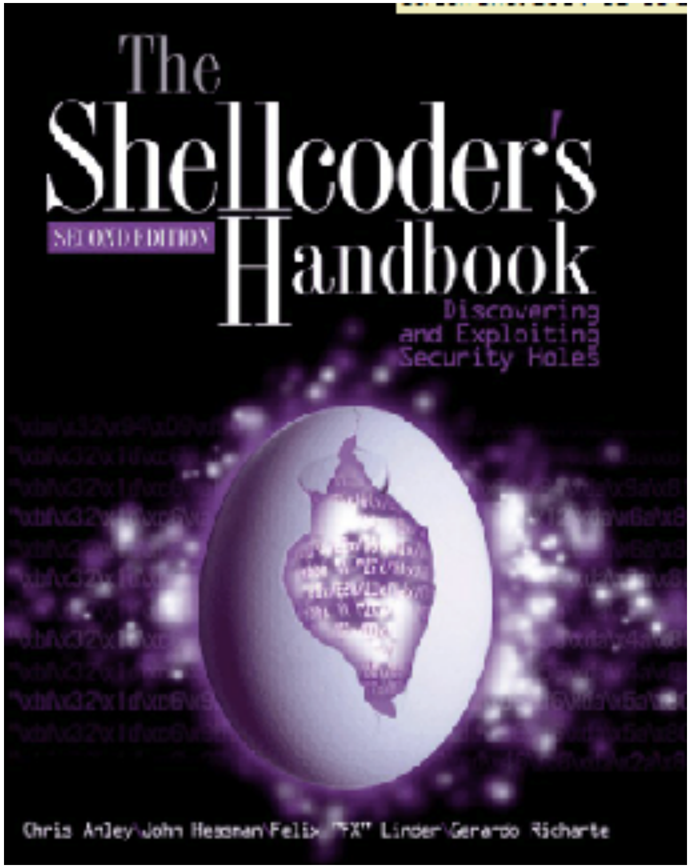
CNIT 127: Exploit Development

Ch 1: Before you begin


The Shellcoder's Handbook
SECOND EDITION
Discovering and Exploiting Security Holes
Chris Anley\John Heasman\Felix "FX" Linder\Gerardo Richarte

Updated 1-14-16

Basic Concepts

Vulnerability

- A flaw in a system that allows an attacker to do something the designer did not intend, such as
  - Denial of service (loss of availability)
  - Elevating privileges (e.g. user to Administrator)
  - Remote Code Execution (typically a *remote shell*)

Exploit

- Exploit (v.)
  - To take advantage of a vulnerability and cause a result the designer did not intend
- Exploit (n.)
  - The code that is used to take advantage of a vulnerability
  - Also called a Proof of Concept (PoC)

0Day and Fuzzer

- 0Day
  - An exploit that has not been publicly disclosed
  - Sometimes used to refer to the vulnerability itself
- Fuzzer
  - A tool that sends a large range of unexpected input values to a system
  - The purpose is to find bugs which could later be exploited

Memory Management

- Specifically for Intel 32-bit architecture
- Most exploits we'll use involve overwriting or overflowing one portion of memory into another
- Understanding memory management is therefore crucial

Instructions and Data

- There is no intrinsic difference between data and executable instructions
  - Although there are some defenses like Data Execution Prevention
- They are both just a series of bytes
- This ambiguity makes system exploitation possible

Program Address Space

- Created when a program is run, including
  - Actual program instructions
  - Required data
- Three types of segments
  - **.text** contains program instructions (read-only)
  - **.data** contains static initialized global variables (writable)
  - **.bss** contains uninitialized global variables (writable)

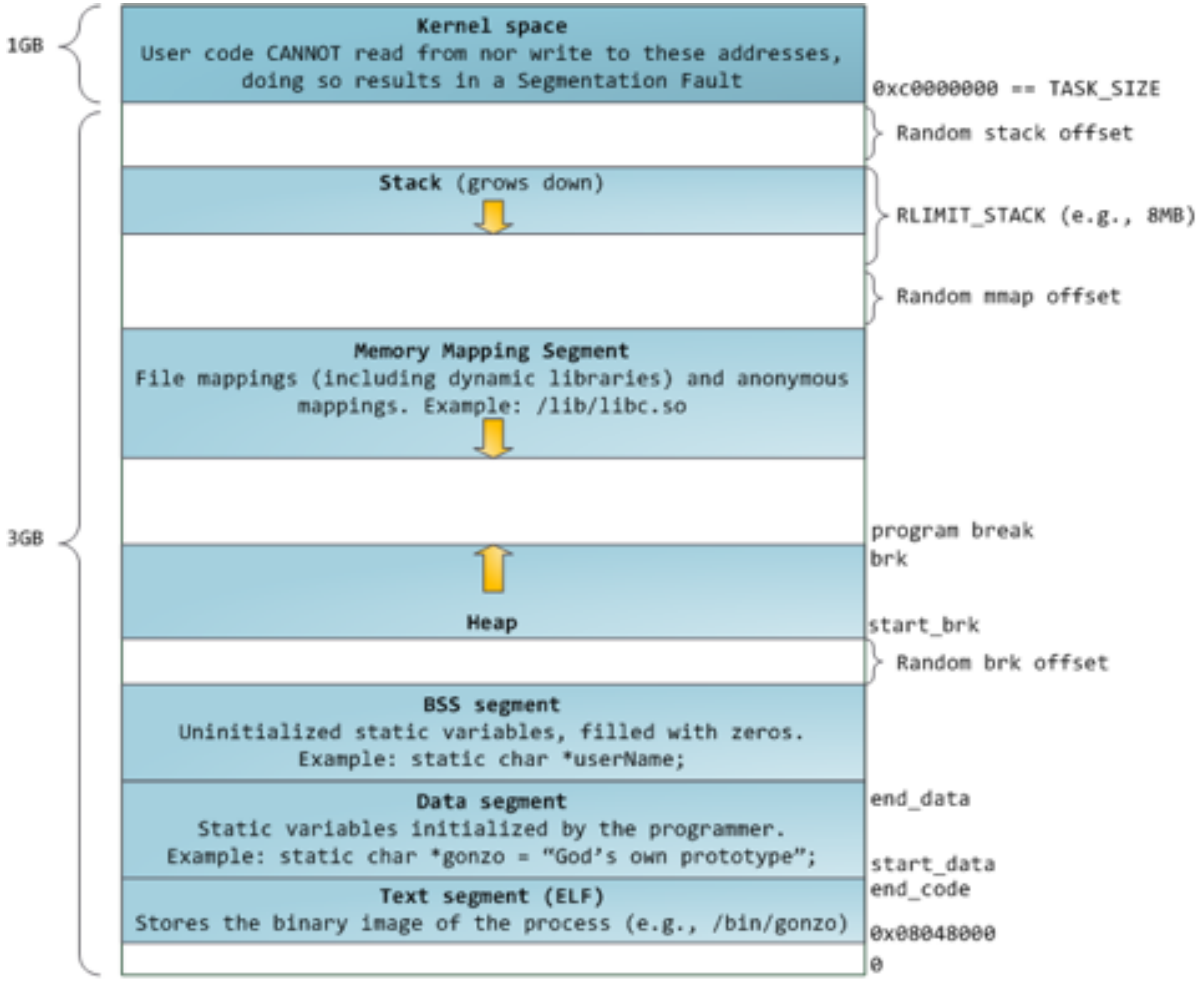
Stack

- Last In First Out (LIFO)
- Most recently **push**ed data is the first **pop**ped
- Ideal for storing transitory information
  - Local variables
  - Information relating to function calls
  - Other information used to clean up the stack after a function is called
- Grows down
  - As more data is added, it uses lower address values

Heap

- Holds dynamic variables
- Roughly First In First Out (FIFO)
- Grows up in address space

Program Layout in RAM



1GB

Kernel space

User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

0xc0000000 == TASK_SIZE

Random stack offset

Stack (grows down)

RLIMIT_STACK (e.g., 8MB)

Random mmap offset

Memory Mapping Segment

File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

3GB

program break

brk

Heap

start_brk

Random brk offset

BSS segment

Uninitialized static variables, filled with zeros. Example: static char *userName;

end_data

Data segment

Static variables initialized by the programmer. Example: static char *gonzo = "God's own prototype";

start_data

end_code

Text segment (ELF)

Stores the binary image of the process (e.g., /bin/gonzo)

0x08048000

0

- From link Ch 1a (.bss = Block Started by Symbols)

Assembly

Assembly Language

- Different versions for each type of processor
- x86 – 32-bit Intel (most common)
- x64 – 64-bit Intel
- SPARC, PowerPC, MIPS, ARM – others
- Windows runs on x86 or x64
- x64 machines can run x86 programs
- Most malware is designed for x86

Instructions

- **Mnemonic** followed by **operands**
- mov ecx 0x42
  - Move into Extended C register the value 42 (hex)
- mov ecx is 0xB9 in hexadecimal
- The value 42 is 0x4200000000
- In binary this instruction is
- 0xB942000000

Endianness

- Big-Endian
  - Most significant byte first
  - 0x42 as a 64-bit value would be 0x00000042
- Little-Endian
  - Least significant byte first
  - 0x42 as a 64-bit value would be 0x42000000
- Network data uses big-endian
- x86 programs use little-endian

IP Addresses

- 127.0.0.1, or in hex, 7F 00 00 01
- Sent over the network as 0x7F000001
- Stored in RAM as 0x0100007F

Operands

- **Immediate**
  - Fixed values like 0x42
- **Register**
  - eax, ebx, ecx, and so on
- **Memory address**
  - Denoted with brackets, like [eax]

Registers

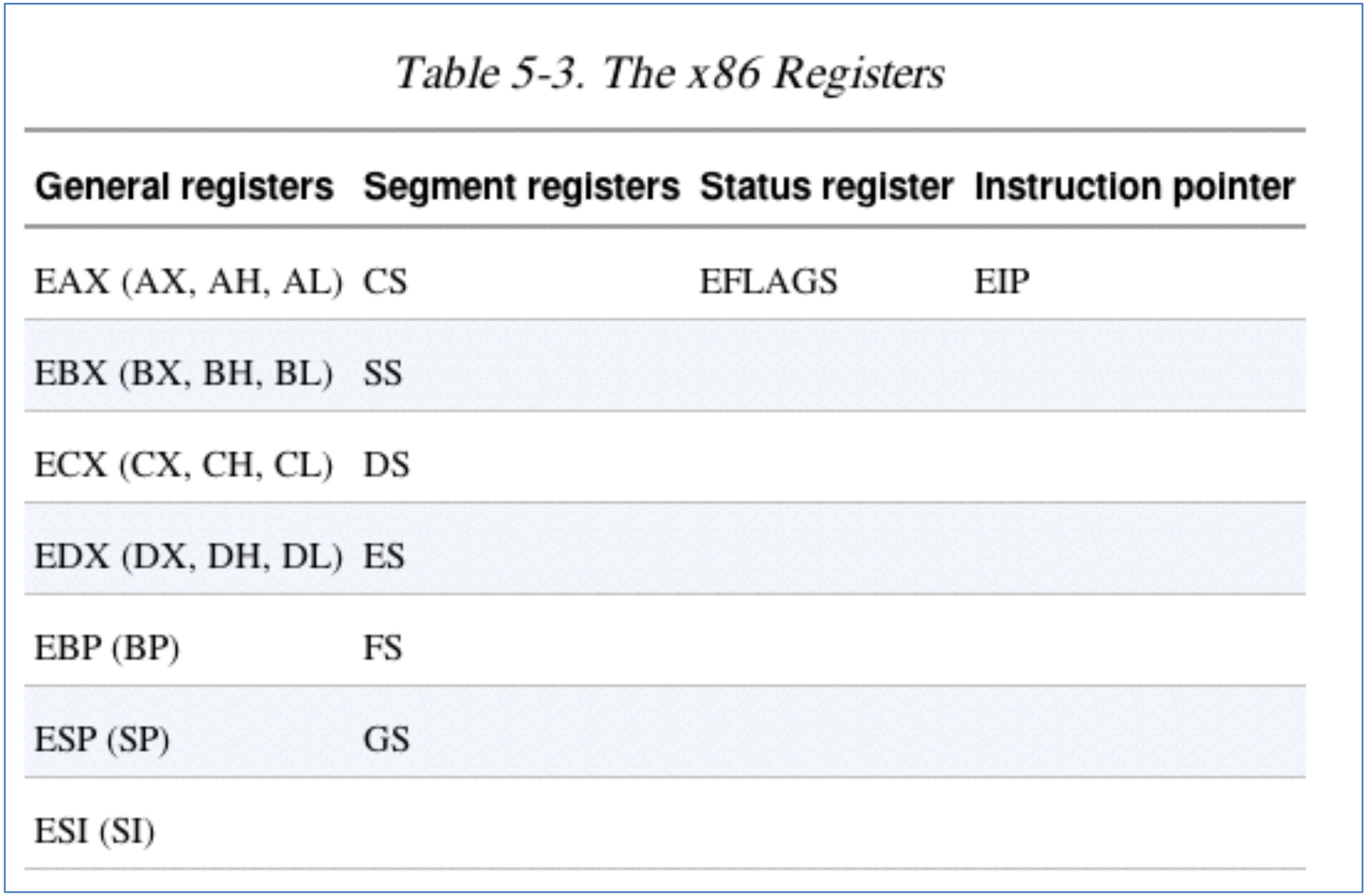


Table 5-3. The x86 Registers

General registers	Segment registers	Status register	Instruction pointer
EAX (AX, AH, AL)	CS	EFLAGS	EIP
EBX (BX, BH, BL)	SS		
ECX (CX, CH, CL)	DS		
EDX (DX, DH, DL)	ES		
EBP (BP)	FS		
ESP (SP)	GS		
ESI (SI)			

Registers

- General registers
  - Used by the CPU during execution
- Segment registers
  - Used to track sections of memory
- Status flags
  - Used to make decisions
- Instruction pointer
  - Address of next instruction to execute

Size of Registers

- General registers are all 32 bits in size
  - Can be referenced as either 32bits (edx) or 16 bits (dx)
- Four registers (eax, ebx, ecx, edx) can also be referenced as 8-bit values
  - AL is lowest 8 bits
  - AH is higher 8 bits

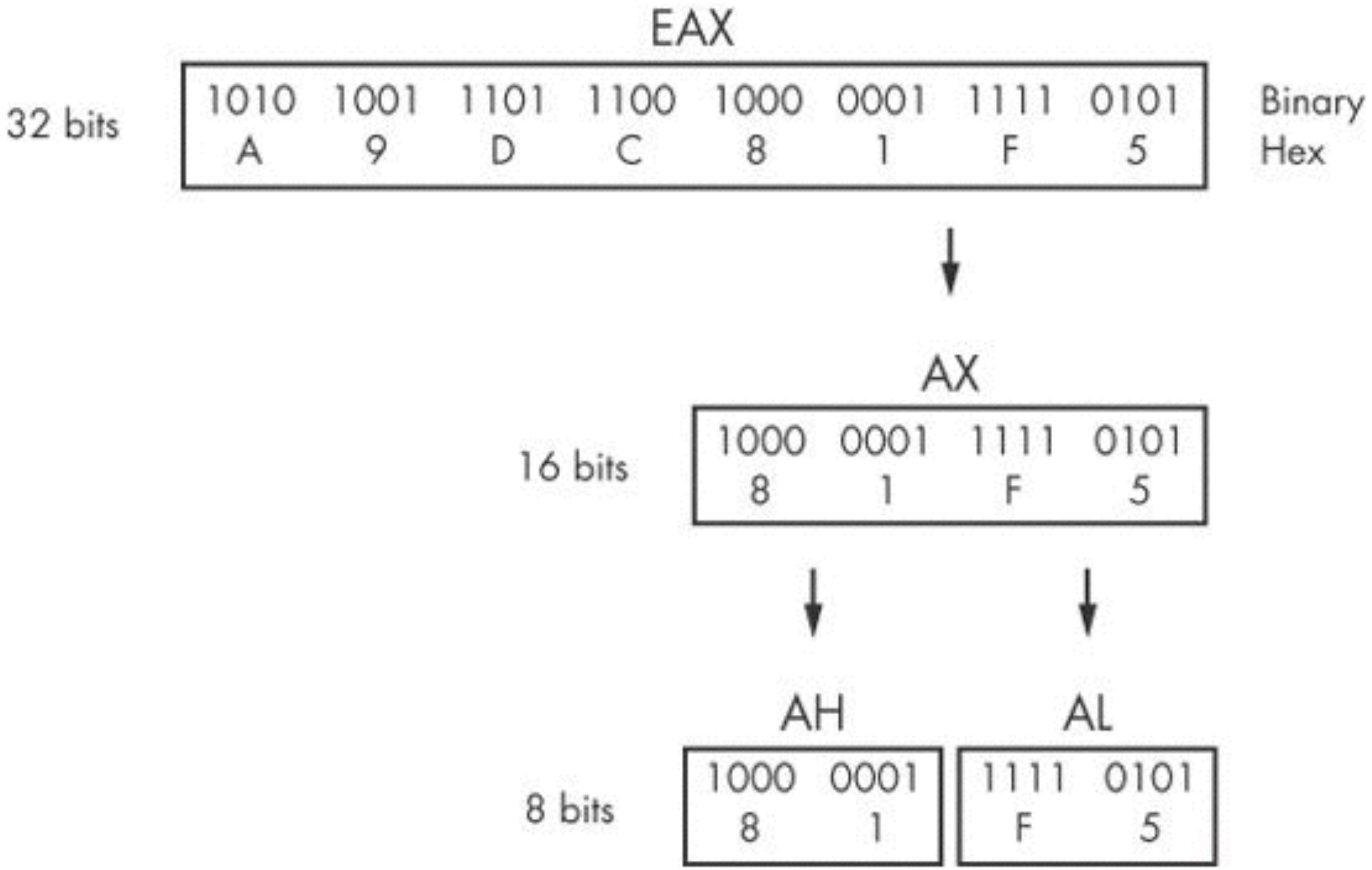


Figure 5-4. x86 EAX register breakdown

General Registers

- Typically store data or memory addresses
- Normally interchangeable
- Some instructions reference specific registers
  - Multiplication and division use EAX and EDX
- **Conventions**
  - Compilers use registers in consistent ways
  - EAX contains the return value for function calls

Flags

- EFLAGS is a status register
- 32 bits in size
- Each bit is a flag
- SET (1) or Cleared (0)

Important Flags

- **ZF** Zero flag
  - Set when the result of an operation is zero
- **CF** Carry flag
  - Set when result is too large or small for destination
- **SF** Sign Flag
  - Set when result is negative, or when most significant bit is set after arithmetic
- **TF** Trap Flag
  - Used for debugging—if set, processor executes only one instruction at a time

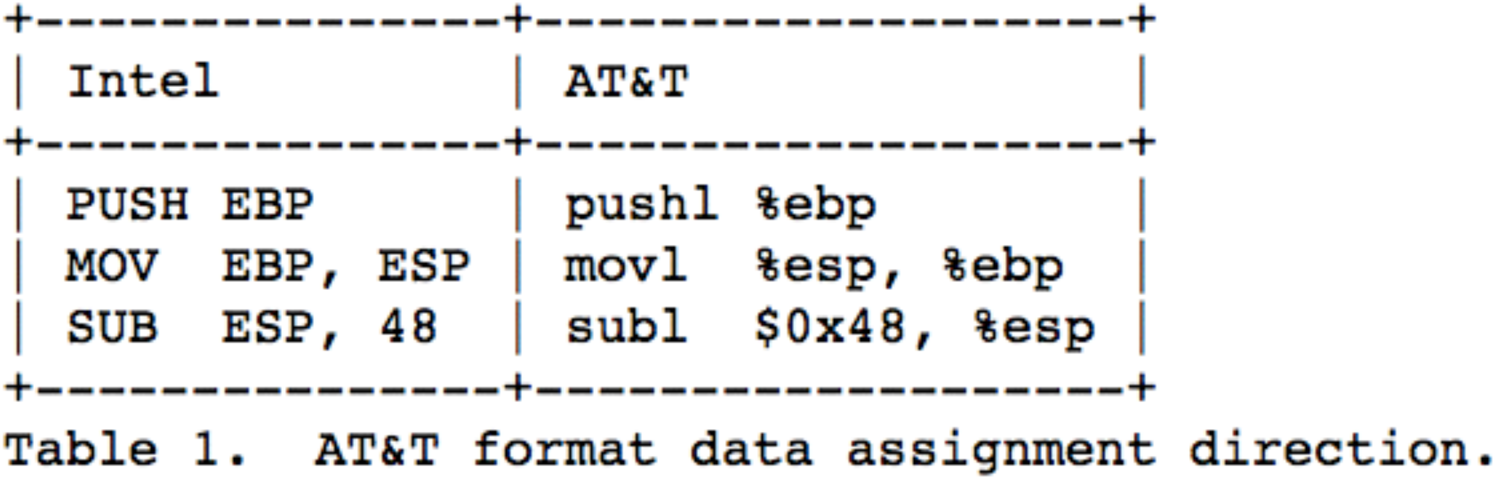
EIP (Extended Instruction Pointer)

- Contains the memory address of the next instruction to be executed
- If EIP contains wrong data, the CPU will fetch non-legitimate instructions and crash
- Buffer overflows target EIP

Simple Instructions

Simple Instructions

- mov destination, source
  - Moves data from one location to another
- Intel format is favored by Windows developers, with destination first



Intel	AT&T
PUSH EBP	pushl %ebp
MOV EBP, ESP	movl %esp, %ebp
SUB ESP, 48	subl $0x48, %esp

Table 1. AT&T format data assignment direction.

Simple Instructions

- Remember indirect addressing
  - [ebx] means the memory location pointed to by EBX

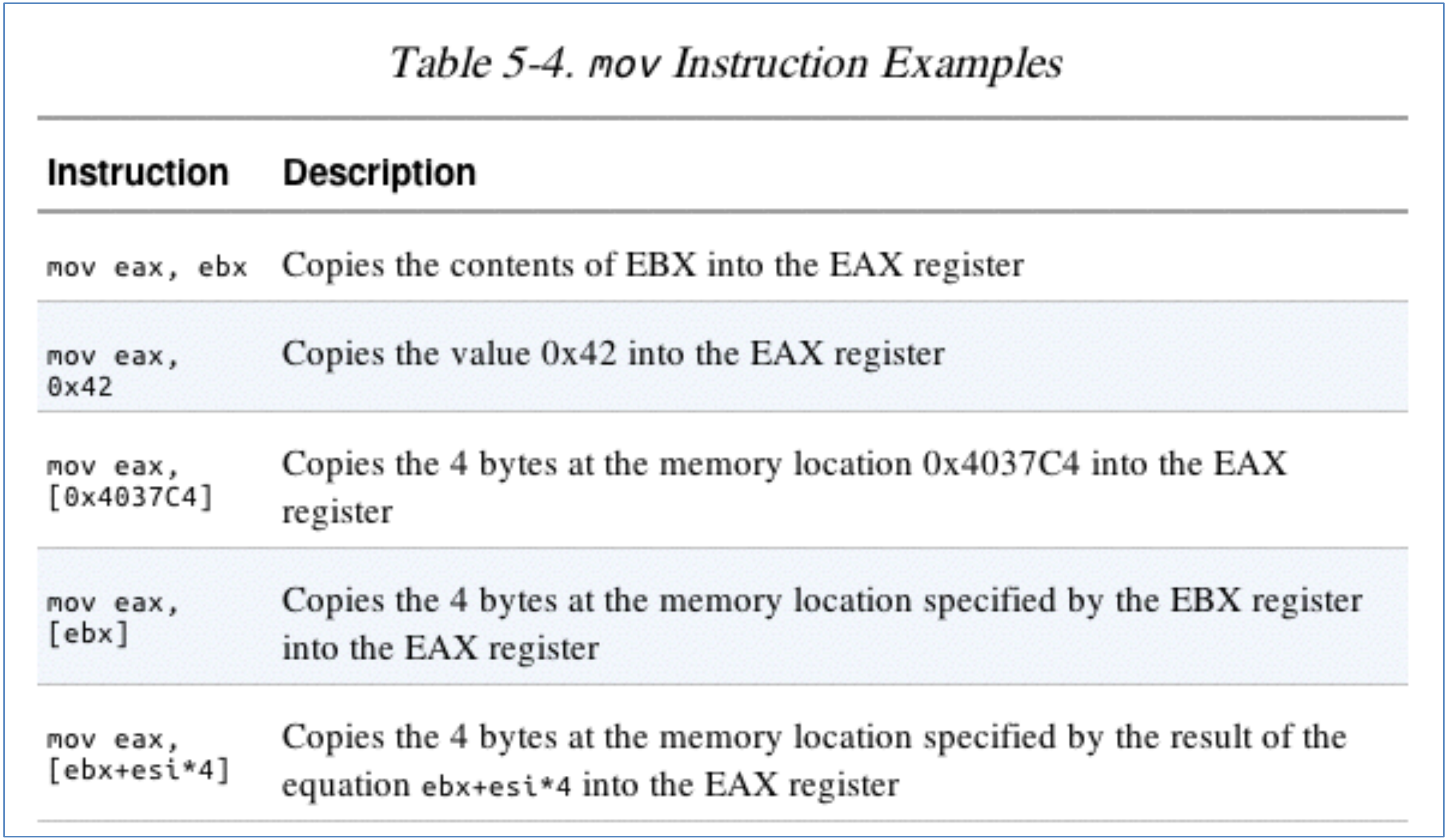


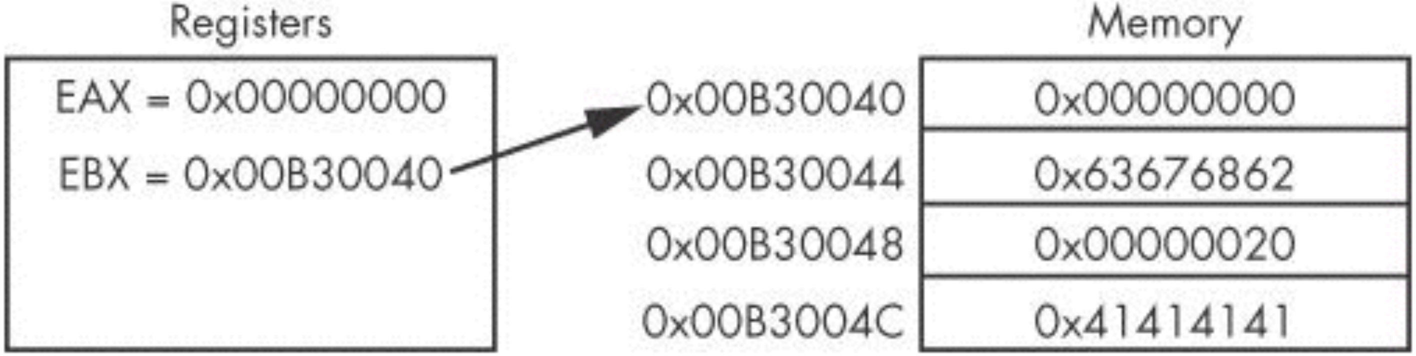
Table 5-4. `mov` Instruction Examples

Instruction	Description
`mov eax, ebx`	Copies the contents of EBX into the EAX register
`mov eax, 0x42`	Copies the value 0x42 into the EAX register
`mov eax, [0x4037C4]`	Copies the 4 bytes at the memory location 0x4037C4 into the EAX register
`mov eax, [ebx]`	Copies the 4 bytes at the memory location specified by the EBX register into the EAX register
`mov eax, [ebx+esi*4]`	Copies the 4 bytes at the memory location specified by the result of the equation `ebx+esi*4` into the EAX register

lea (Load Effective Address)

- lea destination, source
- lea eax, [ebx+8]
  - Puts ebx + 8 into eax
- Compare
  - mov eax, [ebx+8]
  - Moves the data at location ebx+8 into eax

Figure 5-5 shows values for registers EAX and EBX on the left and the information contained in memory on the right. EBX is set to 0xB30040. At address 0xB30048 is the value `0x20`. The instruction `mov eax, [ebx+8]` places the value `0x20` (obtained from memory) into EAX, and the instruction `lea eax, [ebx+8]` places the value `0xB30048` into EAX.



0x00B30040	0x00000000
0x00B30044	0x63676862
0x00B30048	0x00000020
0x00B3004C	0x41414141

Figure 5-5. EBX register used to access memory

Arithmetic

- **sub** Subtracts
- **add** Adds
- **inc** Increments
- **dec** Decrements
- **mul** Multiplies
- **div** Divides

NOP

- Does nothing
- 0x90
- Commonly used as a **NOP Sled**
- Allows attackers to run code even if they are imprecise about jumping to it

The Stack

- Memory for functions, local variables, and flow control
- Last in, First out
- ESP (Extended Stack Pointer) – top of stack
- EBP (Extended Base Pointer) – bottom of stack
- PUSH puts data on the stack
- POP takes data off the stack

Other Stack Instructions

- All used with functions
  - Call
  - Leave
  - Enter
  - Ret

Function Calls

- Small programs that do one thing and return, like printf()
- Prologue
  - Instructions at the start of a function that prepare stack and registers for the function to use
- Epilogue
  - Instructions at the end of a function that restore the stack and registers to their state before the function was called

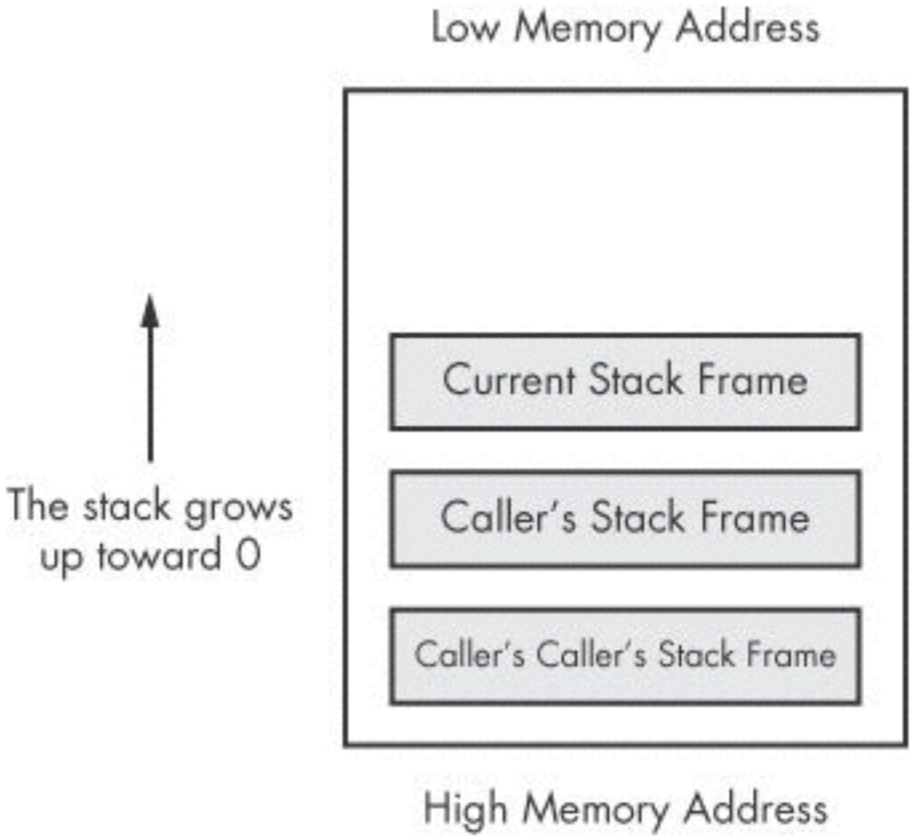


Figure 5-7. x86 stack layout

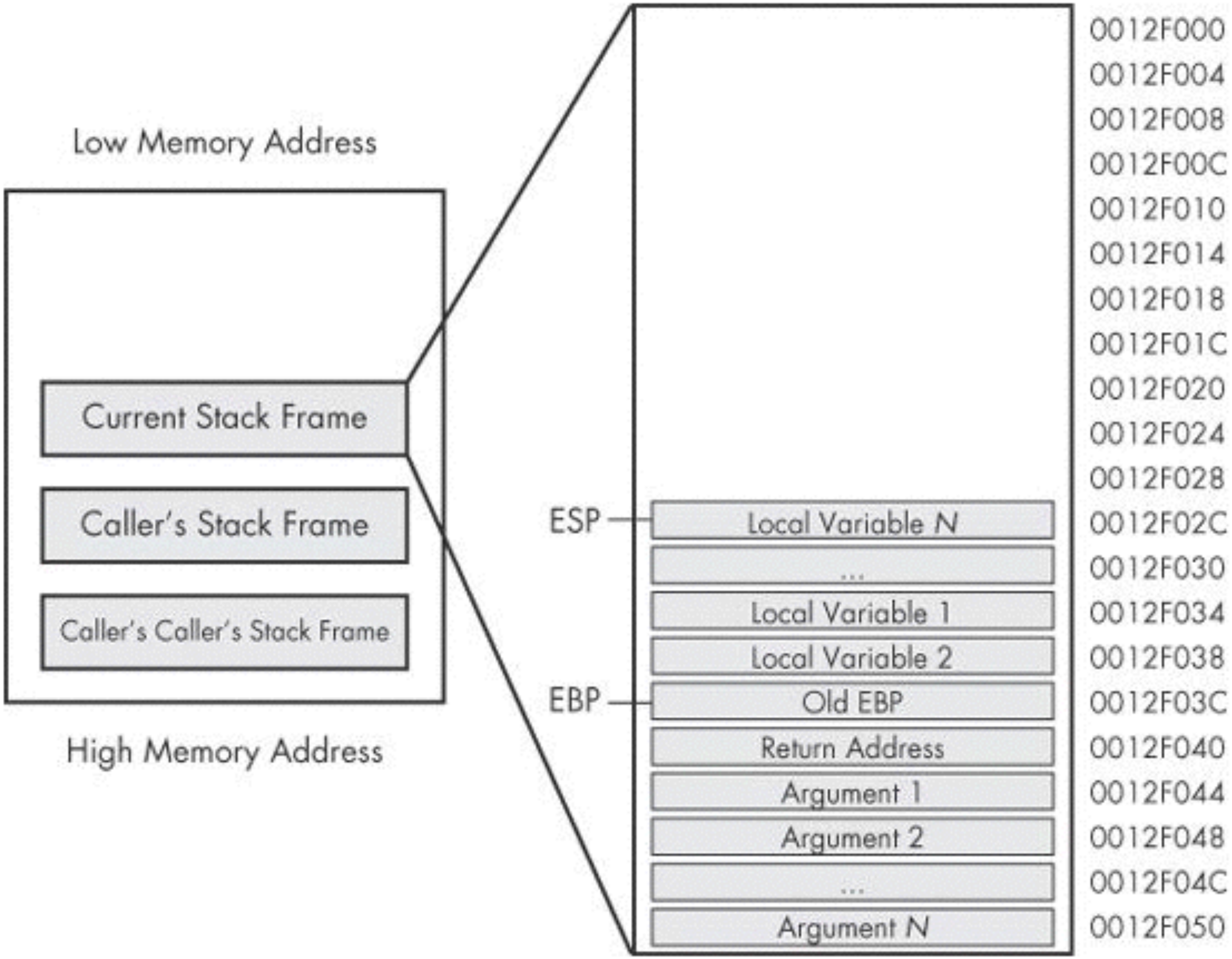


Figure 5-8. Individual stack frame

Conditionals

- test
  - Compares two values the way AND does, but does not alter them
  - test eax, eax
    - Sets Zero Flag if eax is zero
- cmp eax, ebx
  - Sets Zero Flag if the arguments are equal

Branching

- jz loc
  - Jump to loc if the Zero Flag is set
- jnz loc
  - Jump to loc if the Zero Flag is cleared

C Main Method

- Every C program has a main() function
- int main(int argc, char** argv)
  - argc contains the number of arguments on the command line
  - argv is a pointer to an array of names containing the arguments

Example

- cp foo bar
- argc = 3
- argv[0] = cp
- argv[1] = foo
- argv[2] = bar

Recognizing C Constructs in Assembly

Incrementing

C

```
int number;
...
number++;
```

Assembler

```
number dw 0
...
mov eax, number
inc eax
mov number, eax
```

- dw: Define Word

Incrementing

C

```
int number;
if (number<0)
{
...
}
```

Assembler

```
number dw 0
mov eax, number
or eax, eax
jge label
...
label :
```

- or compares numbers, like test (link Ch 1b)

Array

C

```
int array[4];
...
array[2]=9;
```

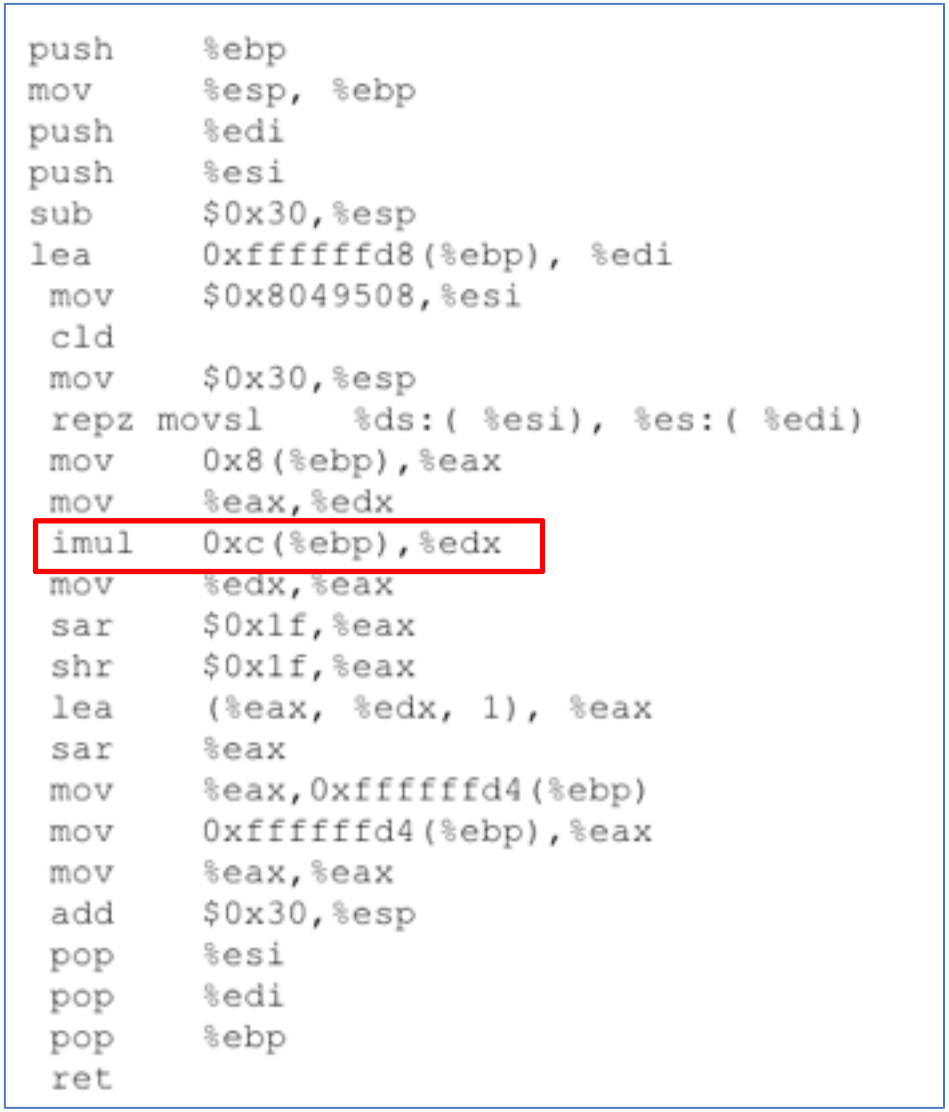
Assembler

```
array dw 0,0,0,0
...
mov ebx, 2
mov array[ebx], 9
```

Triangle

C

```
int triangle (int
width, int height)
{
int array[5] =
{0,1,2,3,4};
int area
area = width *
height/2;
return (area);
}
```



```
push     %ebp
mov      %esp, %ebp
push     %edi
push     %esi
sub      $0x30,%esp
lea      0xffffffd8(%ebp), %edi
 mov     $0x8049508,%esi
 cld
 mov     $0x30,%esp
 repz movsl     %ds:( %esi), %es:( %edi)
 mov     0x8(%ebp),%eax
 mov     %eax,%edx
 imul    0xc(%ebp),%edx
 mov     %edx,%eax
 sar     $0x1f,%eax
 shr     $0x1f,%eax
 lea     (%eax, %edx, 1), %eax
 sar     %eax
 mov     %eax,0xffffffd4(%ebp)
 mov     0xffffffd4(%ebp),%eax
 mov     %eax,%eax
 add     $0x30,%esp
 pop     %esi
 pop     %edi
 pop     %ebp
 ret
```