Practical Malware Analysis

Chapter 4: Crash Course in x86 Disassembly

Akbar Namin

Texas Tech University

Fall 2021

Reference:

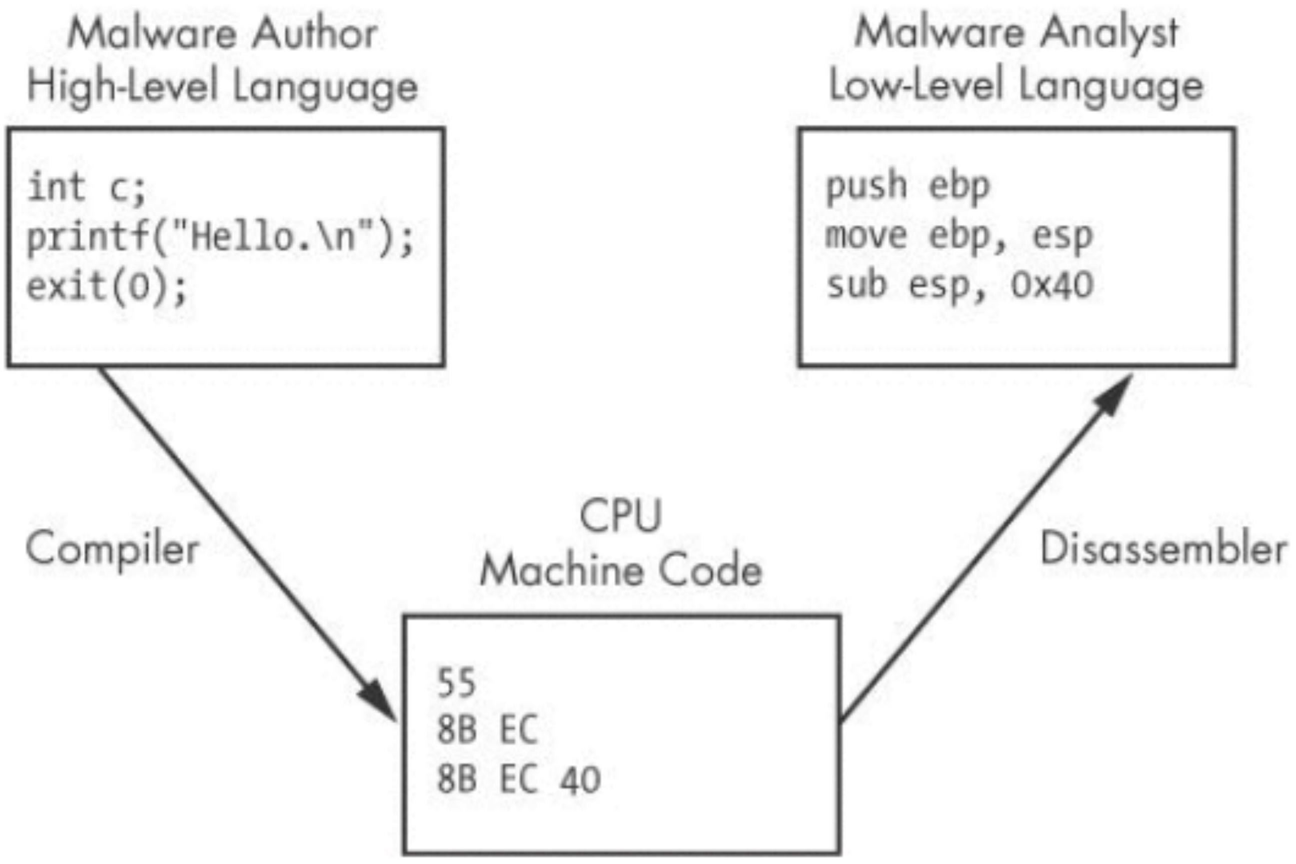
Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software 1st Edition

by Michael Sikorski (Author), Andrew Honig (Author)

Basic Techniques

- Basic static analysis
  - Looks at malware from the outside
- Basic dynamic analysis
  - Only shows you how the malware operates in one case
- Disassembly
  - View code of malware & figure out what it does

Levels of Abstraction



Code level example

Six Levels of Abstraction

- Hardware
- Microcode
- Machine code
- Low-level languages
- High-level languages
- Interpreted languages

- Hardware
  - Digital circuits
  - XOR, AND, OR, NOT gates
  - Cannot be easily manipulated by software
- Microcode
  - Also called firmware
  - Only operates on specific hardware it was designed for
  - Not usually important for malware analysis

- Machine code
  - Opcodes
    - Tell the processor to do something
    - Created when a program written in a high-level language is compiled
- Low-level languages
  - Human-readable version of processor's instruction set
  - Assembly language
    - PUSH, POP, NOP, MOV, JMP ...
  - Disassembler generates assembly language
    - This is the highest-level language that can be reliably recovered from malware when source code is unavailable

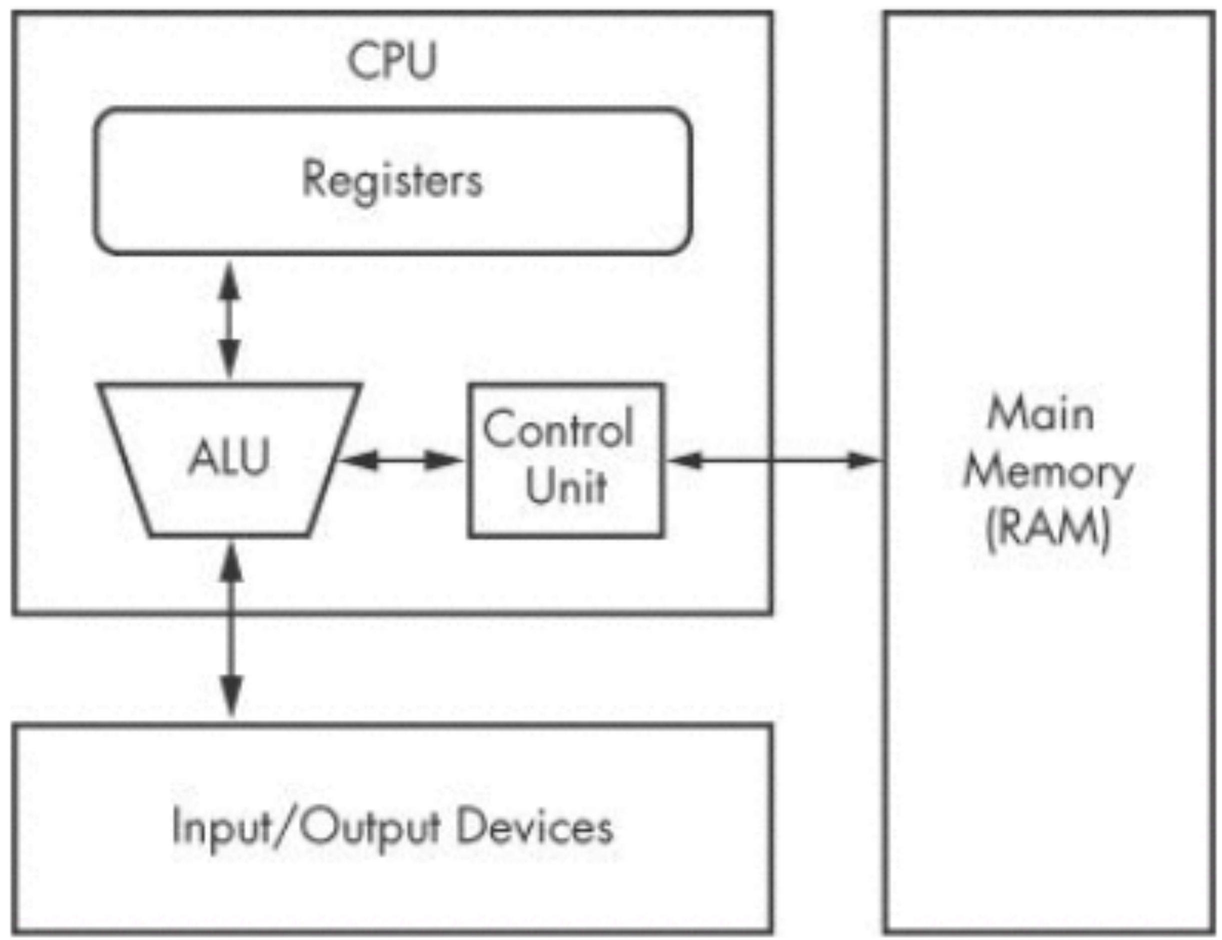
- High-level languages
  - Most programmers use these
  - C, C++, etc.
  - Converted to machine code by a compiler
- Interpreted languages
  - Highest level
  - Java, C#, Perl, .NET, Python
  - Code is not compiled into machine code
  - It is translated into bytecode
    - An intermediate representation
    - Independent of hardware and OS
    - Bytecode executes in an interpreter, which translates bytecode into machine language on the fly at runtime
- Ex: Java Virtual Machine

Reverse Engineering

- Disassembly
  - Malware on a disk is in binary form at the machine code level
  - Disassembly converts the binary form to assembly language
  - IDA Pro is the most popular disassembler
- 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

The x86 Architecture

- CPU (Central Processing Unit) executes code
- RAM stores all data and code
- I/O system interfaces with hard disk, keyboard, monitor, etc.

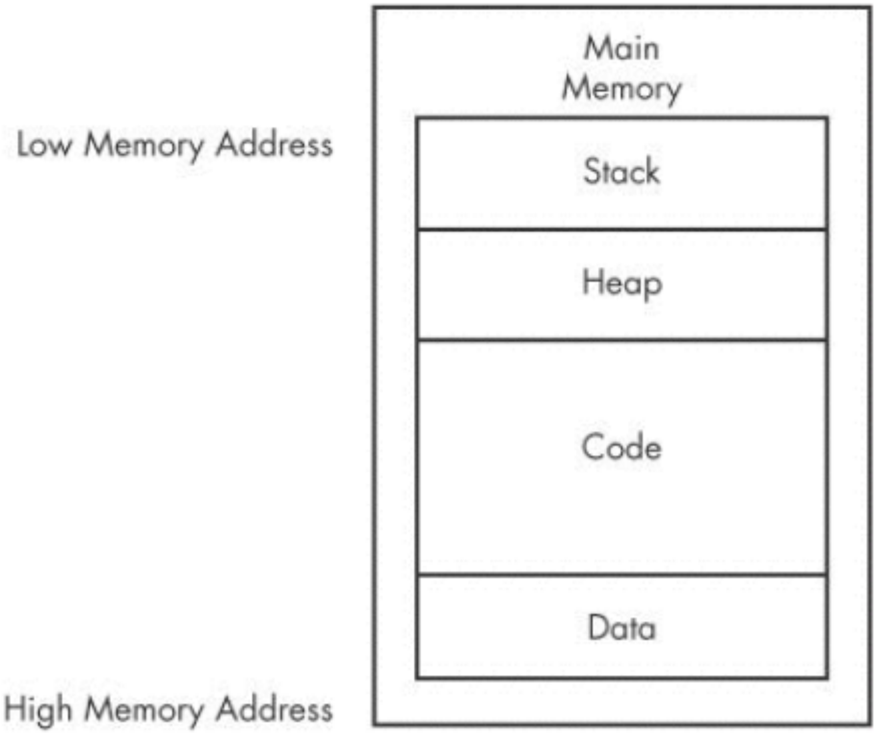


Von Neumann architecture

CPU Components

- Control unit
  - Fetches instructions from RAM using a register named the instruction pointer
- Registers
  - Data storage within the CPU
  - Faster than RAM
- ALU (Arithmetic Logic Unit)
  - Executes an instruction and places results in registers or RAM

Main Memory (RAM)



Basic memory layout for a program

Data

- Values placed in RAM when a program loads
- Sometimes these values are called static
  - They may not change while the program is running
- Sometimes these values are called global
  - Available to any part of the program

❑Code

- Instructions for the CPU
- Controls what the program does

❑**Heap**

- Dynamic memory
- Changes frequently during program execution
- Program creates (allocates) new values ,and eliminates (frees) them when they are no longer needed

- Stack
  - Local variables and parameters for functions
  - Helps programs flow
- 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

Assembly Language Instructions

Mnemonic	Destination operand	Source operand
mov	ecx	0x42

Instruction Format

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

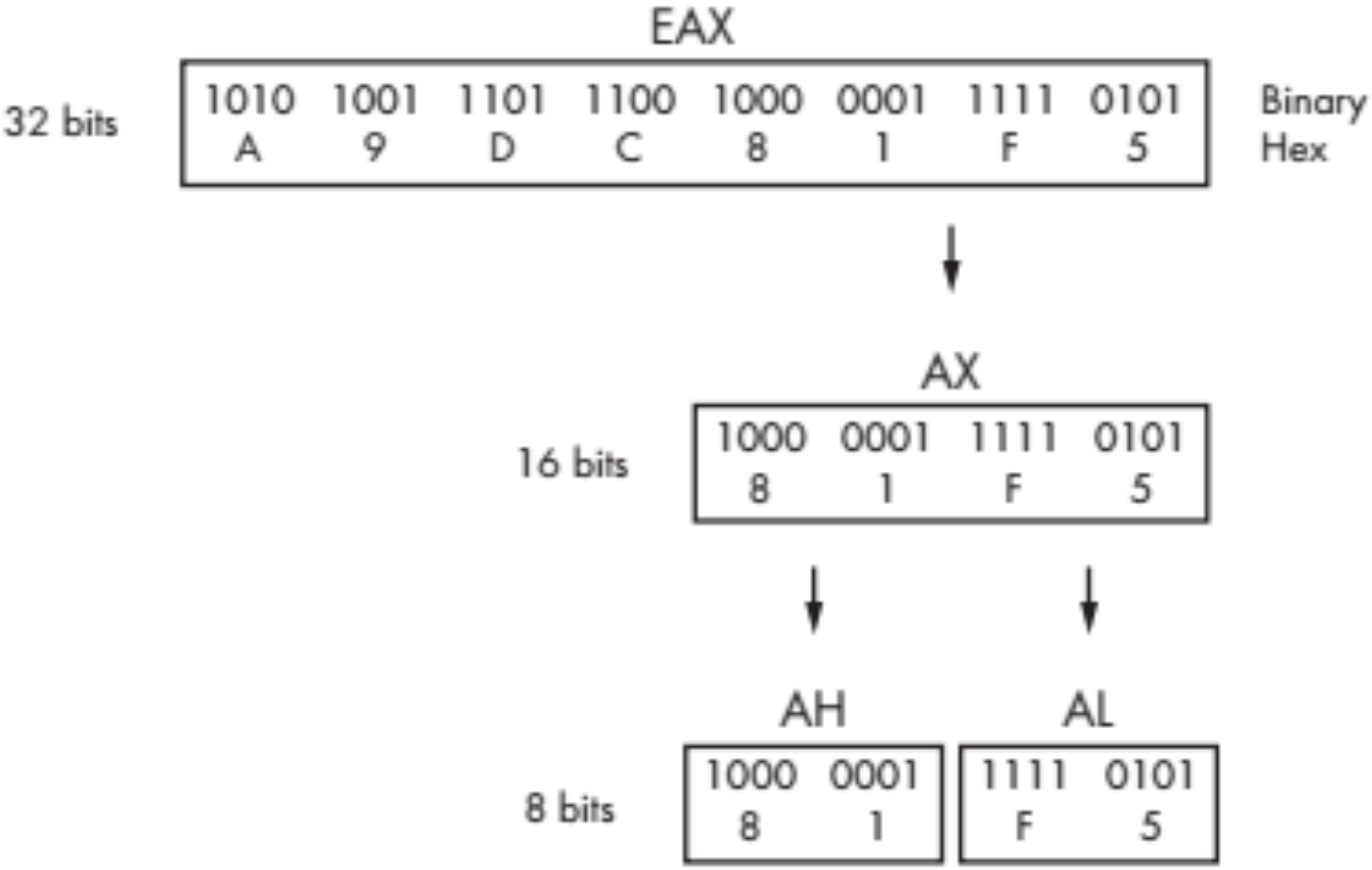
General registers	Segment registers	Status register	Instruction pointer
EAX (AX, AH, AL)	CS	EFLAGS	EIP
EBX (BX, BH, BL)	SS		
ECX (CX, CH, CL)	DS		
EDX (DX, DH, DL)	ES		
EBP (BP)	FS		
ESP (SP)	GS		
ESI (SI)			

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



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

- mov destination, source
  - Moves data from one location to another
- We use Intel format throughout the book, with destination first
- Remember indirect addressing
  - [ebx] means the memory location pointed to by EBX

mov Instruction Examples

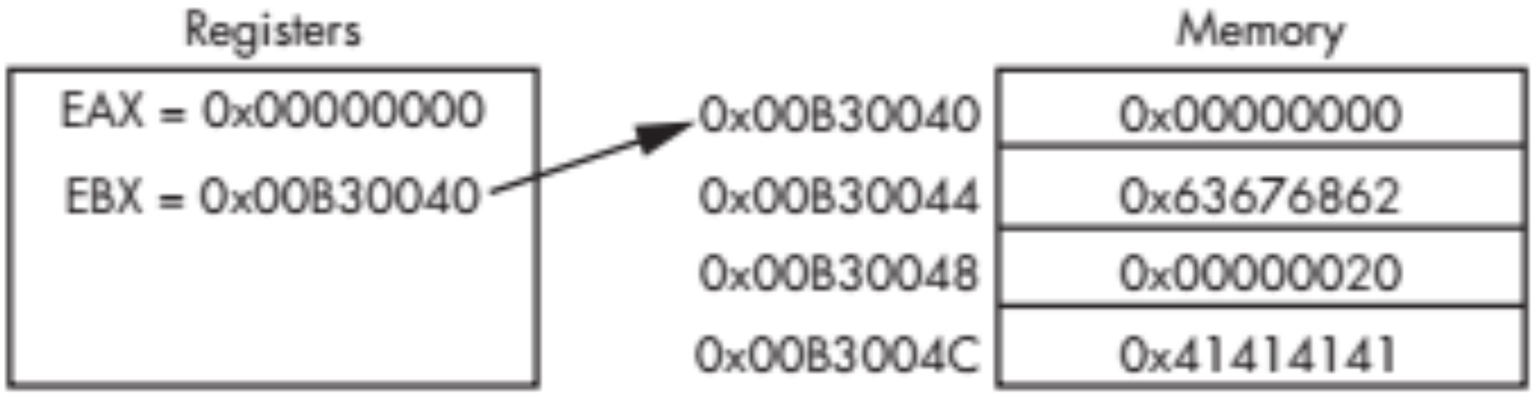
Instruction	Description
`mov eax, ebx`	Copies the contents of EBX into the EAX register
`mov eax, 0x42`	Copies the value 0x42 into the EAX register
`mov eax, [0x4037C4]`	Copies the 4 bytes at the memory location 0x4037C4 into the EAX register
`mov eax, [ebx]`	Copies the 4 bytes at the memory location 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

EBX register used to access memory

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



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

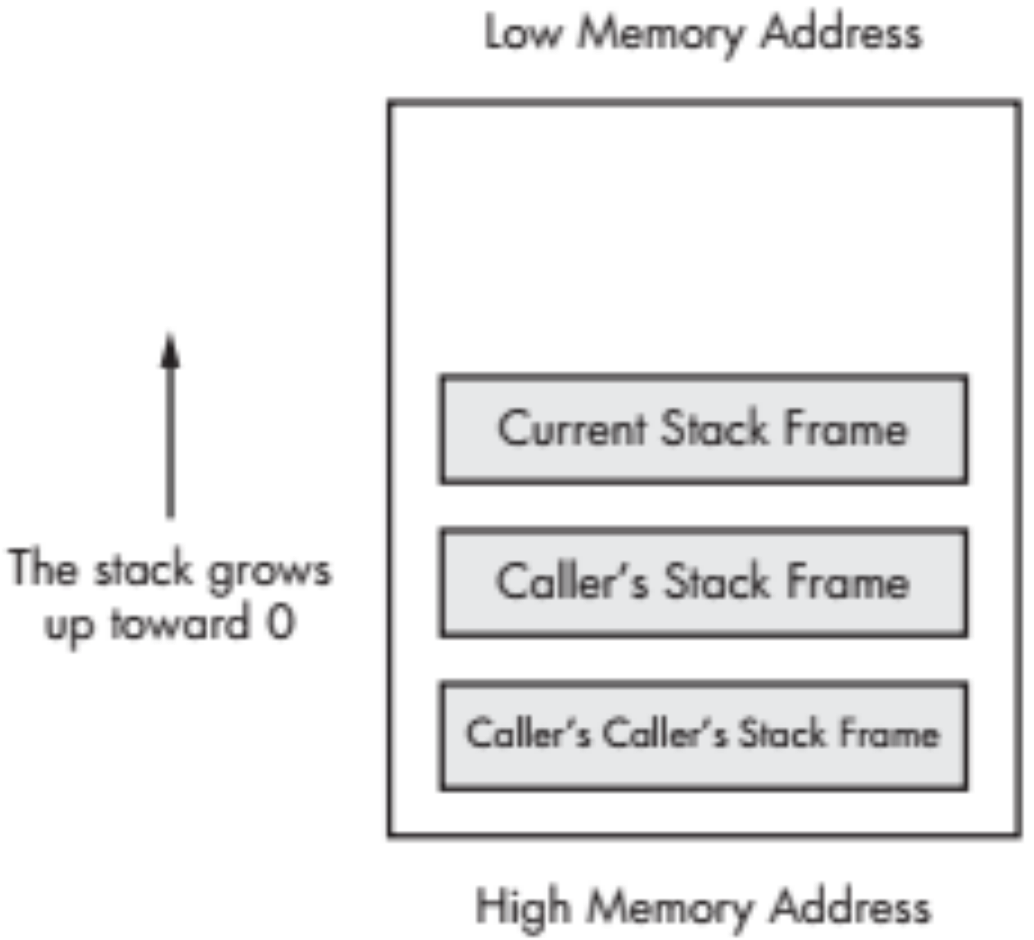
The Stack

- 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
  - To enter a function
    - Call or Enter
  - To exit a function
    - Leave or Ret

Function Calls

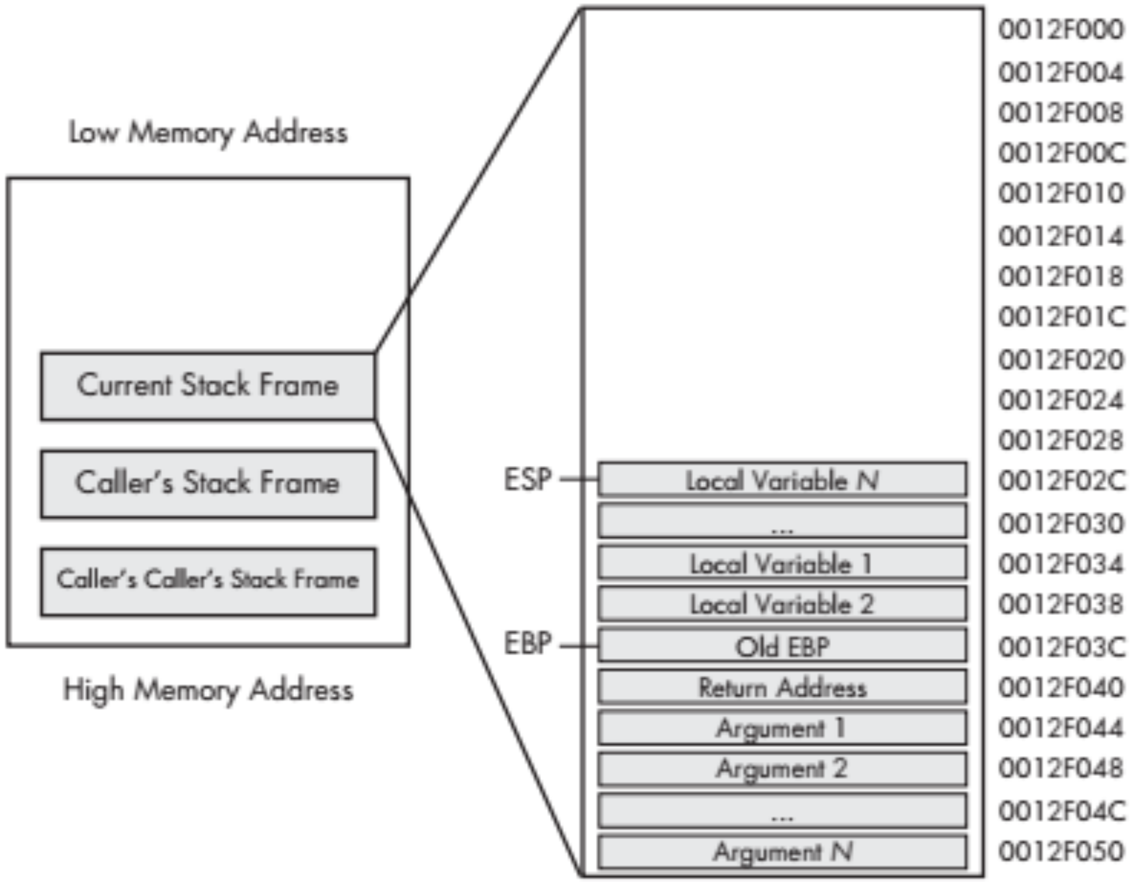
- Small programs that do one thing and return, like printf()
- Prologue
  - Instructions at the start of a function that prepare stack and registers for the function to use
- Epilogue
  - Instructions at the end of a end of a function that restore the stack and registers to their state before the function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

Stack Frames



x86 stack layout

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