Shellcode Creation and Analysis

About me

- Cyber Security Consultant at Deloitte
- 3.5 years of cyber experience
- OSCP certified

Linkedin – https://www.linkedin.com/in/shikhar-joshi-2507b871/

Mail – shikhar.joshi19@gmail.com

Agenda

- Understand Shellcode
- See some shellcode in action through buffer overflow
- Develop concepts on assembly
- Get familiarize with GDB
- A bit of reversing
- Overview of Shellcoding
- Create a basic shellcode
- Deepdive into execve shellcode
- Encoding to decoding of shellcodes
- Polymorphic shellcodes
- Analyzing bigger shellcodes like bind shell.
- Staged and Non Staged Shellcodes

Stack based Buffer Overflow

- Check the input field and see where the program crashes.
- Take control of EIP register
- Identify the address where the execution has to be jumped.
- Modify the EIP register to the address identified.
- Fire up the exploit with the payload, and get a shell or perform code execution.

Before we start

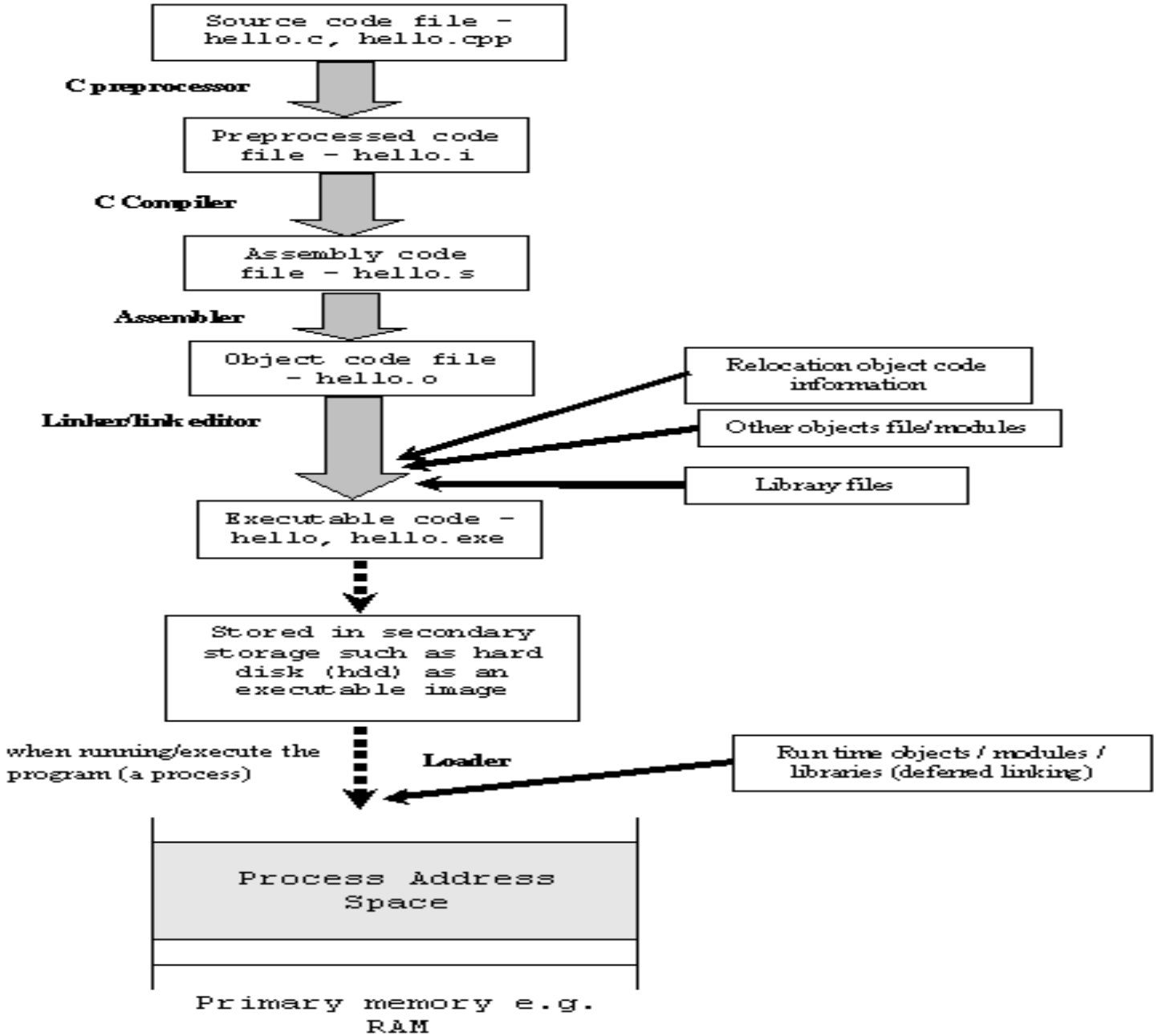
- This is not a complete assembly language session. Hence, we will be covering only those segments which will be needed for our shellcoding exercise.

- Please download all the code files and scripts from the below repository:

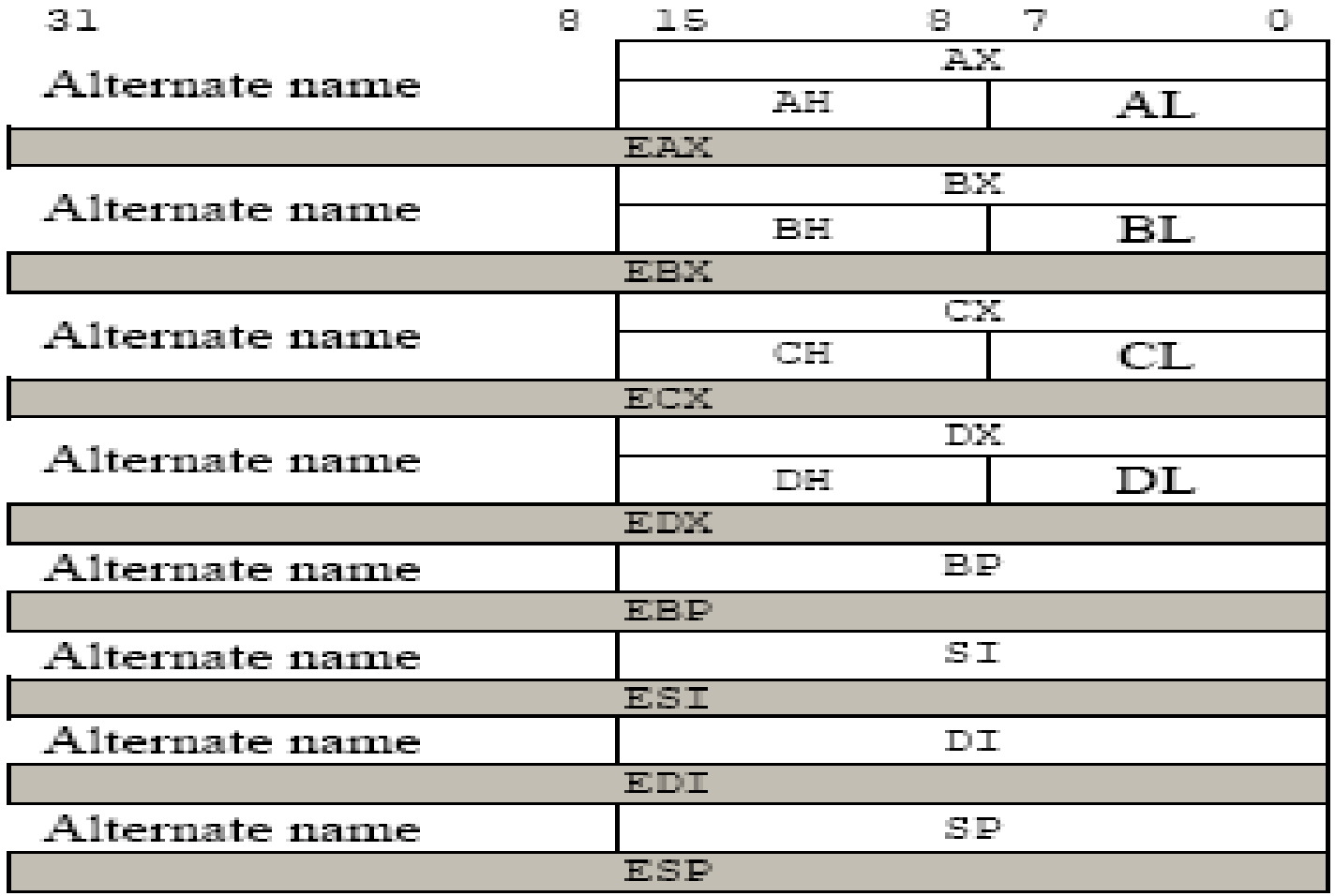
https://github.com/rrd7/Null_puliya_shellcoding

Assembly Language

- It is a low level programming language which communicates with the processor directly.
- We will be dealing with Linux 32 bit intel assembly.
- Different for Intel and ARM.
- Even Intel architecture is divided into 2:
  - IA 32
  - IA 64
- Here we will be specifically dealing with IA 32 LINUX assembly.


Source code file - hello.c, hello.cpp
C preprocessor
Preprocessed code file - hello.i
C Compiler
Assembly code file - hello.s
Assembler
Object code file - hello.o
Linker/link editor
Relocation object code information
Other objects file/modules
Library files
Executable code - hello, hello.exe
Stored in secondary storage such as hard disk (hdd) as an executable image
when running/execute the program (a process)
Loader
Run time objects / modules / libraries (deferred linking)
Process Address Space
Primary memory e.g. RAM

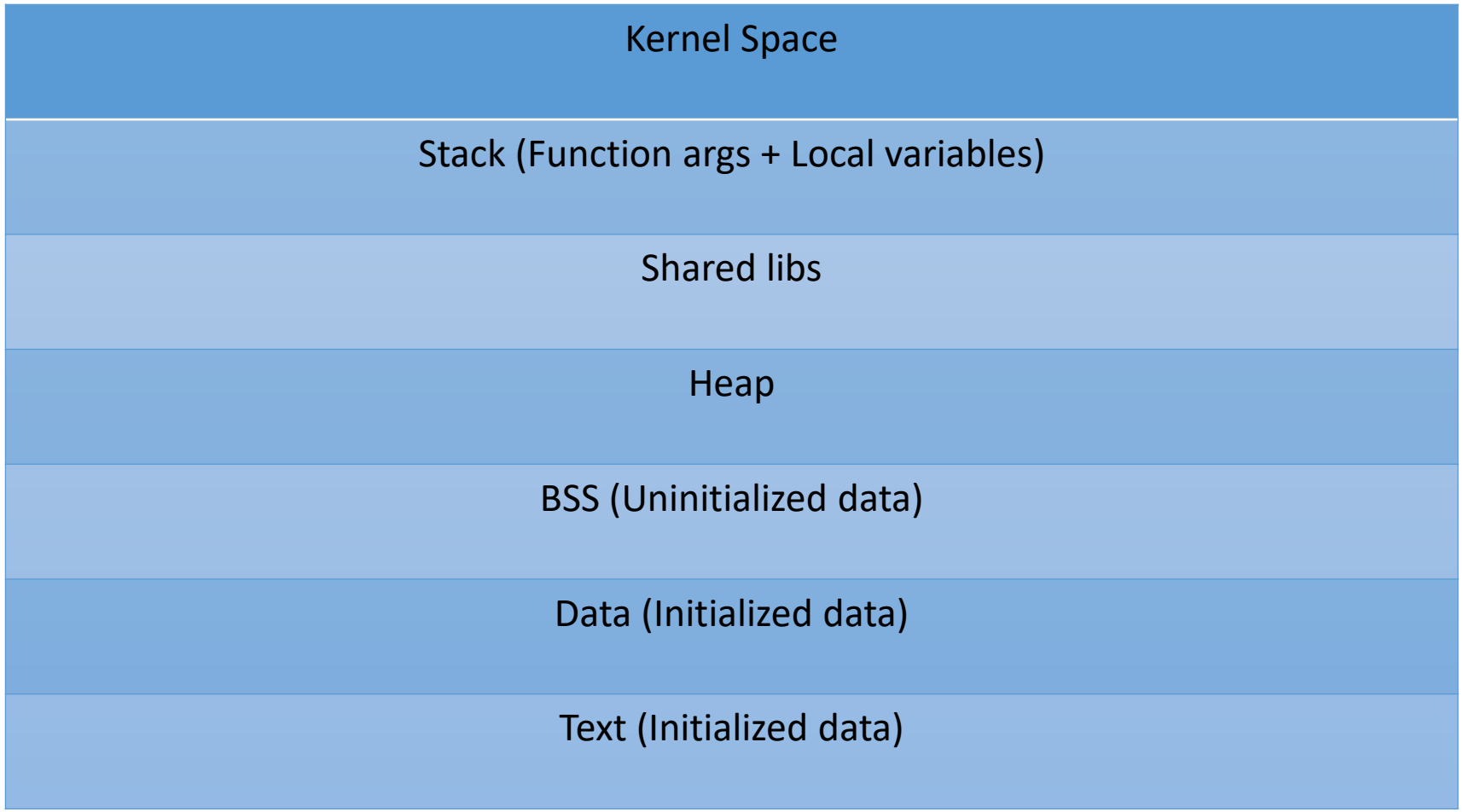
Registers


31
8
15
8
7
0
Alternate name
AX
AH
AL
EAX
Alternate name
BX
BH
BL
EBX
Alternate name
CX
CH
CL
ECX
Alternate name
DX
DH
DL
EDX
Alternate name
BP
EBP
Alternate name
SI
ESI
Alternate name
DI
EDI
Alternate name
SP
ESP

- EAX – Will contain system calls
- EBX – First argument for system calls
- ECX – Second argument for system calls
- EDX – Third argument for system calls
- ESI and EDI – We can use arbitrarily or for the remaining arguments
- EIP – Holy grail of shellcoding. Will contain the address of the next instruction to be executed. (32 bit)
- ESP – Will point to top of the stack.

Note: - Apart from the usage mentioned in this slide, the registers have other usage as well.

Memory Model



System Calls

- Leverage OS for tasks
- Provides a simple interface for user space programs to the kernel
- Int 0x80 to invoke a system call
- /usr/include/1386-linux-gnu/asm/unistd_32.h”
- For write system call
  - *mov eax, (system call number)*
  - *mov ebx, (file descriptor for stdout)*
  - *mov ecx, (pointer to the what has to be written)*
  - *mov edx, (length)*
  - *int 0x80*

Installation instructions

- apt-get install nasm
- apt-get install build-essential make libglib2.0-dev

Assembly Syntax

```
Global _start
Section .text
_start:
        ----
        ----
Section .data
        ----
Section .bss
        -----
```

MOV, LEA and XCHG instructions

- mov eax, 0x4
- mov ebx, eax
- mov eax, [example]

- lea ebx, [eax]
- lea eax, [example]

- xchg eax, ebx

Let’s run our first program

- nasm -f elf32 -o code.o code.nasm
- ld –o code code.o
- ./code

Data Types

- Byte – 8 bits
- Word – 16 bits
- Double Word – 32 bits
- Quad Word – 64 bits
- Double Quad Word – 128 bits

GDB

- Run time analysis
- Debugging
- Changing program flow

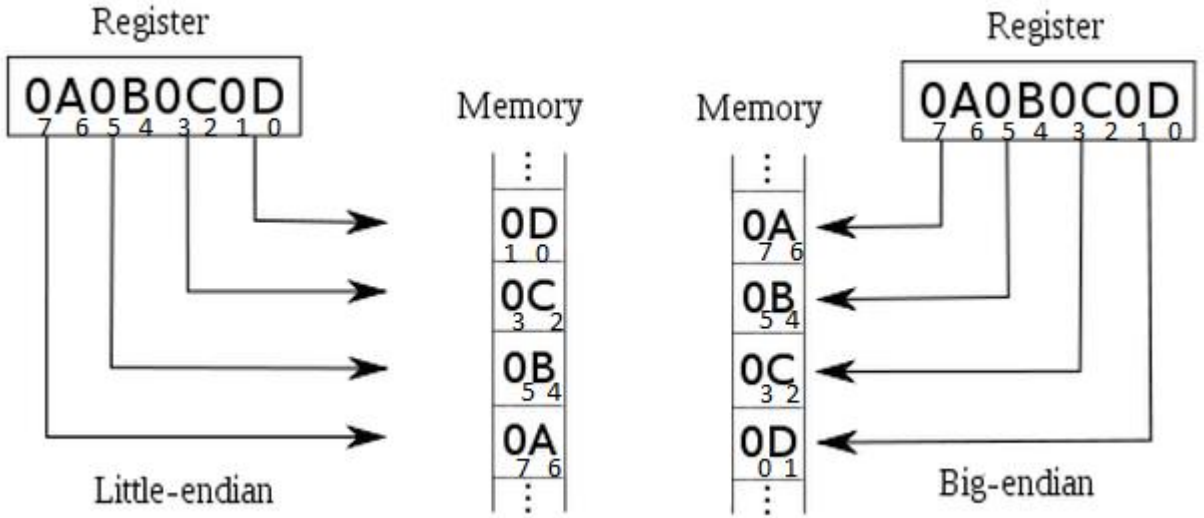
Commands:

- gdb –q code
- Set disassembly-flavor intel
- disassemble

- break _start
- Info registers
- Info functions
- Info breakpoints
- print/x $register
- help x
- x/4xb memory_address or $register
- define hook-stop
  - Print/x $eax
  - x/4xb $esp
  - Disassemble $eip, +10
  - end

Little Endian

- Least significant bit goes to the lower memory address and most significant bit goes to the higher memory address.



Stack

- Stores local variables
- Return addresses
- LIFO data structure
- Grows from higher to lower memory
- PUSH – pushes a value onto Stack
- POP – Removes the topmost value from the stack
- ESP – Point to the top of the stack

Jump Instructions

- Unconditional jump – JMP
- Conditional jumps
  - Uses flags to determine jumps
  - For example a decrement situation led to 0
  - jz, jnz

Shellcoding

- Simply a machine code
- Can be executed by CPU directly

Things to make sure:

- Size
- Bad characters

http://www.shell-storm.org/shellcode

https://www.exploit-db.com/shellcodes

Lets run our first shellcode

- ./Shellcodedump.sh shellcode
- Copy the shellcode in shellcode.c as shellcode[]
- Compile the c program using the below command

```
gcc -fno-stack-protector -z execstack shellcode.c -o shellcode
```

JMP-CALL-POP

```
JMP shellcode
execute:
        pop ebx
        ----
        ----
shellcode:
        call execute:
        example db “Brandon Stark”
```

Execve Shellcode

- Execute a new program from within the shellcode
- “/bin/sh” to get a shell
- System call number - 11
- Does not return if successful
- There is not need or exit() to be called
- Let’s check the blog for details

http://hackoftheday.securitytube.net/2013/04/demystifying-execve-shellcode-stack.html

Encoding and Decoding Shellcodes

- Why encoding?
- Why not shikata-ganai and other Metasploit encoders?
- When will we encode?
- When will we decode?
- And the most important question – how?

XOR encoding and decoding

- A **XOR** B = c
- C **XOR** B = A

A	B	A XOR B
0	0	0
1	1	0
1	0	1
0	1	1

For example:

Shellcode **XOR** 0xAA = encoded_shellcode

Encoded_shellcode **XOR** 0xAA = shellcode

Polymorphism

- As we have seen earlier how AVs could detect our shellcode based on fingerprinting
- Even our decoded stub can be fingerprinted.
- So, how can we use polymorphism to avoid this?
- Lets check out.

Let’s analyze a bind shell?

What does it look like:

- Socketcall
- Bind
- Listen
- Accept
- Dup2
- Execve

Staged vs Non Staged

- Staged
  - Not sent completely in a single go
  - Basically composed of two different payloads
  - One a relatively smaller one and secondary payload is the actual one.
  - Can we used when we have limited size

- Non staged
  - Basic payloads what we have seen till now.
  - Need to have sufficient space.

References

http://www.shell-storm.org/shellcode

https://www.exploit-db.com/shellcodes

http://hackoftheday.securitytube.net/2013/04/demystifying-execve-shellcode-stack.html

https://www.rcesecurity.com/2014/07/slae-shell-bind-tcp-shellcode-linux-x86/

https://www.youtube.com/watch?v=K0g-twyhmQ4&list=PL6brsSrstzga43kcZRn6nbSi_GeXoZQhR