, b):
    return (r >> 1) | (b << 63)

def push_bit_left(r, b):
    return ((r << 1) | b) & 0xffffffffffffffff

#lfsr taps
#0xb000000000000001 => #0b1011000000000000000000000000000000
def step(state):
    r9 = bit(state, 63) ^ bit(state, 61) ^ bit(state, 60) ^ bit(state, 0)
    return push_bit_right(state, r9)

#inverse lfsr
def rstep(state):
    r9 = bit(state, 62) ^ bit(state, 60) ^ bit(state, 59) ^ bit(state, 63)
    return push_bit_left(state, r9)

#...
payload = open("decrypted_vm_memory.bin", "rb").read()[0x8000:0x8000+0x2000]

target_output = payload[-8:]
state = binary_reverse64(struct.unpack(">Q", target_output)[0])
print "Target end state: 0x%x" % state

for i in xrange((0x2000-8) * 8 + 1):
    state = rstep(state)
print "Initial LFSR state = 0x%x " % state
print "Key => %s" % struct.pack("<Q", state).encode("hex").upper()
```

Le script 09_lfsr.py retrouve la clé et déchiffre le payload.

```
$ ./09_lfsr.py
Target end state: 0x40caf153c32a6d56
Initial LFSR state = 0x5b1ad0b11adde15
Key => 15DEAD110BADB105
Writing decrypted payload to payload.zip

$ file payload.zip
payload.zip: Zip archive data, at least v2.0 to extract
```

La clé "15DEAD110BADB105" permet de déchiffrer correctement le payload, qui est en fait une archive zip contenant la seconde partie du challenge.

# Microcontrôleur

## Architecture CPU inconnue

L'archive zip contient 2 fichiers :

```
$ unzip payload.zip
Archive:  payload.zip
  inflating: mcu/upload.py
  inflating: mcu/fw.hex
```

Le script upload.py (Python 3 !) envoie le fichier fw.hex au serveur 178.33.105.197:10101 qui l'exécute.

```
$ ./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.
```

Un commentaire dans le script nous explique cette partie du challenge :

```
#
# 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)              /
#
```

Il va donc falloir déterminer l'architecture du microcontrôleur émulé par le serveur, et trouver un moyen de lire la zone "Secret memory area".

Le firmware fw.hex envoyé au serveur est au format texte Intel HEX[13]. La bibliothèque Python IntelHex peut être utilisée pour décoder le fichier.

[13] http://en.wikipedia.org/wiki/Intel_HEX

```
$ sudo easy_install intelhex
$ hex2bin.py mcu/fw.hex fw.bin
$ strings fw.bin
eVUS
R%s&
0b233
YeahRiscIsGood!
Firmware v1.33.7 starting.
Halting.
```

On remarque que les chaînes "System reset." et "Execution completed in 8339 CPU cycles." ne sont pas présentes dans le firmware. L'analyse commence en observant le firmware sous forme binaire.

```
$ xxd fw.bin
0000000: 2100 111b 2001 108c c0d2 2010 1000 2101  !... ..... ...!.
...
0000170: 1830 6882 f803 5444 a7de d00f 5965 6168  .0h...TD....Yeah
0000180: 5269 7363 4973 476f 6f64 2100 4669 726d  RiscIsGood!.Firm
0000190: 7761 7265 2076 312e 3333 2e37 2073 7461  ware v1.33.7 sta
00001a0: 7274 696e 672e 0a00 4861 6c74 696e 672e  rting...Halting.
00001b0: 0a00 942b 506f ae0c bb1f 39b4 d8ca 05fd  ...+Po....9.....
00001c0: 8a0f 5ae8 b5d4 0d6c e86a a6ac c492 f8f1  ..Z....l.j......
00001d0: 72a7 7ce6 d5a5 6809 21d4 4100            r.
```

On voit que la chaîne "Firmware v1.33.7 starting." est située à l'offset 0x18C et fait 0x1B octets. Les 8 premiers octets du binaire contiennent ces 2 valeurs :

```
0000000: 2100 111b 2001 108c c0d2 2010 1000 2101  !... ..... ...!.
           ^^   ^^   ^^   ^^
           0x001B     0x018C
```

On peut donc essayer de modifier ces valeurs et de soumettre le firmware modifié au serveur pour tenter d'afficher la zone mémoire secrète ou le kernel. Le script 10_upload2.py permet d'envoyer un fichier binaire au serveur.

```
$ diff <(xxd fw.bin) <(xxd fw_print_secret.bin)
1c1
< 0000000: 2100 111b 2001 108c c0d2 2010 1000 2101  !... ..... ...!.
---
> 0000000: 210c 1100 20f0 1000 c0d2 2010 1000 2101  !... ..... ...!.

$ ./10_upload2.py fw_print_secret.bin
:: Serial port connected.
:: Uploading firmware...
done.
00000000 | 53 79 73 74 65 6D 20 72 65 73 65 74 2E 0A 5B 45 | System reset..[E
00000010 | 52 52 4F 52 5D 20 50 72 69 6E 74 69 6E 67 20 61 | RROR] Printing a
00000020 | 74 20 75 6E 61 6C 6C 6F 77 65 64 20 61 64 64 72 | t unallowed addr
00000030 | 65 73 73 2E 20 43 50 55 20 68 61 6C 74 65 64 2E | ess. CPU halted.
00000040 | 0A                                              | .
```

Il n'est pas possible de récupérer directement la zone mémoire secrète de cette façon, par contre il est possible d'afficher la zone correspondant au kernel ([FD00-FFFF]).

```
$ diff <(xxd fw.bin) <(xxd fw_print_kernel.bin)
1c1
< 0000000: 2100 111b 2001 108c c0d2 2010 1000 2101  !... ..... ...!.
---
> 0000000: 2103 1100 20fd 1000 c0d2 2010 1000 2101  !... ..... ...!.

$ ./10_upload2.py fw_print_kernel.bin
:: Serial port connected.
:: Uploading firmware...
done.
00000000 | 53 79 73 74 65 6D 20 72 65 73 65 74 2E 0A 50 00 | System reset..P.
00000010 | A0 6C 21 00 11 03 72 10 A8 12 22 00 12 02 81 02 | .l!...r..."".....
00000020 | 71 12 20 F0 10 00 60 01 C0 94 D0 00 21 00 11 2B | q. ...`.....!..+
00000030 | 20 FE 10 5A C0 BE 30 00 21 FC 11 10 22 00 12 01 |  ..Z..0.!..."...
00000040 | F2 10 B3 F2 20 FC 10 22 C0 74 55 00 20 FC 10 20 | .... .."".tU. .. 
00000050 | C0 6C 51 55 C0 9E D8 00 20 FC 10 20 C0 60 26 FC | .lQU.... .. .`&.
00000060 | 16 12 21 00 11 01 34 44 E5 61 E2 64 E3 64 73 32 | ..!...4D.a.d.ds2
00000070 | A7 F6 23 01 13 00 82 23 41 25 C0 56 D8 00 21 00 | ..#....#A%.V..!.
00000080 | 11 0E 20 FE 10 86 C0 6C 24 00 14 02 21 FD 11 28 | .. ....l$...!..(
00000090 | 20 F0 10 00 C0 3C 60 04 21 FD 11 36 C0 34 60 04 |  ....<`.!..6.4`.
000000A0 | 21 FD 11 4A C0 2C 20 FC 10 20 31 11 22 00 12 36 | !..J., .. 1."..6
000000B0 | C0 32 20 FC 10 3A 21 EF 11 FE C0 16 D8 00 21 00 | .2 ..:!.......!.
000000C0 | 11 01 22 01 12 00 E3 01 71 11 E4 01 84 42 40 34 | .."".....q....B@4
000000D0 | D0 0F 22 00 12 01 23 01 13 00 F1 02 72 22 91 13 | .."...#.....r"..
000000E0 | F1 02 D0 0F 23 00 13 01 52 22 A0 06 72 23 F1 02 | ....#...R"..r#..
000000F0 | B3 F2 D0 0F 5E 00 2D FC 1D 00 2C F0 1C 00 38 88 | ....^.-...,...8.
00000100 | 59 88 2A 00 1A 01 3B BB 51 11 A0 1A 69 E8 79 9C | Y.*...;.Q...i.y.
00000110 | A8 08 69 E8 79 9D AC 02 B0 0E 39 99 E9 E8 F9 DB | ..i.y.....9.....
00000120 | 68 8A 71 1A B3 E2 D0 0F 21 00 11 33 20 FE 10 26 | h.q.....!..3 ..&
00000130 | C3 C2 B3 02 5B 45 52 52 4F 52 5D 20 50 72 69 6E | ....[ERROR] Prin
00000140 | 74 69 6E 67 20 61 74 20 75 6E 61 6C 6C 6F 77 65 | ting at unallowe
00000150 | 64 20 61 64 64 72 65 73 73 2E 20 43 50 55 20 68 | d address. CPU h
00000160 | 61 6C 74 65 64 2E 0A 00 5B 45 52 52 4F 52 5D 20 | alted...[ERROR] 
00000170 | 55 6E 64 65 66 69 6E 65 64 20 73 79 73 74 65 6D | Undefined system
00000180 | 20 63 61 6C 6C 2E 20 43 50 55 20 68 61 6C 74 65 |  call. CPU halte
00000190 | 64 2E 0A 00 53 79 73 74 65 6D 20 72 65 73 65 74 | d...System reset
000001A0 | 2E 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
```

Il va donc falloir écrire un désassembleur et trouver une vulnérabilité dans le noyau pour pouvoir afficher la zone secrète.

En modifiant le firmware pour essayer de faire planter le programme, le serveur nous renvoie l'état du CPU lors du crash, ce qui va nous permettre de déterminer empiriquement le sens des différentes instructions.

```
$ ./10_upload2.py "21AB 0000"
:: Serial port connected.
:: Uploading firmware...
done.
System reset.
-- Exception occurred at 0002: Invalid instruction.
   r0:0000     r1:AB00    r2:0000    r3:0000
   r4:0000     r5:0000    r6:0000    r7:0000
   r8:0000     r9:0000   r10:0000   r11:0000
```

```
  r12:0000    r13:EFFE   r14:0000   r15:0000
   pc:0002 fault_addr:0000 [S:1 Z:0] Mode:user
CLOSING: Invalid instruction.
```

Le script 11_mcu_disas.py implémente un désassembleur (incomplet) qui permet d'étudier le code du firmware et du kernel.

## Analyse du firmware et du kernel

Le pseudo code du firmware est le suivant :

```
syscall_print "Firmware v1.33.7 starting."
init_rc4(0x1000, 0x17c, 0xf) //0x17c => "YeahRiscIsGood!"
decrypt_rc4(0x1000, 0x1B2, 0x29) //=>decrypt "Execution completed in $$$$$ CPU
cycles."
syscall_get_cpu_cycles(0x1100)
str_replace(...) //replace $$$$$ by cpu_cycles value at 0x1100
syscall_print "Execution completed in ..."
syscall_print "Halting"
syscall_halt
```

La chaîne "Execution completed in $$$$$ CPU cycles." est déchiffrée avec l'algorithme RC4 en utilisant la chaîne "YeahRiscIsGood!" comme clé.

```
$ ./12_rc4.py
Execution completed in $$$$$ CPU cycles.
```

Le début du code du kernel gère les syscalls :

```
$ ./11_mcu_disas.py kernel
0000    50 00    and r0, r0, r0      ;
0002    a0 6c    jz r0 0x70          ;
0004    21 00    movhi r1, 0x0       ;
0006    11 03    movlo r1, 0x3       ; r1 = 0x3
0008    72 10    sub r2, r1, r0      ;
000a    a8 12    jz r8 0x1e          ;
000c    22 00    movhi r2, 0x0       ;
000e    12 02    movlo r2, 0x2       ; r2 = 0x2
0010    81 02    mul r1, r0, r2      ;
0012    71 12    sub r1, r1, r2      ;
0014    20 f0    movhi r0, 0xf0      ;
0016    10 00    movlo r0, 0x0       ; r0 = 0xf000
0018    60 01    add r0, r0, r1      ;
001a    c0 94    call 0xb0           ; load_word
001c    d0 00    jmp r0              ;
001e    21 00    movhi r1, 0x0       ;
0020    11 2b    movlo r1, 0x2b      ; r1 = 0x2b
0022    20 fe    movhi r0, 0xfe      ;
0024    10 5a    movlo r0, 0x5a      ; r0 = 0xfe5a
0026    c0 be    call 0xe6           ; kernel_print => [ERROR] Undefined system
call. CPU halted.
...
```

Le noyau supporte 4 syscalls :

- syscall 0 : halt
- syscall 1 : print
- syscall 2 : ?
- syscall 3 : get_cpu_cycles

Les adresses des gestionnaires de syscalls sont stockées à l'adresse 0xF000 dans la zone secrète.

Comme vu précédemment, le syscall "print" ne permet pas de lire dans la zone secrète. On suppose que le syscall "get_cpu_cycles" ne vérifie pas l'adresse de destination qui lui est passée, ce qui permettrait d'écrire une valeur n'importe où en mémoire. On teste cette hypothèse avec le programme suivant :

```
20F0    movhi r0, 0xF0
1000    movlo r0, 0x00
C803    syscall_3
C801    syscall_1
```

```
$ ./10_upload2.py "20F0 1000 C803 C801"
:: Serial port connected.
:: Uploading firmware...
done.
System reset.
-- Exception occurred at 07C0: Invalid instruction.
   r0:07C0     r1:0000    r2:0100    r3:00C0
   r4:0700     r5:0000    r6:0000    r7:0000
   r8:0000     r9:0000   r10:0000   r11:0000
  r12:0000    r13:EFFE   r14:0000   r15:FD1C
   pc:07C0 fault_addr:0000 [S:0 Z:0] Mode:kernel
CLOSING: Invalid instruction.
```

Le pointeur vers le gestionnaire du syscall 1 a bien été réécrit à la valeur 0x7C0. Il suffit donc de charger du code à cette adresse (accessible dans la zone [0000-07FF] - Firmware) pour exécuter du code en mode kernel.

## Exploitation du noyau

Le code permettant d'exploiter le bug est le suivant :

```
start:
    0000    20 f0    movhi r0, 0xf0       ;
    0002    10 00    movlo r0, 0x0        ; r0 = 0xf000
    0004    c8 03    syscall 0x3          ; write 0x7C0 at 0xf000
    0006    c8 01    syscall 0x1          ; call 0xf000[syscallnum-1]=0x7c0
    0008    ff ff    str r15, [ r15 + r15];
    ...
    07c0    2d fc    movhi r13, 0xfc     ;
    07c2    1d 00    movlo r13, 0x0      ; r13 = 0xfc00 (print char hw reg)
    07c4    2a 00    movhi r10, 0x0      ;
    07c6    1a 01    movlo r10, 0x1      ; r10 = 0x1
    07c8    2e f0    movhi r14, 0xf0     ;
    07ca    1e 00    movlo r14, 0x0      ; r14 = 0xf000 (secret memory addr)
```

```
    07cc    21 0b    movhi r1, 0xb       ;
    07ce    11 ff    movlo r1, 0xff      ; r1 = 0xbff   (secret memory size)
print_loop:
    07d0    e9 e8    ldr r9, [ r14 + r8] ;
    07d2    f9 db    str r9, [ r13 + r11]; (r11 = 0)
    07d4    68 8a    add r8, r8, r10     ;
    07d6    71 1a    sub r1, r1, r10     ;
    07d8    b3 -a    jnz r3 0x7d0        ;
```

Les instructions du "shellcode kernel" ont été copié-collées à partir du code du syscall "print" : chaque octet est écrit à l'adresse 0xfc00 qui correspond à un registre "matériel" d'affichage de caractères sur le "port série" du microcontrôleur.

En envoyant ce code au serveur, on reçoit le contenu de la zone mémoire [F000-FBFF] :

```
./10_upload2.py exploit
:: Serial port connected.
:: Uploading firmware...
done.
System reset.
...
Exception occurred at 07D0: Memory access violation.
   r0:07C0     r1:F3FF    r2:0100    r3:00C0
   r4:0700     r5:0000    r6:0000    r7:0000
   r8:1800     r9:0000   r10:0001   r11:0000
  r12:0000    r13:FC00   r14:F000   r15:FD1C
   pc:07D0 fault_addr:0800 [S:1 Z:0] Mode:kernel
CLOSING: Memory access violation.

Response written to result_5d7c5d7cedd1595e231da8ba07827f12.bin

$ strings result_5d7c5d7cedd1595e231da8ba07827f12.bin
...
  WOW
   SUCH EXPLOIT
    VERY CHALLENGING
   SO OPERATIONAL
  MUCH WIN
 <66a65dc050ec0c84cf1dd5b3bbb75c8c@challenge.sstic.org>
```

On obtient l'adresse de validation du challenge, ainsi qu'un Shiba Inu en ascii art :-)

## Conclusion

Merci à Guillaume pour ce challenge de qualité !

# Sources

## 01_extract_badbios.py

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

import struct

data=open("usbtrace","rb").read()
usbtrace=""
for l in data.splitlines():
    if l.find("=") == -1:
        continue
    usbtrace += l[l.find("=")+1:].replace(" ","").decode("hex")

open("usbtrace.bin","wb").write(usbtrace)

wrte = ""
o = 0
while True:
    o = usbtrace.find("WRTE", o)
    if o == -1:
        break
    l = struct.unpack("<L", usbtrace[o+12:o+16])[0]
    wrte += usbtrace[o+24:o+24+l]
    o += 24+l

open("wrte.bin","wb").write(wrte)

o = 0
badbios = ""
while True:
    o = wrte.find("DATA", o)
    if o == -1:
        break
    l = struct.unpack("<L", wrte[o+4:o+8])[0]
    badbios += wrte[o+8:o+8+l]
    o += 8+l

open("badbios.bin","wb").write(badbios)
```

## 02_dump_badbios2.sh

```
#!/usr/bin/gdb -x

file badbios.bin
break * 0x0102C0
commands
dump binary memory badbios2.bin 0x400000 0x400000+0x3000
append binary memory badbios2.bin 0x500000 0x500000+0x11000
quit
```

```
end
run
```

## 03_fix_badbios2_pheader.py

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

import os
import struct

fd = os.open("badbios2.bin", os.O_RDWR)

#http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf
#typedef struct
#{
#unsigned char e_ident[16]; /* ELF identification */
#Elf64_Half e_type; /* Object file type */
#Elf64_Half e_machine; /* Machine type */
#Elf64_Word e_version; /* Object file version */
#Elf64_Addr e_entry; /* Entry point address */
#Elf64_Off e_phoff; /* Program header offset */
#Elf64_Off e_shoff; /* Section header offset */
#Elf64_Word e_flags; /* Processor-specific flags */
#Elf64_Half e_ehsize; /* ELF header size */
#Elf64_Half e_phentsize; /* Size of program header entry */
#Elf64_Half e_phnum; /* Number of program header entries */
#Elf64_Half e_shentsize; /* Size of section header entry */
#Elf64_Half e_shnum; /* Number of section header entries */
#Elf64_Half e_shstrndx; /* Section name string table index */
#} Elf64_Ehdr;

#set e_shoff=0
os.lseek(fd, 0x28, os.SEEK_SET)
os.write(fd, struct.pack("<Q", 0x0))

#set e_shentsize=0, e_shnum=0, e_shstrndx=0
os.lseek(fd, 0x3A, os.SEEK_SET)
os.write(fd, struct.pack("<HHH", 0x0, 0x0, 0x0))

#typedef struct
#{
#Elf64_Word p_type; /* Type of segment */
#Elf64_Word p_flags; /* Segment attributes */
#Elf64_Off p_offset; /* Offset in file */
#Elf64_Addr p_vaddr; /* Virtual address in memory */
#Elf64_Addr p_paddr; /* Reserved */
#Elf64_Xword p_filesz; /* Size of segment in file */
#Elf64_Xword p_memsz; /* Size of segment in memory */
#Elf64_Xword p_align; /* Alignment of segment */
#} Elf64_Phdr;

def Elf64_Phdr(a,b,c,d,e,f,g,h):
    return struct.pack("<LLQQQQQQ", a,b,c,d,e,f,g,h)
```

```
#Start of program headers:          64 (bytes into file)
os.lseek(fd, 64, os.SEEK_SET)

os.write(fd, Elf64_Phdr(1, 0x5, 0x0,  0x400000, 0x0, 0x3000, 0x3000, 0x10000) +
             Elf64_Phdr(1, 0x6, 0x3000,  0x500000, 0x0, 0x11000, 0x11000,
0x10000))
os.close(fd)
```

## 04_decrypt_vm_memory.py

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

from chacha import ChaCha

key="0BADB1050BADB1050BADB1050BADB105".decode("hex")
iv="0000000000000000".decode("hex")

cc8 = ChaCha(key, iv)

datain = open("badbios.bin", "rb").read()[0x10b0:0x10b0+0x10000]
dataout = cc8.decrypt(datain)

open("decrypted_vm_memory.bin","wb").write(dataout)
if dataout.find("Trying") != -1:
    print "ok"
```

## 05_badbios.h

```
#define uint8_t __int8
#define uint32_t __int32
#define uint64_t __int64

typedef struct
{
    uint8_t tau[0x10];
    uint8_t kstuff[0x20];
    uint32_t iv[4];
} chacha_state;

typedef struct
{
    uint64_t    field0;
    uint64_t    pagenum_in_encrypted_ram;
    uint64_t    some_flag;//init to 2
} page_table_entry;//sizeof=0x18

typedef struct
{
    uint32_t flags;
    uint32_t error_code;
    uint64_t field_unk;
```

```
    chacha_state chacha_ctx;//+0x10

    //+0x50
    uint8_t* mmaped_encrypted_ram;//sizeof=0x10000

    //+0x58
    page_table_entry page_table[0x20];//0x20*0x18=0x300

    //+0x358
    uint8_t* decrypted_ram_view;//size=0x1000 => 0x0000007fb7fed000

    //+0x360
    void* instruction_handlers[0x1F];//31 instructions

    //+0x458
} badbios_context;
```

## 06_dump_vm_context.sh

```
#!/usr/bin/gdb -x

file badbios2.bin
break * 0x04000D4
commands
dump binary memory vm_context_struct.bin $x0 $x0+0x458
quit
end
run
```

## 07_rename_vm_handlers_ida.py

```
import struct

h = open("vm_context_struct.bin","rb").read()[0x360:]

for i in xrange(0,len(h)-8,8):
    x = struct.unpack("<Q", h[i:i+8])[0]
    name ="instr_%02x" % (i/8)
    print "%d => %x" % (i/8, x)
    MakeName(x, name)
```

## 08_vm_disas.py

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

import struct

OPCODES={
    0x1:  "mov",
    0x2:  "mov w[reg]",
    0x4:  "mov b[reg]",
    0x7:  "mov_mem",
    0x8:  "jmp",
```

```
    0xa:  "xor",
    0xb:  "or",
    0xc:  "and",
    0xd:  "lsl",
    0xe:  "lsr",

    0x12: "add",
    0x13: "cmp",
    0x16: "inc",
    0x17: "dec",
    0x1C: "exit",
    0x1d: "syscall",
    0x1E: "lfsr"
}

LABELS = {
    0xCA: "convert_hexa",

    0x106: "wut",
    0x108: "wut1",
    0x12C: "wut2",

    0x194: "decrypt_loop",
    0x1F6: "decrypt_loop2",

    0x204: "end_decrypt_loop",
    0x226: "check_padding",

    0x2da: "bad_padding",
    0x2b4: "bad_key_format",
    0x2b2: "exit"
}


def disass():
    data = open("decrypted_vm_memory.bin", "rb").read()
    pc = 0x40

    while pc < 0x32A:
    #while pc < 0x234:
        opcode = ord(data[pc])
        assert (opcode < 0x1F), "invalid opcode %x" % opcode

        pc1 = pc
        if opcode <= 8:
            sz = 4
            op2 = struct.unpack("<L", data[pc:pc+4])[0]
            pc += 4
        else:
            op2 = struct.unpack("<H", data[pc:pc+2])[0]
            pc += 2

        comment = ""
```

```
        r1 = (op2 >> 8) & 0xF
        r2 = (op2 >> 12) & 0xF

        if opcode == 0x0:
            comment = "mov r%d, 0x%x" % (r1, r2 << 16)
        elif opcode == 0x1:
            val = (op2 >> 12)
            comment = "or r%d, 0x%04x" % (r1, val)
            if val >= 0x326:
                s = data[val:data.find("\x00",val)].split("\n")[0]
                comment += "   %s" % (s)
        elif opcode == 0x2:
            off = (op2 >> 16)
            if r2 == 0:#hax, read mapped register
                comment = "mov r%d, r%d" % (r1, (off/4)+1)
            else:
                comment = "mov r%d, w[r%d + 0x%x]" % (r1, r2, off)
        elif opcode == 0x4:
            comment = "mov r%d, byte[r%d]" % (r1, r2)

        elif opcode == 0x7:
            off = (op2 >> 16)
            comment = "mov b[r%d + 0x%x], r%d" % (r2, off, r1)

        elif opcode == 0x8:
            dst = op2 >> 16
            c1 = (op2 >> 9) & 7
            c2 = (op2 >> 0xD) & 3
            cond = "%d %d" % (c1, c2)

            comment = "%s %s" % (cond, LABELS.get(dst, "%04X" % dst))

        elif opcode == 0x1E:
            comment = "lfsr r%d, r%d" % (r1, r2)
        elif opcode == 0xA:
            comment = "xor r%d, r%d" % (r1, r2)
        elif opcode == 0xB:
            comment = "or r%d, r%d" % (r1, r2)
        elif opcode == 0xC:
            comment = "and r%d, r%d" % (r1, r2)
        elif opcode == 0xD:
            comment = "lsl r%d, r%d" % (r1, r2)
        elif opcode == 0xE:
            comment = "lsr r%d, r%d" % (r1, r2)
        elif opcode == 0x12:
            comment = "add r%d, r%d" % (r1,r2)
        elif opcode == 0x13:
            comment = "cmp r%d, r%d" % (r1,r2)
        elif opcode == 0x16:
            comment = "inc r%d" % r1
        elif opcode == 0x17:
            comment = "dec r%d" % r1
```

```
        if LABELS.has_key(pc1):
            print "%s:" % LABELS[pc1]

        print "\t0x%04x : %s   %08x    %s" % (pc1, OPCODES.get(opcode, "op_%02X
" % opcode).ljust(10), op2, comment)


disass()
```

## 09_lfsr.py

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

import struct

def bit(r, b): return (r >> b) & 1

def push_bit_right(r, b):
    return (r >> 1) | (b << 63)

def push_bit_left(r, b):
    return ((r << 1) | b) & 0xffffffffffffffff

#lfsr taps
#0xb000000000000001 => #0b1011000000000000000000000000000000
def step(state):
    r9 = bit(state, 63) ^ bit(state, 61) ^ bit(state, 60) ^ bit(state, 0)
    return push_bit_right(state, r9)

#inverse lfsr
def rstep(state):
    r9 = bit(state, 62) ^ bit(state, 60) ^ bit(state, 59) ^ bit(state, 63)
    return push_bit_left(state, r9)

def decrypt_payload(buf, key):
    res = ""

    if type(key) == str:
        state = struct.unpack("<Q", key.decode("hex"))[0]
    else:
        state = key

    for i in xrange(len(payload)):
        output_byte = 0x0
        current_bit_pos = 8

        while current_bit_pos > 0:
            state = step(state)
            current_bit_pos -= 1
            output_byte |= ((state & 1) << current_bit_pos)

        res += chr(ord(payload[i]) ^ output_byte)
```

```
    return res

def binary_reverse64(num):
    z = bin(num)[2:]
    z = z.rjust(64,"0")
    return int(''.join(reversed(z)), 2)

if __name__ == "__main__":
    payload = open("decrypted_vm_memory.bin", "rb").read()[0x8000:0x8000+0x2000]

    target_output = payload[-8:]
    state = binary_reverse64(struct.unpack(">Q", target_output)[0])
    print "Target end state: 0x%x" % state

    for i in xrange((0x2000-8) * 8 + 1):
        state = rstep(state)
    print "Initial LFSR state = 0x%x " % state
    print "Key => %s" % struct.pack("<Q", state).encode("hex").upper()

    res = decrypt_payload(payload, state)
    assert res[-8:] == "\x00"*8

    print "Writing decrypted payload to payload.zip"
    open("payload.zip","wb").write(res)
```

## 10_upload2.py

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

import hashlib, socket, select, sys

hex = lambda data: " ".join("%02X" % ord(i) for i in data)
ascii = lambda data: "".join(c if 31 < ord(c) < 127 else "." for c in data)

def hexdump(d):
    for i in xrange(0,len(d),16):
        data = d[i:i+16]
        print "%08X | %s | %s" % (i, hex(data).ljust(47), ascii(data))

def bintohex(data):
    res = ""
    for i in xrange(0,len(data),16):
        d = data[i:i+16]
        line = "%02X" % len(d)
        line += "%04X" % i
        line += "00"
        line += d.encode("hex")

        csum = (sum(map(ord, line.decode("hex"))) ^ 0xFF) + 1
        line += "%02X" % (csum & 0xFF)
        res += ":" + line.upper() + "\x0A"
    res += ":00000001FF"
```

```
    return res

def upload(data):
    input = bintohex(data)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('178.33.105.197', 10101))

    print ":: Serial port connected."
    print ":: Uploading firmware... "

    s.send(input)

    print "done."

    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

    s.close()
    outputfilename = "result_%s.bin" % hashlib.md5(input).hexdigest()
    open(outputfilename,"wb").write(resp)
    #hexdump(resp)
    print resp

    print "Response written to %s" % outputfilename

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print "Usage: %s fw.bin|hex string"
        exit(0)
    arg = sys.argv[1]
    if arg.endswith(".bin"):
        data = open(arg, 'rb').read()
    elif arg == "exploit":
        data = "20F01000C803C801FFFF".decode("hex")
        data = data.ljust(0x7C0)
        data += """2d fc 1D 00
        2A 00 1A 01
        2E F0 1E 00
        21 0B 11 FF

        E9 E8
        F9 DB
```

```
        68 8A
        71 1A
        B3 F6""".replace(" ","").replace("\n", "").decode("hex")
        open("mcu_exploit.bin","wb").write(data)
        #exit(0)
    else:
        data = sys.argv[1].replace(" ","").decode("hex")
    upload(data)
```

## 11_mcu_disas.py

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

import struct
import sys

LABELS_fw = {
    0x54: "init_rc4",
    0x9C: "rc4_decrypt",
    0xd8: "syscall1_halt",
    0xdc: "print",
    0xe0: "syscall_numcycles",
    0xe4: "load_word",
    0xf8: "replace_string"
}
LABELS_kernel = {
    0xb0: "load_word",
    0xC4: "store_word",
    0xe6: "kernel_print",
    0x126: "data"
}

def disas(filename, base=0, labels={}):
    data = open(filename, "rb").read()
    regs = {}

    for i in xrange(0, len(data), 2):
        byte1 = ord(data[i])
        byte2 = ord(data[i+1])

        op = byte1 >> 4
        reg = byte1 & 0xf
        op2 = byte2 >> 4
        op3 = byte2 & 0xf
        comment = ""

        if op == 2:
            desc = "movhi r%d, 0x%x" % (reg, byte2)
            regs[reg] = (regs.get(reg, 0) & 0xFF) | byte2 << 8
        elif op == 1:
            regs[reg] = (regs.get(reg, 0) & 0xFF00) | byte2
            desc = "movlo r%d, 0x%x" % (reg, byte2)
            comment = "r%d = 0x%x" % (reg, regs[reg])
```

```
        elif op == 3:
            desc = "eor r%d, r%d, r%d" % (reg, op2, op3)
        elif op == 4:
            desc = "or r%d, r%d, r%d" % (reg, op2, op3)
        elif op == 5:
            desc = "and r%d, r%d, r%d" % (reg, op2, op3)
        elif op == 6:
            desc = "add r%d, r%d, r%d" % (reg, op2, op3)
        elif op == 7:
            desc = "sub r%d, r%d, r%d" % (reg, op2, op3)
        elif op == 8:
            desc = "mul r%d, r%d, r%d" % (reg, op2, op3)
        elif op == 0x9:
            desc = "div r%d, r%d, r%d" % (reg, op2, op3)
        elif op == 0xe:
            desc = "ldr r%d, [ r%d + r%d]" % (reg, op2, op3)
        elif op == 0xf:
            desc = "str r%d, [ r%d + r%d]" % (reg, op2, op3)
        elif byte1 == 0xc0:
            dest = byte2 + i + 2
            comment = labels.get(dest, "")
            if comment.find("print") != -1:
                comment += " => " + data[regs[0]-base:regs[0]-base+regs[1]]
            desc = "call 0x%x" % dest
        elif op == 0xa:
            dest = byte2 + i + 2
            desc = "jz r%d 0x%x" % (reg, dest)
        elif byte1 == 0xC1:
            dest = byte2 + i + 2
            comment = labels.get(dest, "")
            desc = "jxx 0x%x" % dest
        elif op == 0xB:
            byte2 = struct.unpack("<b", struct.pack("<B", byte2))[0]
            dest = (byte2 + i + 2)
            comment = labels.get(dest, "")
            desc = "jnz r%d 0x%x" % (reg, dest)
        elif byte1 == 0xC8:
            desc = "syscal 0x%x" % (byte2)
        elif op == 0xD:
            desc = "jmp r%d" % (byte2)

        else:
            desc = "%02x %02x" % (byte1, byte2)

        if labels.has_key(i):
            print "%s:" % labels[i]
        print "\t%04x    %02x %02x    %s" % (i, byte1, byte2, desc.ljust(20) +
"; " + comment)


if "fw" in sys.argv:
    disas("fw.bin", labels=LABELS_fw)
elif "kernel" in sys.argv:
```

```
    disas("kernel_FD00.bin", 0xFD00, LABELS_kernel)
else:
    disas(sys.argv[1], {})
```

## 12_rc4.py

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

from Crypto.Cipher import ARC4
hex = lambda data: " ".join("%02X" % ord(i) for i in data)
ascii = lambda data: "".join(c if 31 < ord(c) < 127 else "." for c in data)

def hexdump(d):
    for i in xrange(0,len(d),16):
        data = d[i:i+16]
        print "%08X | %s | %s" % (i, hex(data).ljust(47), ascii(data))

data = open("fw.bin", "rb").read()
key = data[0x17c:0x17c+0xF]
#print key
blob = data[0x1b2:0x1b2+0x29]

dec = ARC4.new(key).encrypt(blob)

print dec
```

## chacha.py

```
"""
    chacha.py

    An implementation of ChaCha in about 130 operative lines
    of 100% pure Python code.

    Copyright (c) 2009-2011 by Larry Bugbee, Kent, WA
    ALL RIGHTS RESERVED.

    chacha.py IS EXPERIMENTAL SOFTWARE FOR EDUCATIONAL
    PURPOSES ONLY.  IT IS MADE AVAILABLE "AS-IS" WITHOUT
    WARRANTY OR GUARANTEE OF ANY KIND.  USE SIGNIFIES
    ACCEPTANCE OF ALL RISK.

    To make your learning and experimentation less cumbersome,
    chacha.py is free for any use.

    This implementation is intended for Python 2.x.

    Larry Bugbee
    May 2009     (Salsa20)
    August 2009  (ChaCha)
    rev June 2010
    rev March 2011  - tweaked _quarterround() to get 20-30% speed gain
"""
```

```
import struct
try:
    import psyco            # a specializing [runtime] compiler
    have_psyco = True       # for 32-bit architectures
    print 'psyco enabled'
except:
    have_psyco = False

#---------------------------------------------------------------------

class ChaCha(object):
    """
        ChaCha is an improved variant of Salsa20.

        Salsa20 was submitted to eSTREAM, an EU stream cipher
        competition.  Salsa20 was originally defined to be 20
        rounds.  Reduced round versions, Salsa20/8 (8 rounds) and
        Salsa20/12 (12 rounds), were later submitted.  Salsa20/12
        was chosen as one of the winners and 12 rounds was deemed
        the "right blend" of security and efficiency.  Salsa20
        is about 3x-4x faster than AES-128.

        Both ChaCha and Salsa20 accept a 128-bit or a 256-bit key
        and a 64-bit IV to set up an initial 64-byte state.  For
        each 64-bytes of data, the state gets scrambled and XORed
        with the previous state.  This new state is then XORed
        with the input data to produce the output.  Both being
        stream ciphers, their encryption and decryption functions
        are identical.

        While Salsa20's diffusion properties are very good, some
        claimed the IV/keystream correlation was too strong for
        comfort.  To satisfy, another variant called XSalsa20
        implements a 128-bit IV.  For the record, EU eSTREAM team
        did select Salsa20/12 as a solid cipher providing 128-bit
        security.

        ChaCha is a minor tweak of Salsa20 that significantly
        improves its diffusion per round.  ChaCha is more secure
        than Salsa20 and 8 rounds of ChaCha, aka ChaCha8, provides
        128-bit security.  (FWIW, I have not seen any calls for a
        128-bit IV version of ChaCha or XChaCha.)

        Another benefit is that ChaCha8 is about 5-8% faster than
        Salsa20/8 on most 32- and 64-bit PPC and Intel processors.
        SIMD machines should see even more improvement.

        Sample usage:
          from chacha import ChaCha

          cc8 = ChaCha(key, iv)
          ciphertext = cc8.encrypt(plaintext)
```

```
      cc8 = ChaCha(key, iv)
      plaintext = cc8.decrypt(ciphertext)

    Remember, the purpose of this program is educational; it
    is NOT a secure implementation nor is a pure Python version
    going to be fast.  Encrypting large data will be less than
    satisfying.  Also, no effort is made to protect the key or
    wipe critical memory after use.

    Note that psyco, a specializing compiler somewhat akin to
    a JIT, can provide a 2x+ performance improvement over
    vanilla Python 32-bit architectures.  A 64-bit version of
    psyco does not exist.  See http://psyco.sourceforge.net

    For more information about Daniel Bernstein's ChaCha
    algorithm, please see http://cr.yp.to/chacha.html

    All we need now is a keystream AND authentication in the
    same pass.

    Larry Bugbee
    May 2009     (Salsa20)
    August 2009  (ChaCha)
    rev June 2010
"""

TAU    = ( 0x61707865, 0x3120646e, 0x79622d36, 0x6b206574 )
SIGMA  = ( 0x61707865, 0x3320646e, 0x79622d32, 0x6b206574 )
ROUNDS = 8                          # ...10, 12, 20?

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def __init__(self, key, iv, rounds=ROUNDS):
    """ Both key and iv are byte strings.  The key must be
        exactly 16 or 32 bytes, 128 or 256 bits respectively.
        The iv must be exactly 8 bytes (64 bits) and MUST
        never be reused with the same key.

        The default number of rounds is 8.

        If you have several encryptions/decryptions that use
        the same key, you may reuse the same instance and
        simply call iv_setup() to set the new iv.  The previous
        key and the new iv will establish a new state.
    """
    self._key_setup(key)
    self.iv_setup(iv)
    self.rounds = rounds

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
    def _key_setup(self, key):
        """ key is converted to a list of 4-byte unsigned integers
            (32 bits).

            Calling this routine with a key value effectively resets
            the context/instance.  Be sure to set the iv as well.
        """
        if len(key) not in [16, 32]:
            raise Exception('key must be either 16 or 32 bytes')
        key_state = [0]*16
        if len(key) == 16:
            k = list(struct.unpack('<4I', key))
            key_state[0]  = self.TAU[0]
            key_state[1]  = self.TAU[1]
            key_state[2]  = self.TAU[2]
            key_state[3]  = self.TAU[3]
            key_state[4]  = k[0]
            key_state[5]  = k[1]
            key_state[6]  = k[2]
            key_state[7]  = k[3]
            key_state[8]  = k[0]
            key_state[9]  = k[1]
            key_state[10] = k[2]
            key_state[11] = k[3]
            # 12 and 13 are reserved for the counter
            # 14 and 15 are reserved for the IV

        elif len(key) == 32:
            k = list(struct.unpack('<8I', key))
            key_state[0]  = self.SIGMA[0]
            key_state[1]  = self.SIGMA[1]
            key_state[2]  = self.SIGMA[2]
            key_state[3]  = self.SIGMA[3]
            key_state[4]  = k[0]
            key_state[5]  = k[1]
            key_state[6]  = k[2]
            key_state[7]  = k[3]
            key_state[8]  = k[4]
            key_state[9]  = k[5]
            key_state[10] = k[6]
            key_state[11] = k[7]
            # 12 and 13 are reserved for the counter
            # 14 and 15 are reserved for the IV
        self.key_state = key_state

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def iv_setup(self, iv):
        """ self.state and other working structures are lists of
            4-byte unsigned integers (32 bits).

            The iv is not a secret but it should never be reused
            with the same key value.  Use date, time or some other
```

```
        counter to be sure the iv is different each time, and
        be sure to communicate the IV to the receiving party.
        Prepending 8 bytes of iv to the ciphertext is the usual
        way to do this.

        Just as setting a new key value effectively resets the
        context, setting the iv also resets the context with
        the last key value entered.
    """
    if len(iv) != 8:
        raise Exception('iv must be 8 bytes')
    v = list(struct.unpack('<2I', iv))
    iv_state = self.key_state[:]
    iv_state[12] = 0
    iv_state[13] = 0
    iv_state[14] = v[0]
    iv_state[15] = v[1]
    self.state = iv_state
    self.lastblock_sz = 64      # init flag - unsafe to continue
                                # processing if not 64

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def encrypt(self, datain):
    """ Encrypt a chunk of data.  datain and the returned value
        are byte strings.

        If large data is submitted to this routine in chunks,
        the chunk size MUST be an exact multiple of 64 bytes.
        Only the final chunk may be less than an even multiple.
        (This function does not "save" any uneven, left-over
        data for concatenation to the front of the next chunk.)

        The amount of available memory imposes a poorly defined
        limit on the amount of data this routine can process.
        Typically 10's and 100's of KB are available but then,
        perhaps not.  This routine is intended for educational
        purposes so application developers should take heed.
    """
    if self.lastblock_sz != 64:
        raise Exception('last chunk size not a multiple of 64 bytes')
    dataout = []
    while datain:
        # generate 64 bytes of cipher stream
        stream = self._chacha_scramble();
        # XOR the stream onto the next 64 bytes of data
        dataout.append(self._xor(stream, datain))
        if len(datain) <= 64:
            self.lastblock_sz = len(datain)
            return ''.join(dataout)
        # increment the iv.  In this case we increment words
        # 12 and 13 in little endian order.  This will work
        # nicely for data up to 2^70 bytes (1,099,511,627,776GB)
```

```
            # in length.  After that it is the user's responsibility
            # to generate a new nonce/iv.
            self.state[12] = (self.state[12] + 1) & 0xffffffff
            if self.state[12] == 0:           # if overflow in state[12]
                self.state[13] += 1           # carry to state[13]
                # not to exceed 2^70 x 2^64 = 2^134 data size ??? <<<<
            # get ready for the next iteration
            datain = datain[64:]
        # should never get here
        raise Exception('Huh?')

    decrypt = encrypt

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def _chacha_scramble(self):     # 64 bytes in
        """ self.state and other working strucures are lists of
            4-byte unsigned integers (32 bits).

            output must be converted to bytestring before return.
        """
        x = self.state[:]           # makes a copy
        for i in xrange(0, self.rounds, 2):
            # two rounds per iteration
            self._quarterround(x, 0, 4, 8,12)
            self._quarterround(x, 1, 5, 9,13)
            self._quarterround(x, 2, 6,10,14)
            self._quarterround(x, 3, 7,11,15)

            self._quarterround(x, 0, 5,10,15)
            self._quarterround(x, 1, 6,11,12)
            self._quarterround(x, 2, 7, 8,13)
            self._quarterround(x, 3, 4, 9,14)

        for i in xrange(16):
            x[i] = (x[i] + self.state[i]) & 0xffffffff
        output = struct.pack('<16I',
                             x[ 0], x[ 1], x[ 2], x[ 3],
                             x[ 4], x[ 5], x[ 6], x[ 7],
                             x[ 8], x[ 9], x[10], x[11],
                             x[12], x[13], x[14], x[15])
        return output               # 64 bytes out

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    '''
    # as per definition - deprecated
    def _quarterround(self, x, a, b, c, d):
        x[a] = (x[a] + x[b]) & 0xFFFFFFFF
        x[d] = self._rotate((x[d]^x[a]), 16)
        x[c] = (x[c] + x[d]) & 0xFFFFFFFF
        x[b] = self._rotate((x[b]^x[c]), 12)
```

```
        x[a] = (x[a] + x[b]) & 0xFFFFFFFF
        x[d] = self._rotate((x[d]^x[a]), 8)
        x[c] = (x[c] + x[d]) & 0xFFFFFFFF
        x[b] = self._rotate((x[b]^x[c]), 7)

    def _rotate(self, v, n):        # aka ROTL32
        return ((v << n) & 0xFFFFFFFF) | (v >> (32 - n))
    '''

    # surprisingly, the following tweaks/accelerations provide
    # about a 20-40% gain
    def _quarterround(self, x, a, b, c, d):
        xa = x[a]
        xb = x[b]
        xc = x[c]
        xd = x[d]

        xa  = (xa + xb)  & 0xFFFFFFFF
        tmp =  xd ^ xa
        xd  = ((tmp << 16) & 0xFFFFFFFF) | (tmp >> 16)  # 16=32-16
        xc  = (xc + xd)  & 0xFFFFFFFF
        tmp =  xb ^ xc
        xb  = ((tmp << 12) & 0xFFFFFFFF) | (tmp >> 20)  # 20=32-12

        xa  = (xa + xb)  & 0xFFFFFFFF
        tmp =  xd ^ xa
        xd  = ((tmp <<  8) & 0xFFFFFFFF) | (tmp >> 24)  # 24=32-8
        xc  = (xc + xd)  & 0xFFFFFFFF
        tmp =  xb ^ xc
        xb  = ((tmp <<  7) & 0xFFFFFFFF) | (tmp >> 25)  # 25=32-7

        x[a] = xa
        x[b] = xb
        x[c] = xc
        x[d] = xd


    def _xor(self, stream, datain):
        dataout = []
        for i in xrange(min(len(stream), len(datain))):
            dataout.append(chr(ord(stream[i])^ord(datain[i])))
        return ''.join(dataout)

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    if have_psyco:
        # if you psyco encrypt() and _chacha_scramble() you
        # should get a 2.4x speedup over vanilla Python 2.5.
        # The other functions seem to offer only negligible
        # improvement.  YMMV.

        _key_setup = psyco.proxy(_key_setup)    # small impact
        iv_setup   = psyco.proxy(iv_setup)      # small impact
```

```
        encrypt    = psyco.proxy(encrypt)                    # 18-32%
        _chacha_scramble = psyco.proxy(_chacha_scramble)    # big help, 2x
        _quarterround    = psyco.proxy(_quarterround)       # ???
#       _rotate = psyco.proxy(_rotate)          # minor impact
        _xor    = psyco.proxy(_xor)             # very small impact
        pass

#--------------------------------------------------------------------------
#--------------------------------------------------------------------------
#--------------------------------------------------------------------------
```

## upload.py

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

import socket, select

#
# Microcontroller architecture appears to be undocumented.
# No disassembler is available.
#
# The datasheet only gives us the following information:
#
#   == MEMORY MAP ==
#
#   [0000-07FF] - Firmware                  \
#   [0800-0FFF] - Unmapped                  | User
#   [1000-F7FF] - RAM                       /
#   [F000-FBFF] - Secret memory area        \
#   [FC00-FCFF] - HW Registers              | Privileged
#   [FD00-FFFF] - ROM (kernel)              /
#

FIRMWARE = "fw.hex"

print("-------------------------------------------")
print("----- Microcontroller firmware uploader -----")
print("-------------------------------------------")
print()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('178.33.105.197', 10101))

print(":: Serial port connected.")
print(":: Uploading firmware... ", end='')

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

print("done.")
print()

resp = b''
while True:
```

```
    ready, _, _ = select.select([s], [], [], 10)
    if ready:
        try:
            data = s.recv(32)
        except:
            break
        if not data:
            break
        resp += data
    else:
        break

print(resp.decode("utf-8"))
s.close()
```