SIGPwny @ UIUC

Intro to Reverse Engineering

Announcements

- DefCamp Quals this weekend
- Semester ramp-up

News of the Week

- Deloitte Breach
- Adobe Publishes PGP Key
- Broadcom WiFi Overflow Released
- Derbycon 2017 Videos
- P0 DOM Fuzzing Post
- Signal Private Contact Discovery

Reverse Engineering

What is RE

Reverse engineering is the process of understanding a program’s functionality and behavior though the analysis of code

What is RE used for

- Understanding software
- Adding features to legacy code and poorly documented code
- Deciphering proprietary file format and protocols
- Finding bugs and vulnerabilities in code
- Analyzing behavior of malware
- Cracking software

What is RE used for

- You work for a security consultant and get a malware sample
- You want to know:
  - How it works
  - What files it affects
  - Custom obfuscated network protocols

What is RE used for

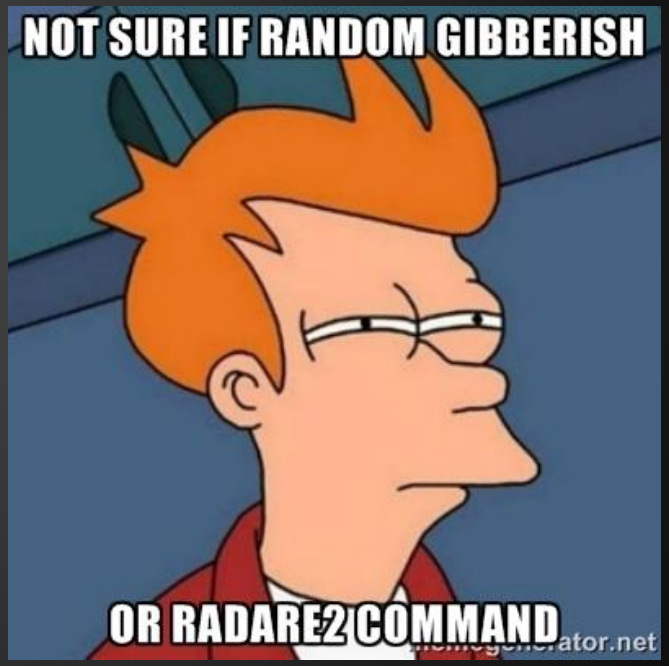
- “I wonder how this electronic voter box works”
- “I wonder if it does authentication”
  - “I bet it doesn’t”

What is RE used for

- You work for a company that's been around since the '70s
- They are upgrading for the first time
- "We have this great tool but it won't be compatible on the new system. The guys who built this are now retired. Can you port it?"

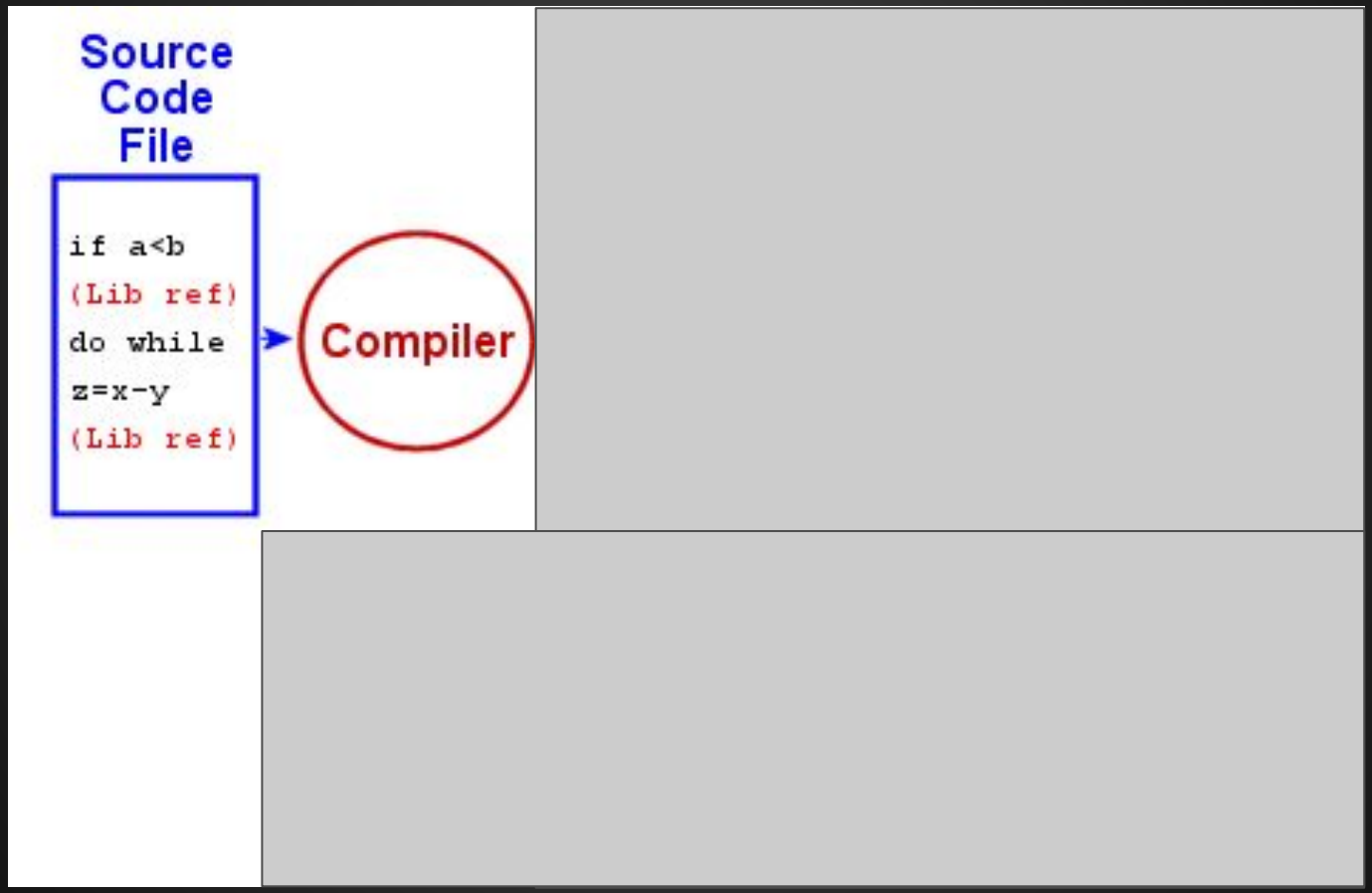
Tools used in RE

- Disassemblers/Decompilers
  - IDA Pro/Hex Rays
  - Hopper
  - Radare2
  - Binary Ninja
- Debuggers
  - OllyDbg
  - GDB
  - WinDBG

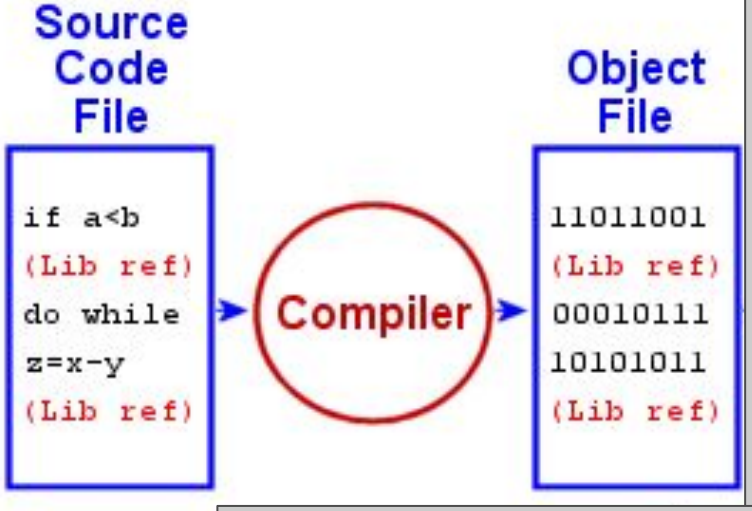


Compiling

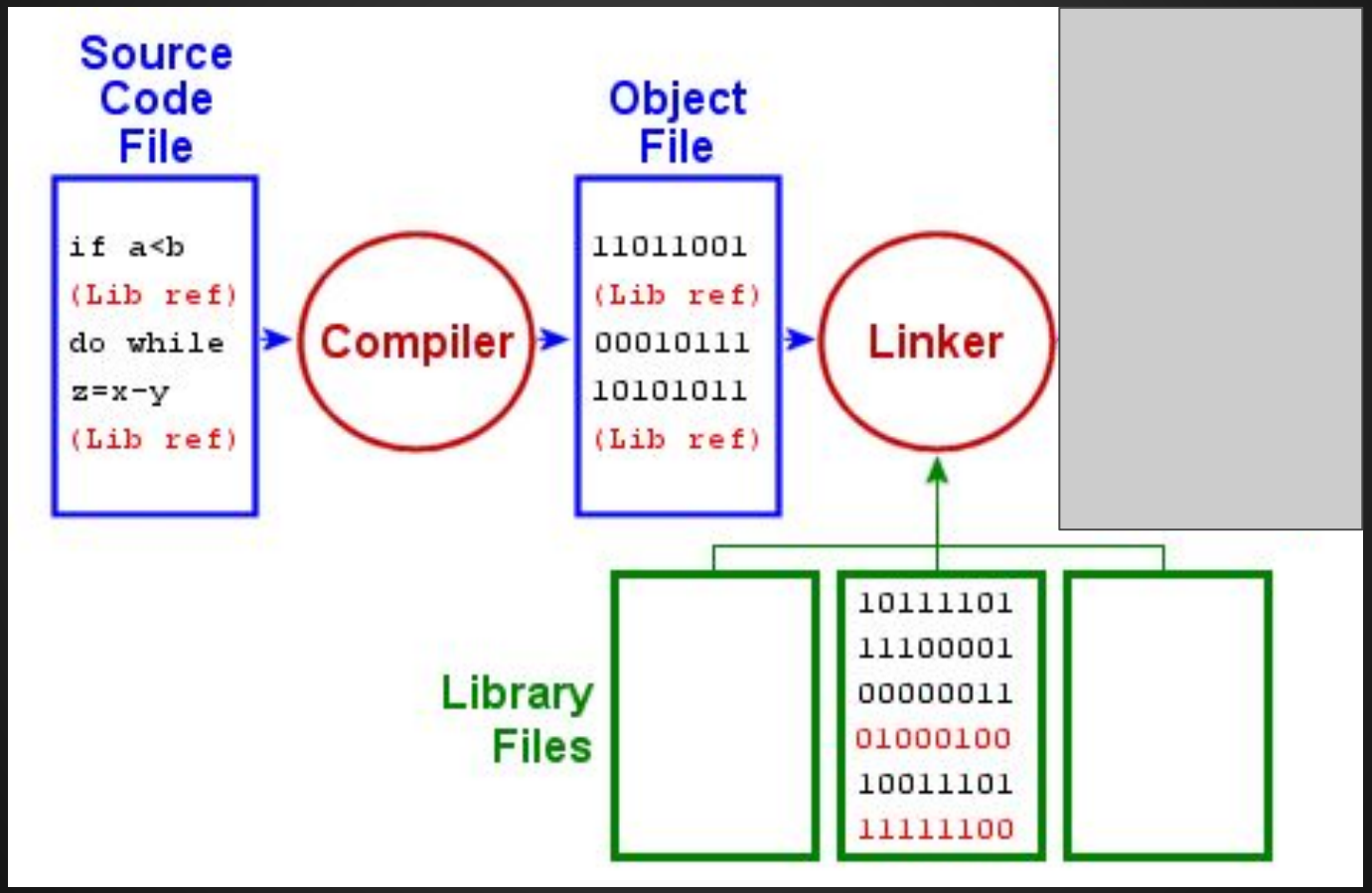
Compiling



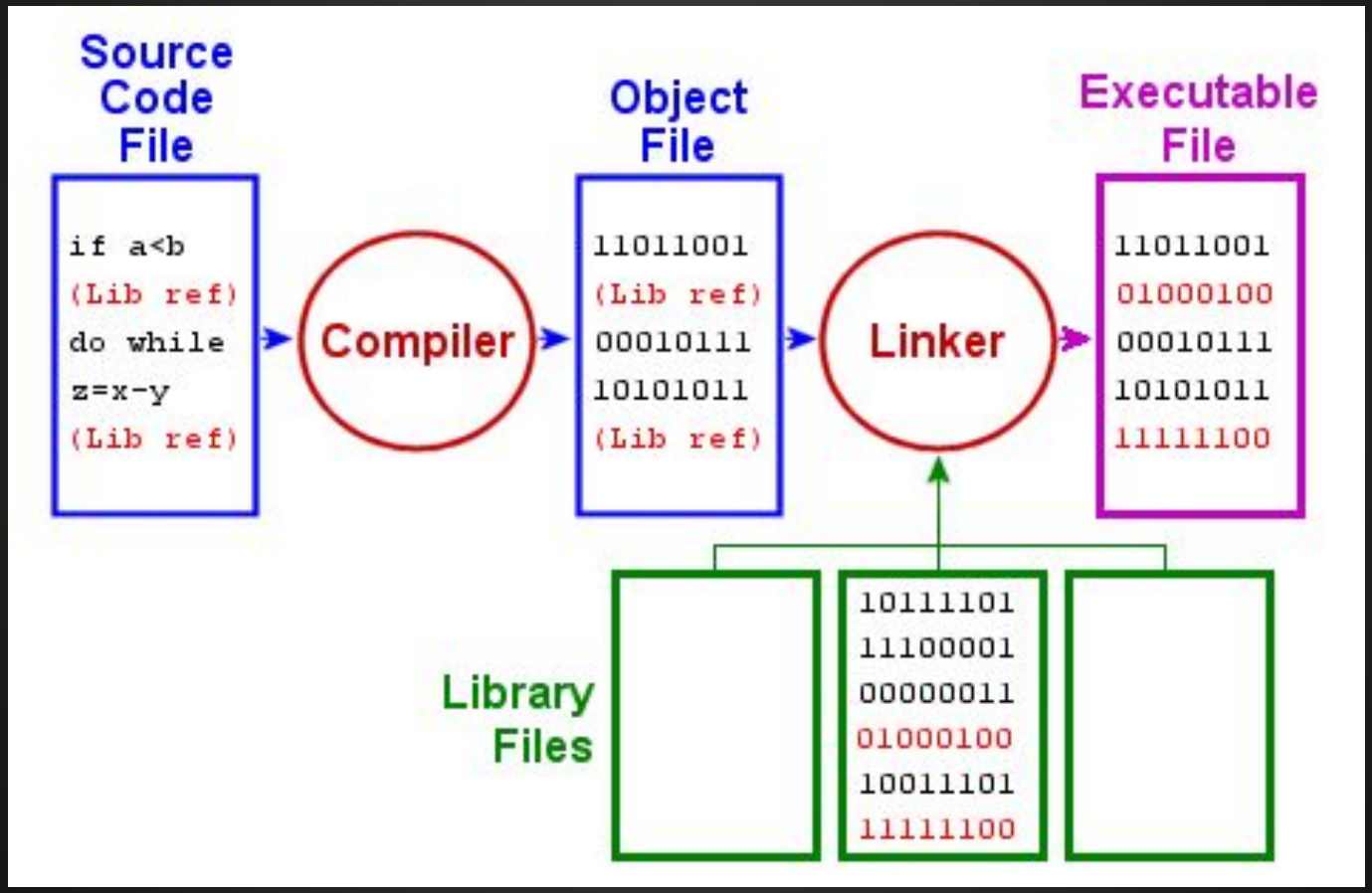
Compiling


Source Code File
if a<b
(Lib ref)
do while
z=x-y
(Lib ref)
Compiler
Object File
11011001
(Lib ref)
00010111
10101011
(Lib ref)

Compiling

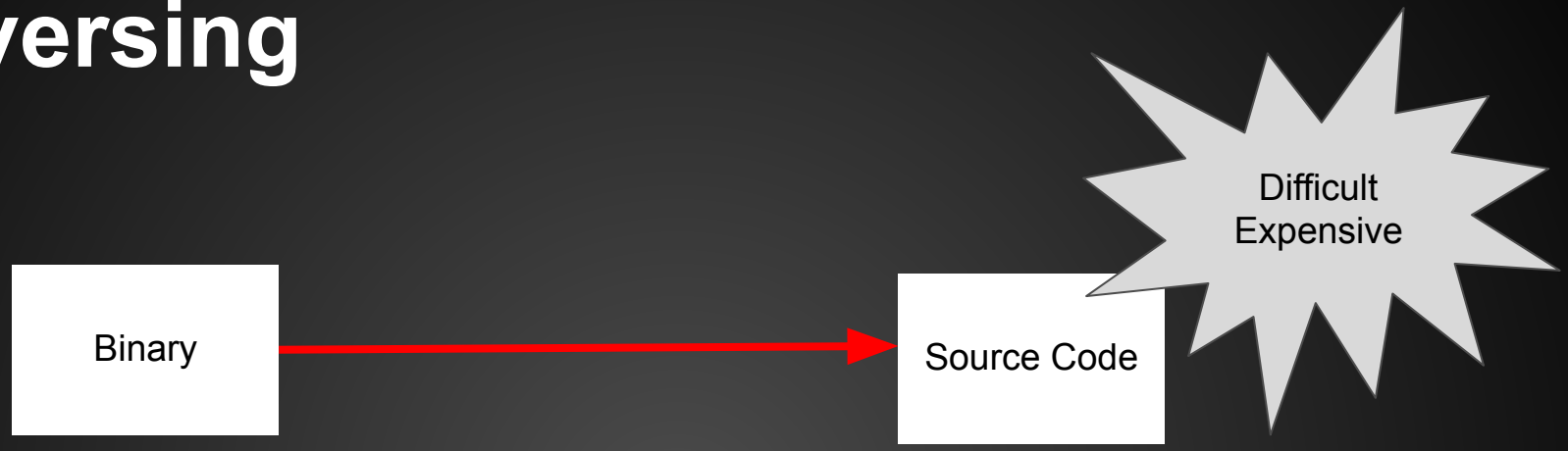

Source Code File
if a<b
(Lib ref)
do while
z=x-y
(Lib ref)
Compiler
Object File
11011001
(Lib ref)
00010111
10101011
(Lib ref)
Linker
Library Files
10111101
11100001
00000011
01000100
10011101
11111100

Compiling

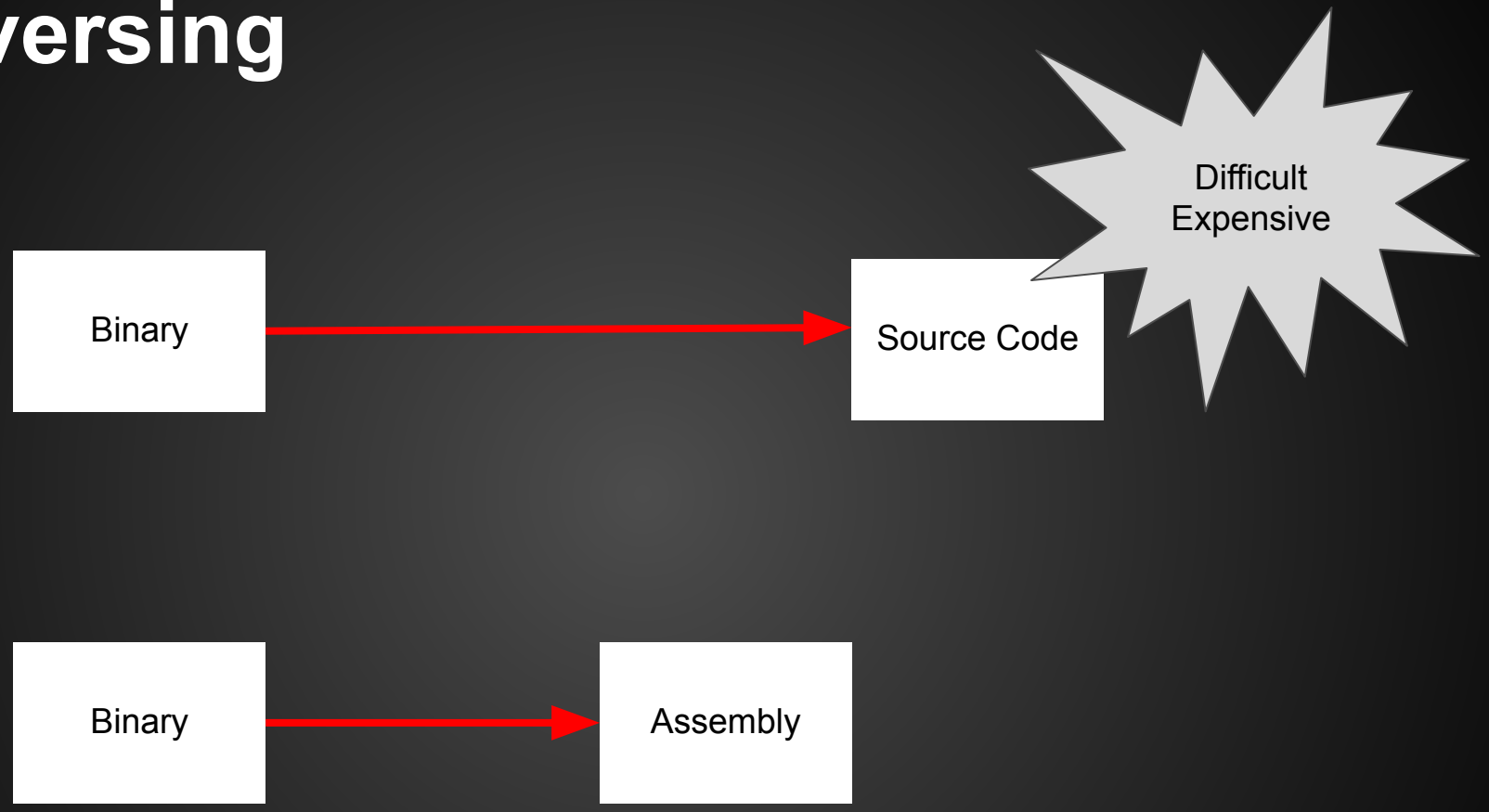

Source Code File
if a<b
(Lib ref)
do while
z=x-y
(Lib ref)
Compiler
Object File
11011001
(Lib ref)
00010111
10101011
(Lib ref)
Linker
Executable File
11011001
01000100
00010111
10101011
11111100
Library Files
10111101
11100001
00000011
01000100
10011101
11111100

Reversing

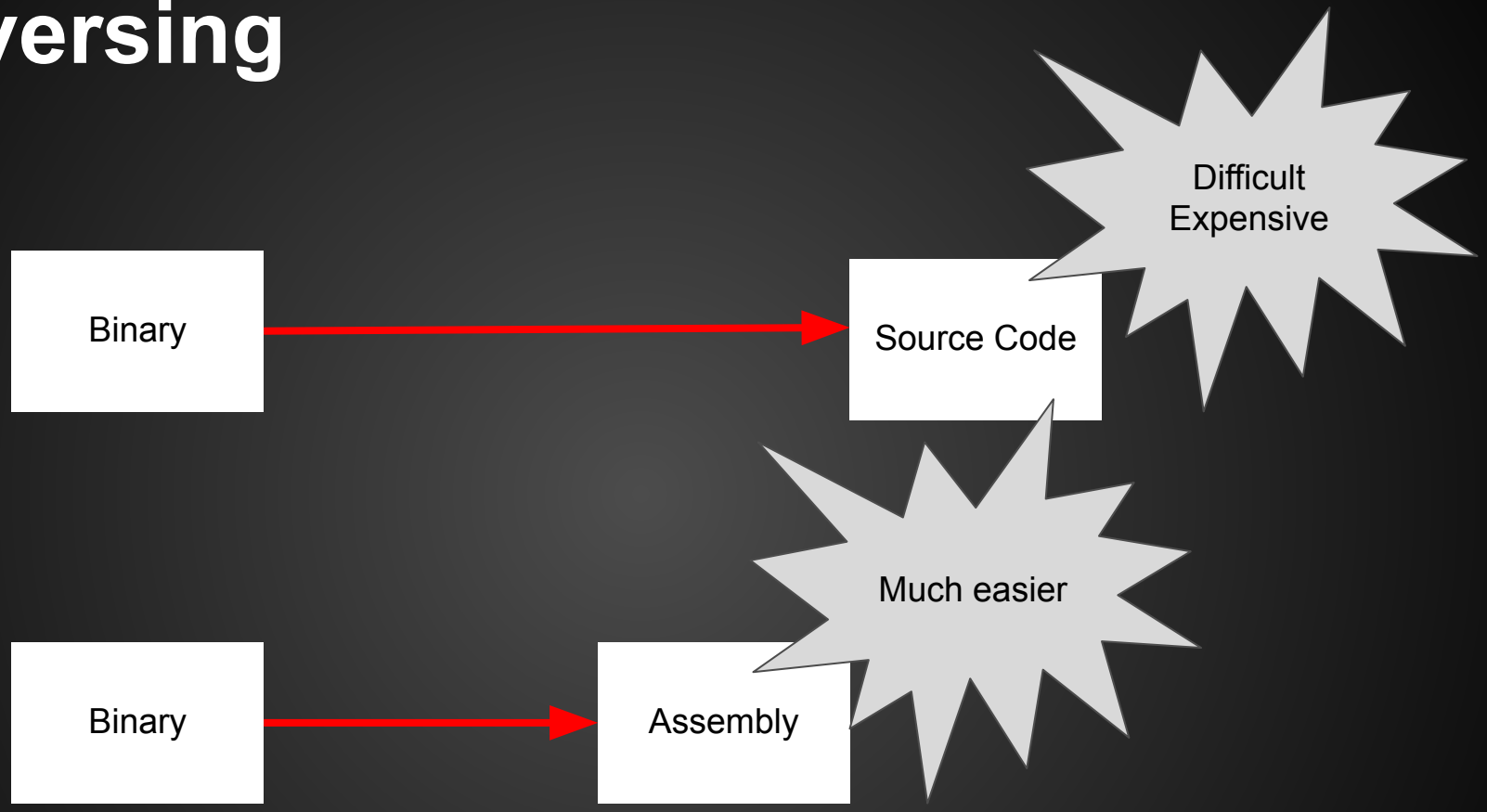
Reversing



Reversing



Reversing


Binary
Source Code
Difficult
Expensive
Binary
Assembly
Much easier

```c
int add5(int num) {
    return num + 5;
}
```

```asm
add5(int):
  push rbp
  mov rbp, rsp
  mov DWORD PTR [rbp-4], edi
  mov eax, DWORD PTR [rbp-4]
  add eax, 5
  pop rbp
  ret
```

Approaches to RE

- Static code analysis
  - Read assembly/source code
  - Identify I/O, important functions, and data structures
- Dynamic code analysis
  - Run code through debugger
  - Observe behavior
  - Notice register values, memory values, etc.

EZ stuff

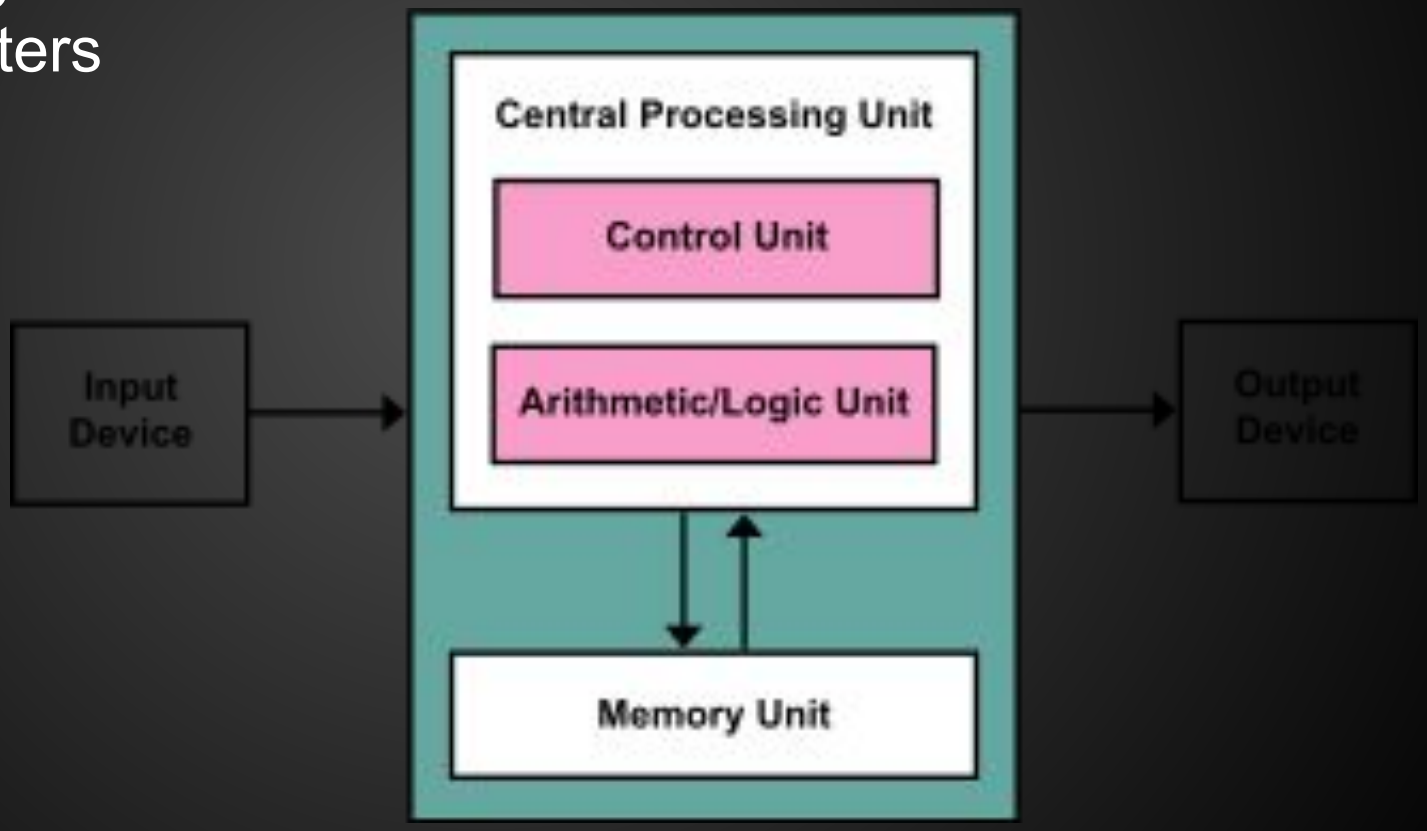
- Before you dive into reversing assembly in IDA, check the easy stuff first
- Run strings on the binary
  - “version”? Source code is often available online
  - “password” Looking for some sort of authentication
  - strings that give away a service (http, dhcp)
  - other interesting strings…
- Imports and Exports
- Run it and see what happens

Strings

- Prints all the ASCII strings in a file
- Quick and easy
- Gives us clues as to what the program roughly does

Von Neumann Architecture

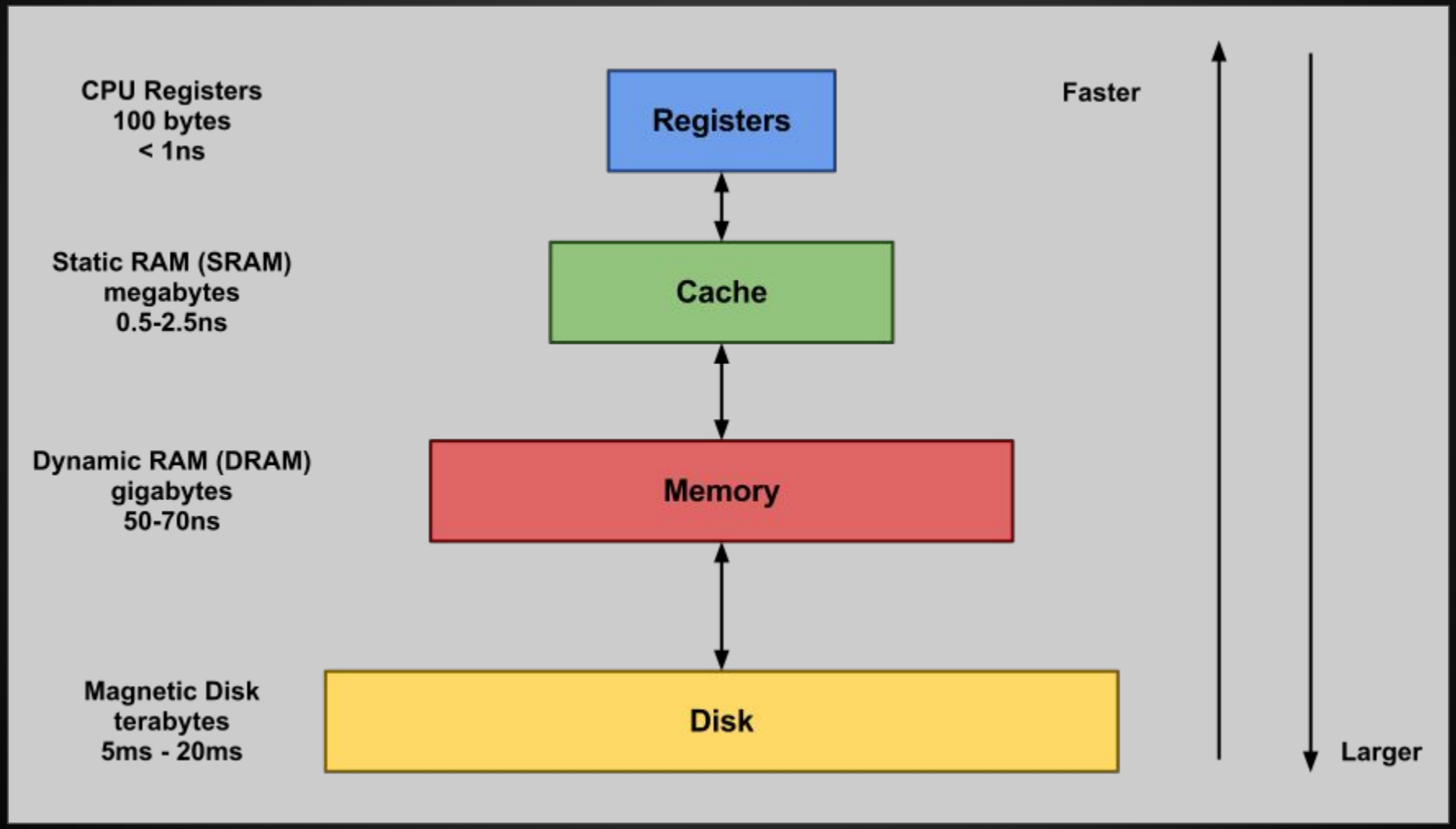
- Memory
  - Registers
  - RAM
- CPU

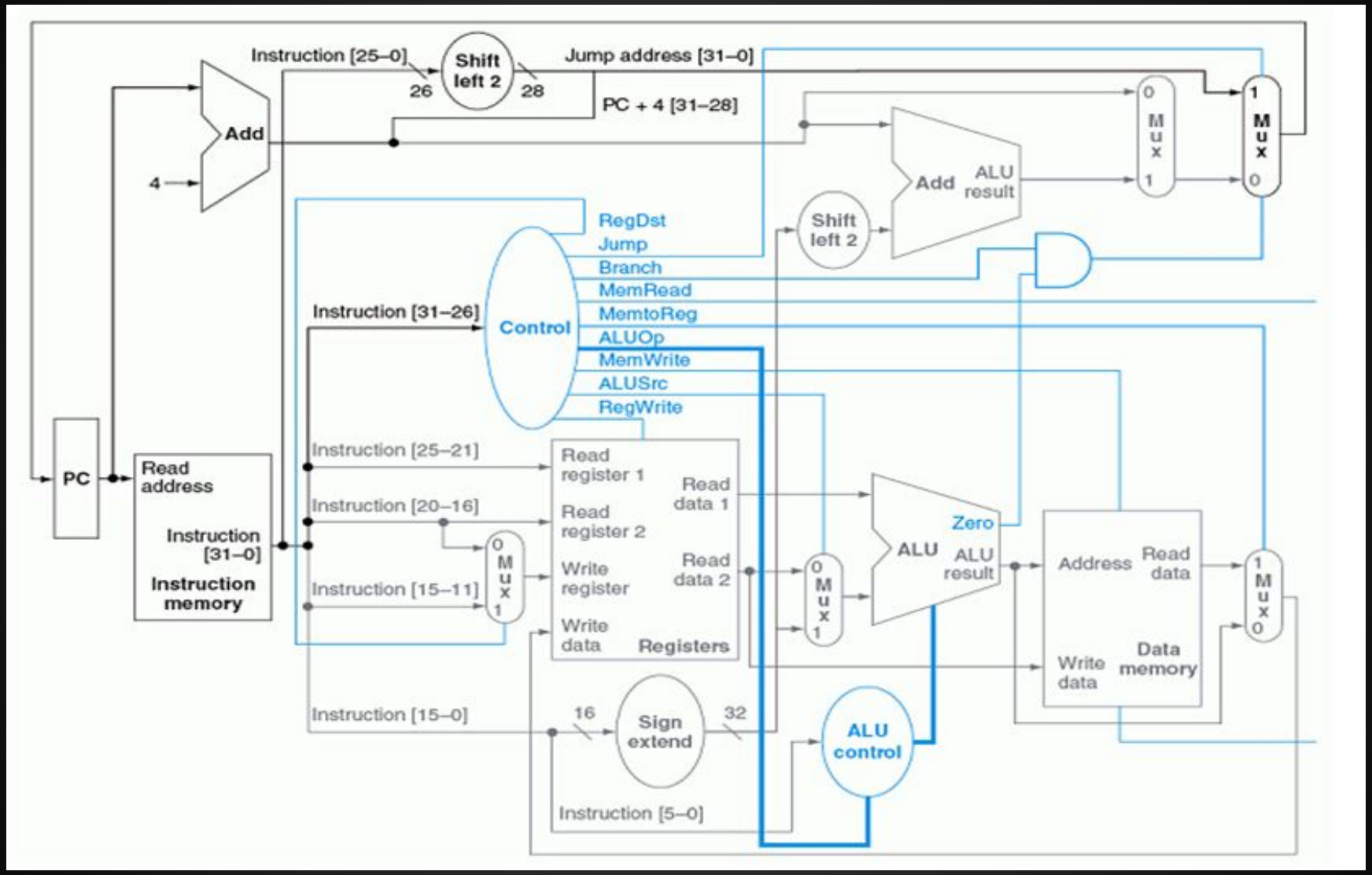


Registers

- Temporary storage
- Very fast to read/write
- (Almost) the only way to compute on data

Memory Levels



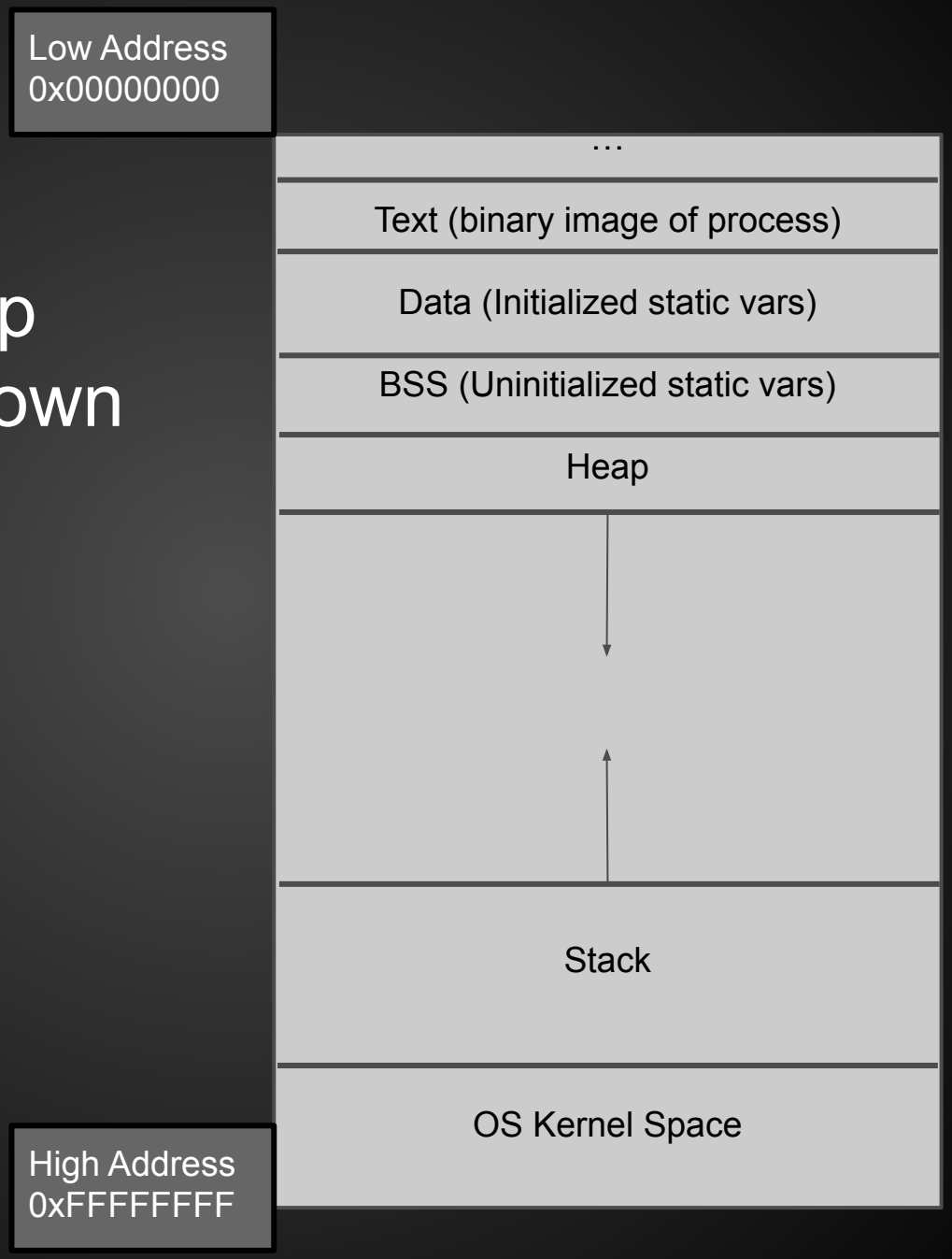

Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC + 4 [31–28]
Add
4
Add
ALU result
Shift left 2
M u x
M u x
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
Instruction [15–0]
16
Sign extend
32
ALU control
Instruction [5–0]

Instructions

- Depends on architecture instruction set
- Assembly that manipulates data
  - Move from memory to register
  - Add, subtract, multiply, divide
  - Bit shifts
  - Store data from register to memory
  - Jump to different memory address
- Syntax:
  - Operation destination_register, source_register(s)
  - ADD r0, r0, r1 // ARM
  - ADD EAX, EBX // x86

Memory

- Stack grows up
- Heap grows down



https://microcorruption.com

Disassembly

```
4436:  fa23          jnz     #0x442c <__do_clear_bss+0x8>
4438 <main>
4438:  3150 9cff     add     #0xff9c, sp
443c:  3f40 a844     mov     #0x44a8 "Enter the password to continue", r15
4440:  b012 5845     call    #0x4558 <puts>
4444:  0f41          mov     sp, r15
4446:  b012 7a44     call    #0x447a <get_password>
444a:  0f41          mov     sp, r15
444c:  b012 8444     call    #0x4484 <check_password>
4450:  0f93          tst     r15
4452:  0520          jnz     #0x445e <main+0x26>
4454:  3f40 c744     mov     #0x44c7 "Invalid password; try again.", r15
4458:  b012 5845     call    #0x4558 <puts>
445c:  063c          jmp     #0x446a <main+0x32>
445e:  3f40 e444     mov     #0x44e4 "Access Granted!", r15
4462:  b012 5845     call    #0x4558 <puts>
4466:  b012 9c44     call    #0x449c <unlock_door>
446a:  0f43          clr     r15
446c:  3150 6400     add     #0x64, sp
4470 <__stop_progExec__>
```

Live Memory Dump

```
0000:   0000 4400 0000 0000 0000 0000 0000 0000   ..D.............
0010:   *
4400:   3140 0044 1542 5c01 75f3 35d0 085a 3f40   1@.D.B\.u.5..Z?@
4410:   0000 0f93 0724 8245 5c01 2f83 9f4f 8645   .....$.E\./..O.E
4420:   0024 f923 3f40 0000 0f93 0624 8245 5c01   .$.#?@.....$.E\.
4430:   1f83 cf43 0024 fa23 3150 9cff 3f40 a844   ...C.$.#1P..?@.D
4440:   b012 5845 0f41 b012 7a44 0f41 b012 8444   ..XE.A..zD.A...D
4450:   0f93 0520 3f40 c744 b012 5845 063c 3f40   ... ?@.D..XE.<?@
4460:   e444 b012 5845 b012 9c44 0f43 3150 6400   .D..XE...D.C1Pd.
4470:   32d0 f000 fd3f 3040 8445 3e40 6400 b012   2....?0@.E>@d...
4480:   4845 3041 6e4f 1f53 1c53 0e93 fb23 3c90   HE0AnO.S.S...#<.
4490:   0900 0224 0f43 3041 1f43 3041 3012 7f00   ...$.C0A.C0A0.▫.
44a0:   b012 f444 2153 3041 456e 7465 7220 7468   ...D!S0AEnter th
44b0:   6520 7061 7373 776f 7264 2074 6f20 636f   e password to co
```

pc

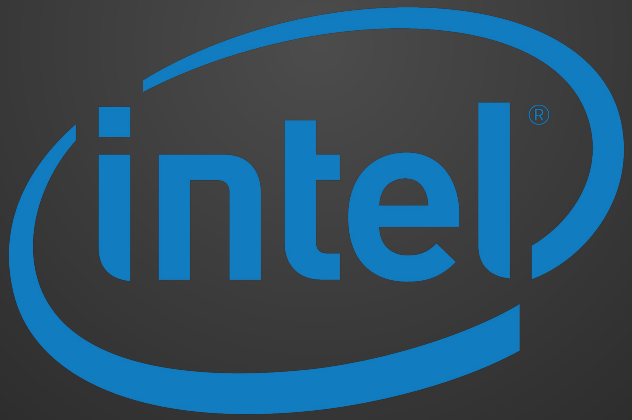
Register State

```
pc  443c  sp  439c  sr  0001  cg  0000
r04 0000  r05 5a08  r06 0000  r07 0000
r08 0000  r09 0000  r10 0000  r11 0000
r12 0000  r13 0000  r14 0000  r15 0000
```

Current Instruction

```
3f40 a844
mov #0x44a8, r15
```

Introduction to IA-32 (x86)



x86 Architectural Features

- “Intel Architecture, 32-bit”
- Sometimes called i386, x86
- 32 bit version of the x86 architecture

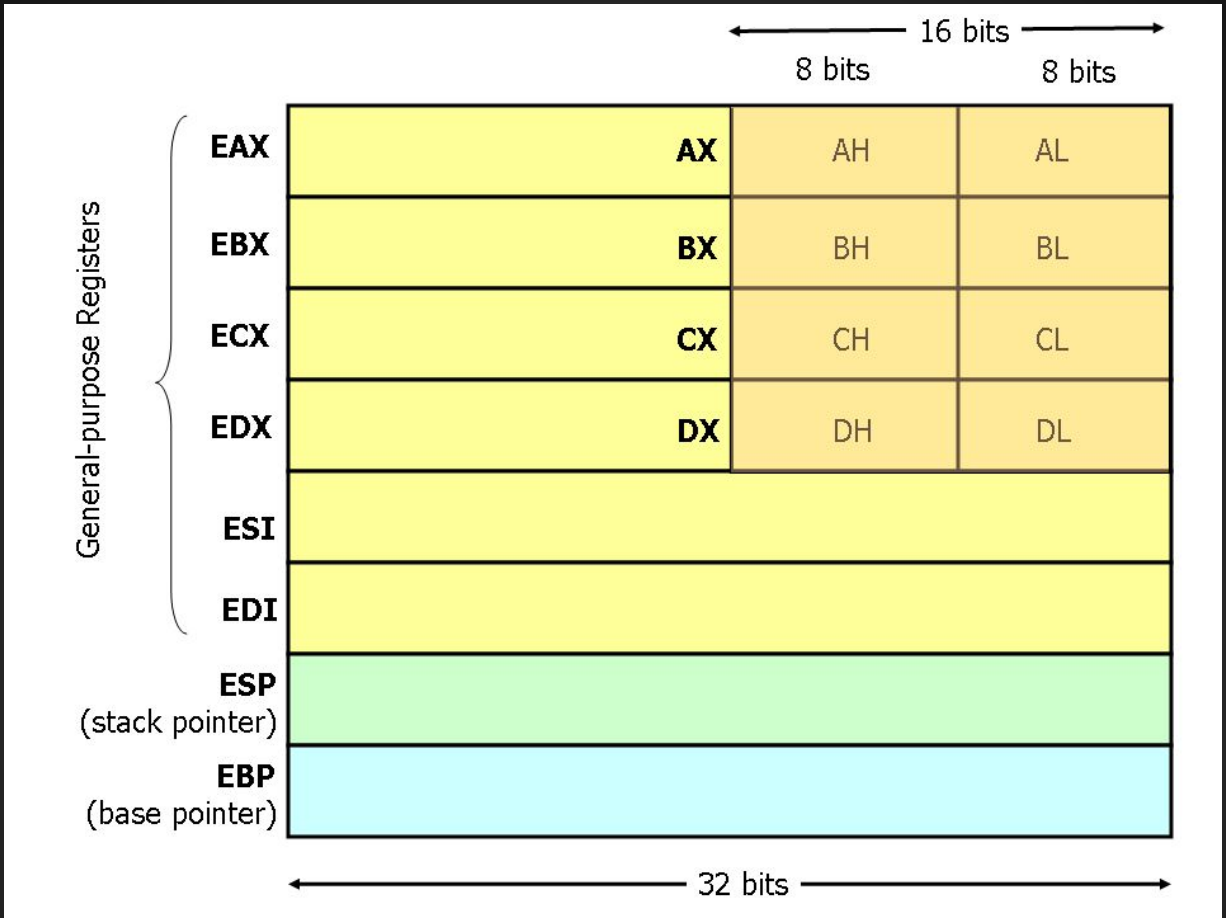
x86 Outline

- Registers
- Syntax
- Common Instructions

Registers

Registers

- Memory that the processor can access much faster than RAM
- There are a lot of them, but we'll focus on a few of the more important ones
- EAX, EBX, ECX, EDX, ESI, EDI can be used as general storage registers
- "E" stands for extended (32 bits vs 16 bits)
- RAX, RBX, etc. for 64-bit registers


16 bits
8 bits
8 bits
General-purpose Registers
EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
EDI
ESP
(stack pointer)
EBP
(base pointer)
32 bits

Registers

- Conventional use - not so much in practice
- EAX
  - Accumulator
  - **Return value**
- EBX
  - Base index (arrays)
- ECX
  - Counter (loops)
- EDX
  - Data

Registers

- ESI
  - Source index (memory copying operations)
- EDI
  - Destination index (memory copying operations)
- EBP
  - Base pointer (base of the current stack frame)
- ESP
  - Stack pointer (address of highest element on stack)

Registers

- EIP
  - Instruction Pointer (pointer to next instruction)
- EFLAGS
  - Relevant flags are carry flag (CF), zero flag (ZF), Sign flag (SF), and overflow flag (OF)
  - Used for conditional statements
- You can’t directly move values into these registers

Instruction Syntax

Intel Syntax

- Intel
  - Operation Destination, Source
  - Parameter size derived from name of register (rax, eax, ax, al/ah)
  - No prefixes on immediates or registers
  - mov eax, 0x05

AT&T Syntax

- AT&T (GAS)
  - Operation Source, Destination
  - Suffix for size of operands: q,l,w,b
  - Immediates prefixed with $ and registers prefixed with %
  - movl $0x05, %eax

Common Instructions

Instructions

- We will be using Intel Syntax
  - destination, source
- Like registers, there are a lot of x86 instructions.
  - We will focus on some of the more common ones
- When starting RE, don’t focus on memorizing instructions.
  - Look them up as needed

Instructions

- MOV
  - mov eax, 1 // eax = 1
- ADD, SUB, etc
  - ADD eax, 4 // eax += 4
  - SUB eax, 8 // eax -= 8
- AND, OR, NOT, XOR
  - xor eax, ebx // eax = eax ^ ebx
- SAL, SAR, SHL, SHR
  - shl edx, 4 // edx = edx * 16

Instructions

- LEA
  - “Load Effective Address”
  - Often used to load an absolute address from a relative offset in a general purpose register
  - Good Stackoverflow descriptions of LEA
- PUSH, POP
  - Stack Manipulation
- CALL, RET
- Stack set up and teardown per C calling convention

Instructions

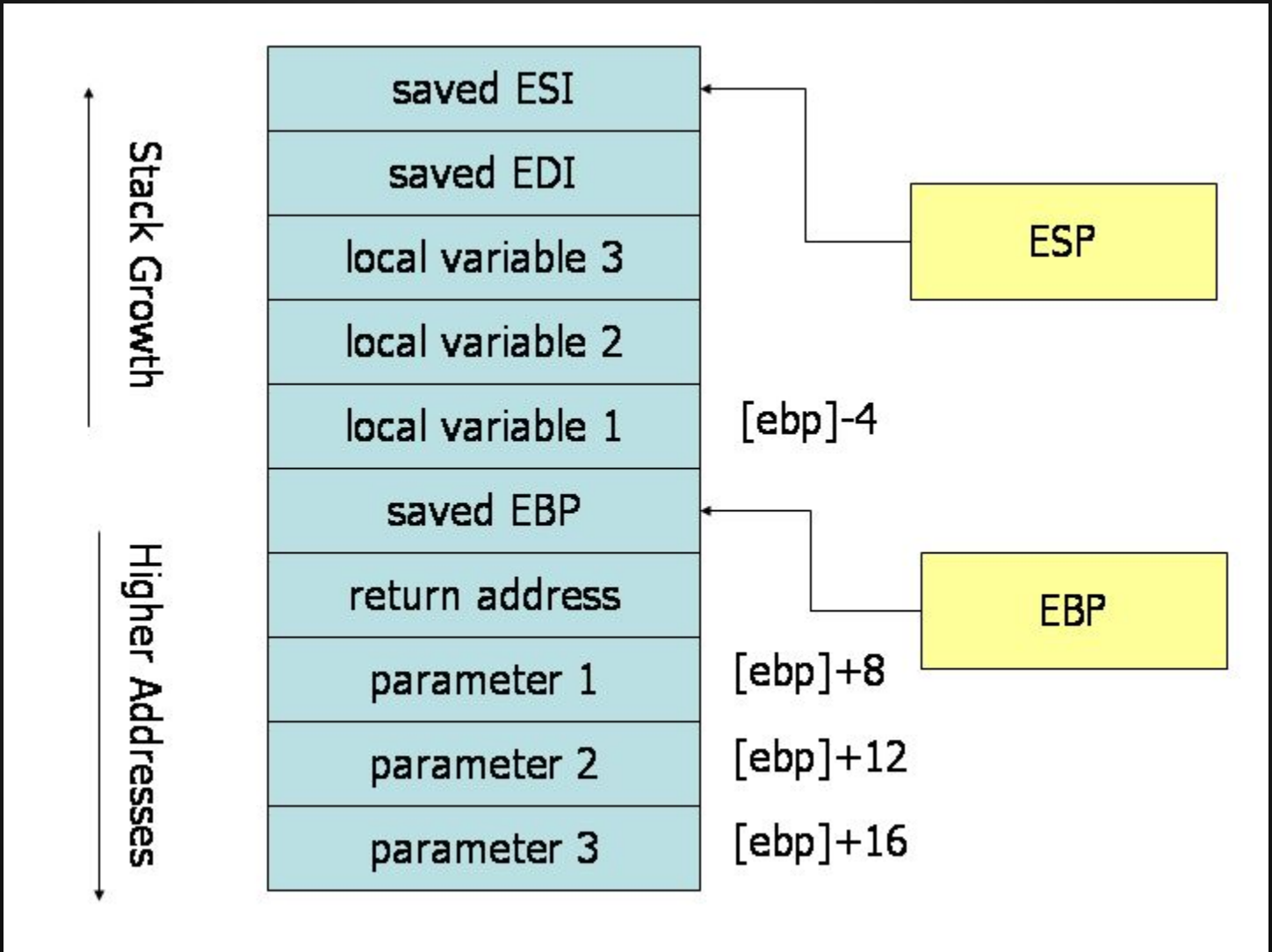
- CMP
  - Subtracts operands but discards result
  - Sets flags
- TEST
  - ANDs operands but discards result
  - Sets flags
- JMP/Jxx
  - JNE, JAE, etc

Memory addressing

- mov eax, [ebx+4*ecx]
  - eax = *(ebx + 4*ecx)
  - [ ] dereferences an address

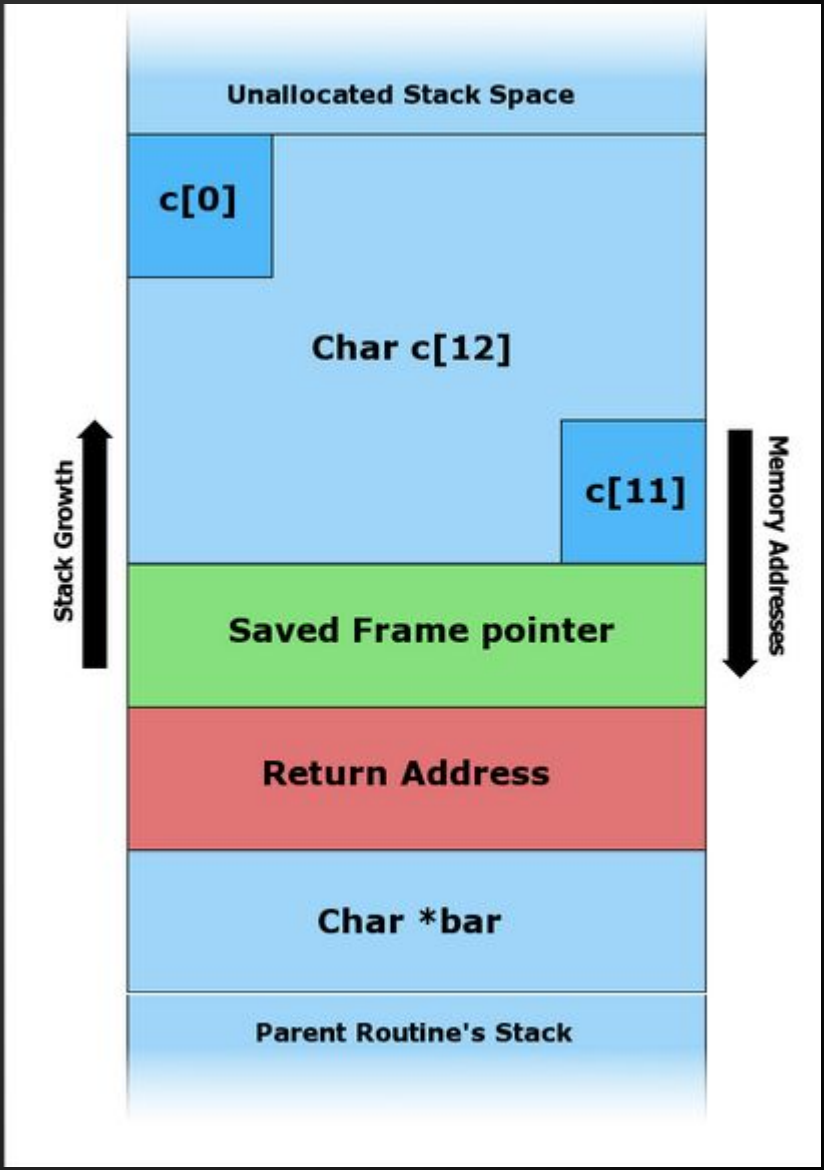
The Stack and C Calling Convention

int func(param1, param2, param3) { int var1, var2, var3; }



C Call Stack

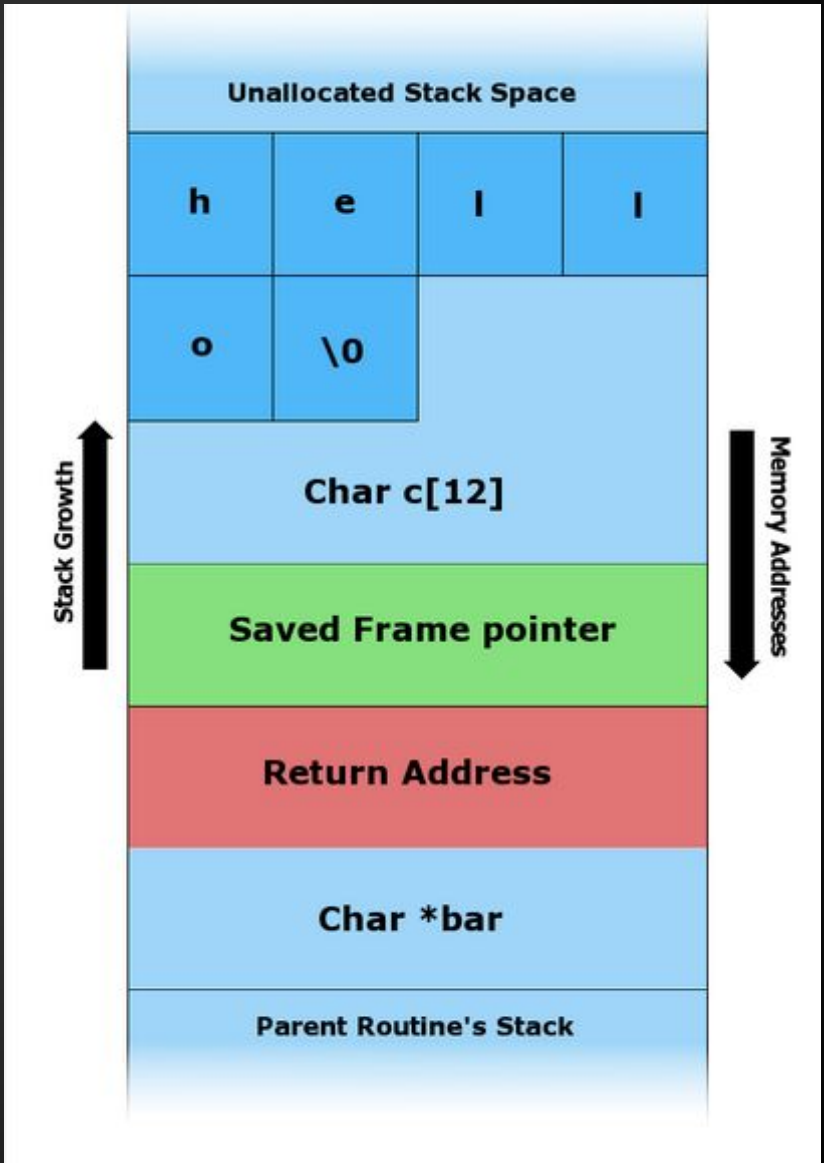
```c
#include <string.h>

void foo (char *bar)
{
   char  c[12];

   strcpy(c, bar);  // no bounds checking
}

int main (int argc, char **argv)
{
   foo(argv[1]);
}
```



C Call Stack

Input: “Hello”

```c
#include <string.h>

void foo (char *bar)
{
   char  c[12];

   strcpy(c, bar);  // no bounds checking
}

int main (int argc, char **argv)
{
   foo(argv[1]);
}
```



C Call Stack

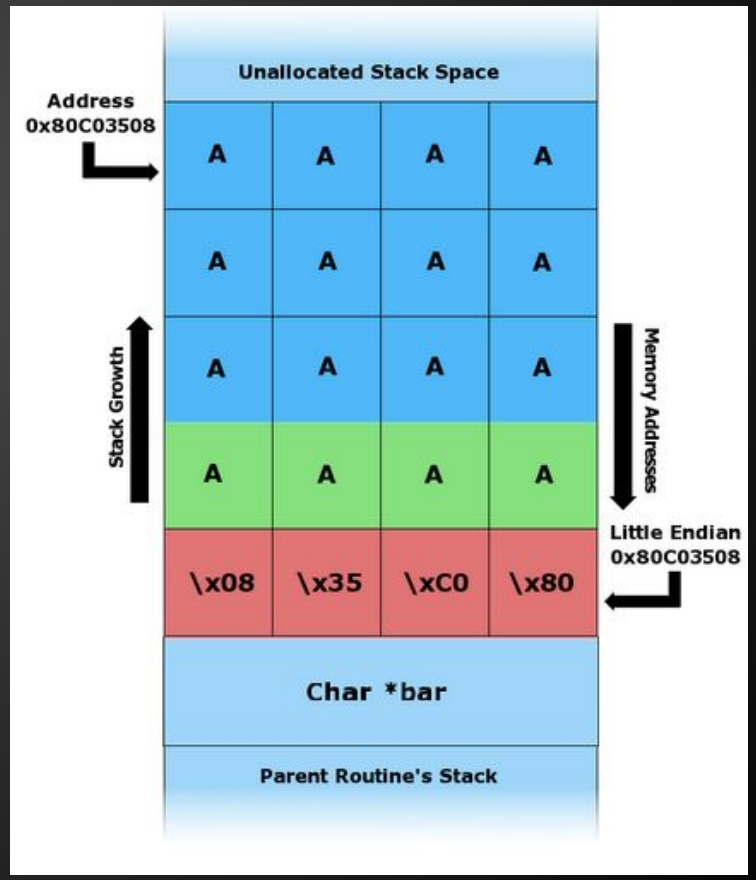
Input: “AAAAAAAAAAAAAAAA\x08\x35\xC0\x80”

```c
#include <string.h>

void foo (char *bar)
{
   char  c[12];

   strcpy(c, bar);  // no bounds checking
}

int main (int argc, char **argv)
{
   foo(argv[1]);
}
```

A char = 1 Byte

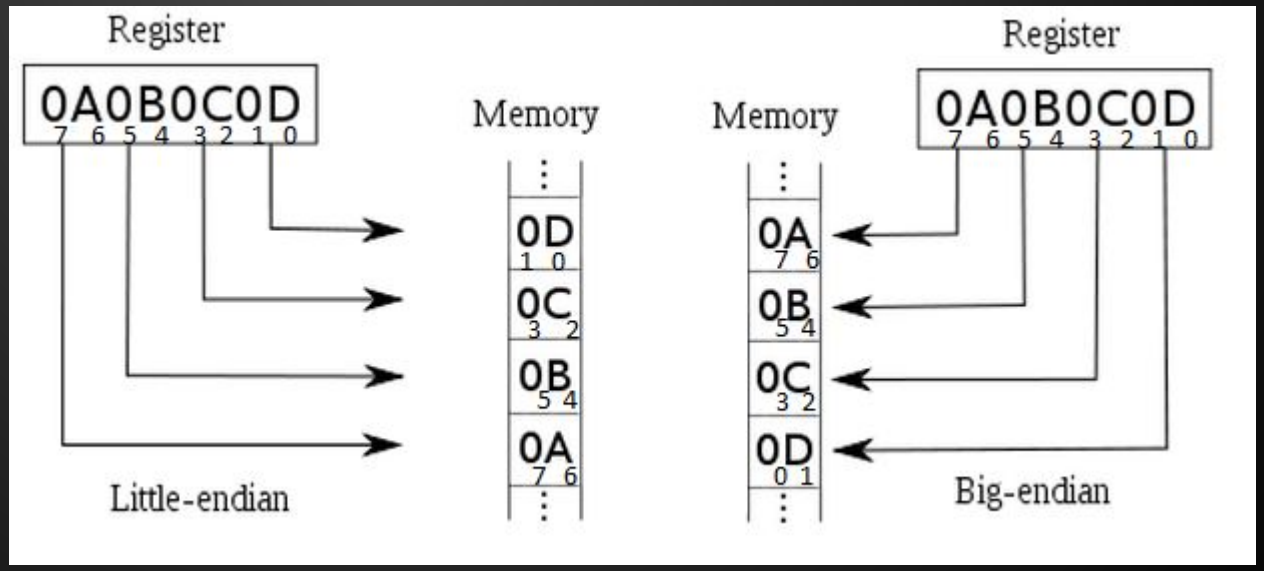
Hex (“\x0”) = 4 Bits = ½ Byte

2 Hex (“\x00”) = 1 Byte



Endianness

- Intel x86 uses little endian
  - “Little end” (least significant byte) goes into lower memory address
- Bytes are in reverse order in little endian
  - Address 0x0A0B0C0D looks like 0x0D0C0B0A in memory.



Some IDA Commands

- n // rename function, variable, register
- h // toggle between hex and decimal
- alt + t // search for text
- spacebar // toggle between views
- x // cross references to function, variable
- ctrl - left click // highlight multiple nodes
- right click -> group nodes // groups nodes
- y // change type (int, char) of variable
- alt-Q // list of structures

GDB commands

- gdb <program> // runs program in gdb
- set disassembly-flavor intel
- b <address/function> // sets breakpoint
- r // run (restart) program
- p $<register> // prints register value
- n // next instruction
- s // step into function
- si // step 1 assembly instruction
- c // continue executing
- x <address> // examine memory

GDB stuff

- "set follow-fork-mode child"
- gdb <program> // opens and loads program into gdb
  - file <program> // loads program into gdb
  - r <params> // runs the loaded program with params as argument
  - r < <file.txt> // runs the program with contents of file.txt as parameter
  - \xCC // Debugger trap, when executed in gdb, program should exit with SIGTRAP. Use to test if you get code execution.
  - b <function name or line number> // set breakpoint

Some radare2 commands

- r2 <program> // runs in read mode
- r2 -A <program> // run and analyze funcs
- r2 -Aw <program> // analyze and write-mode

- s <address> // set selector to address
- pd <size> // print disassembly at selector
- pdr // print disassembled function (if -A)
- aa // analyze functions and bbs
- ag $$ > a.dot // creates basic block graph
- agc $$ > a.dot // creates call graph
- $$ = at this location

Some radare2 debugger commands

- db @ <address/function> // sets breakpoint
- do // reopen program in debugger
- dr // prints all register values
- dr?<register> // prints register value
- dr eax=0 // set register eax=0
- ds // step 1 assembly instruction
- dc // continue executing

radare2 commands

- Cheatsheet:

https://github.com/pwntester/cheatsheets/blob/master/radare2.md

Exercises

- https://github.com/sigpwny/RE_Labs
- Combination
- CD_Key
- Mr.E

Combination

This executable is like a lock. There are multiple stages that need to be unlocked one at a time. Dynamic analysis is a must!

CD_key

You downloaded Winrar, but it asks for a CD key before it will install. Ha. It should be fairly easy to crack the CD key validator on this. You might even go so far as to create a keygen...

Mr.E

Your name is Ben Bitdiddle. You are an ECE student at UIUC. Your GPA is 1.5 due to constantly ing your group projects with your horrible suggestions. You need to raise your GPA to at least a 2.0 by the end of the semester to graduate, but you don't know how. Your classmate Alyssa P. Hacker feels bad for you. She hacks the school network and brings you a flash drive containing a single file, "Mr.E", telling you that it is your ticket to graduation. What does this file do? How can this help you? Do you really trust her? Only one way to find out…

So you want to learn more..

- Books
  - Reversing: Secrets of Reverse Engineering
  - The IDA Pro Book
  - Practical Reverse Engineering
  - Practical Malware Analysis
- OpenSecurityTraining
  - Intro classes on x86, ARM, Reverse Engineering and more!
- CTF challenges

Troubleshooting the Challenges

- Turn off ASLR:

  as root: echo 0 > /proc/sys/kernel/randomize_va_space

  This only persists until next reboot

- If you get a Makefile compile error about missing libraries (probably if you are using a 64-bit machine) install g++-multilib

  sudo apt-get install g++-multilib