

SIGPwny
Intro to
Reverse Engineering

Announcements

- UIUCTF Challenge Idea Submissions
(Closing at Midnight) -
<https://goo.gl/forms/fmp7eWsl6MEDqGyq1>
- Manticore Focus Group

News of the Week

- [Cloudflare](#)
- [AWS Outage](#)
- [Google Discloses Microsoft Vuln \(again \(again\)\)](#)
- [Bash Bunny](#)
- [IOT Teddy Bear Data Leaked](#)
- [Slack Bug Fixed](#)
- [Push for ISP-level Piracy Mitigations](#)
- [CBP Verifying CS Degrees](#)

Reverse Engineering

What is RE

Reverse engineering is the process of understanding a program's functionality and behavior through 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”

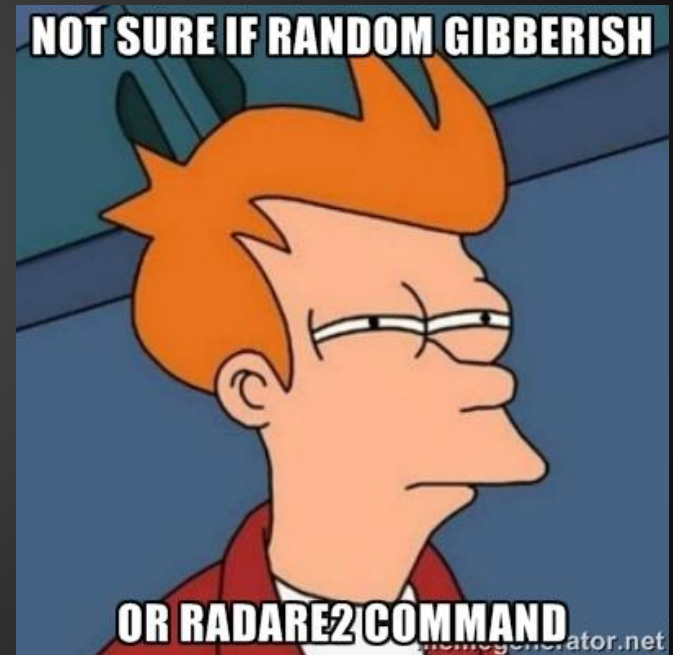
What is RE used for

- 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

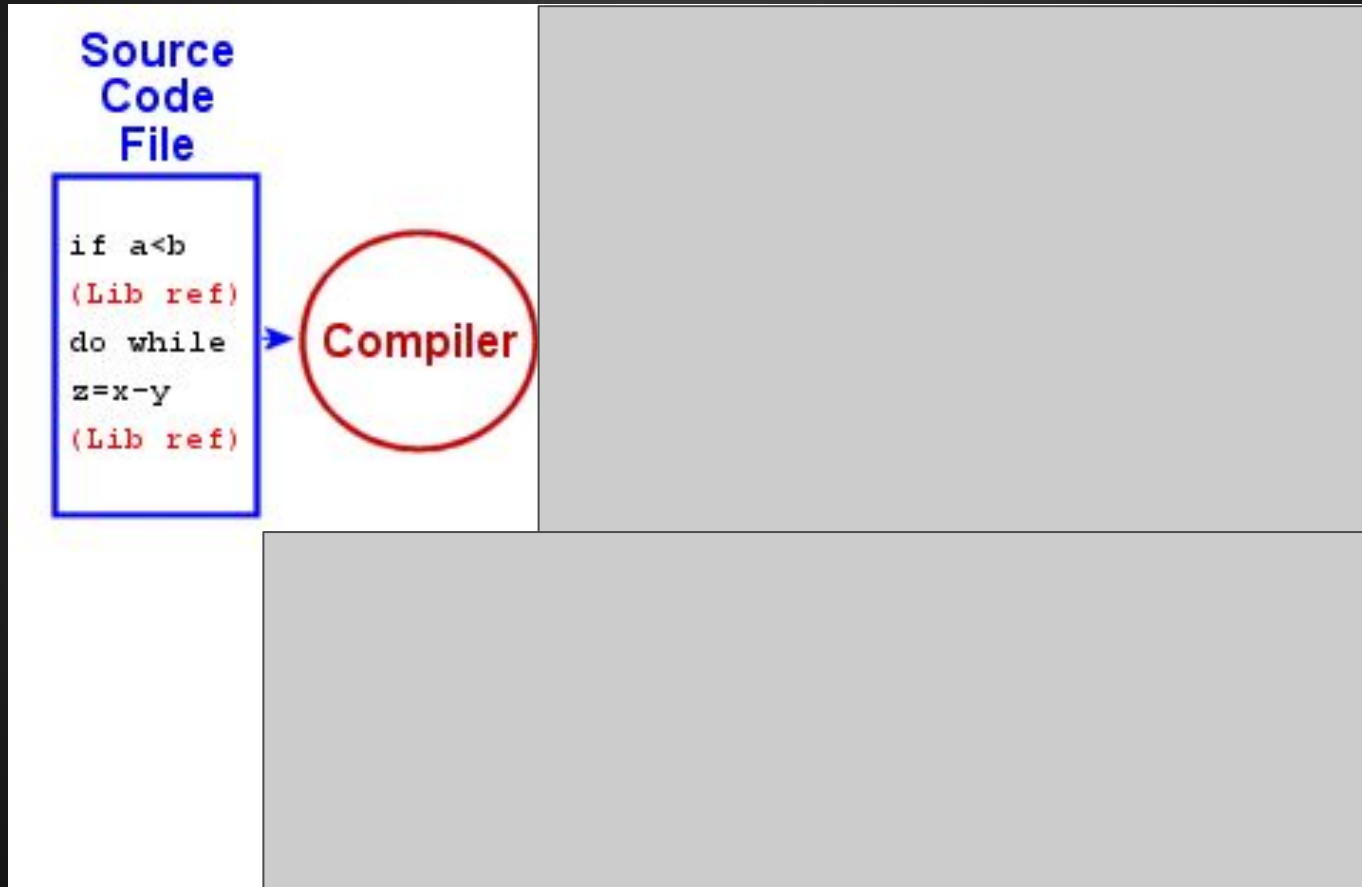


Compiling

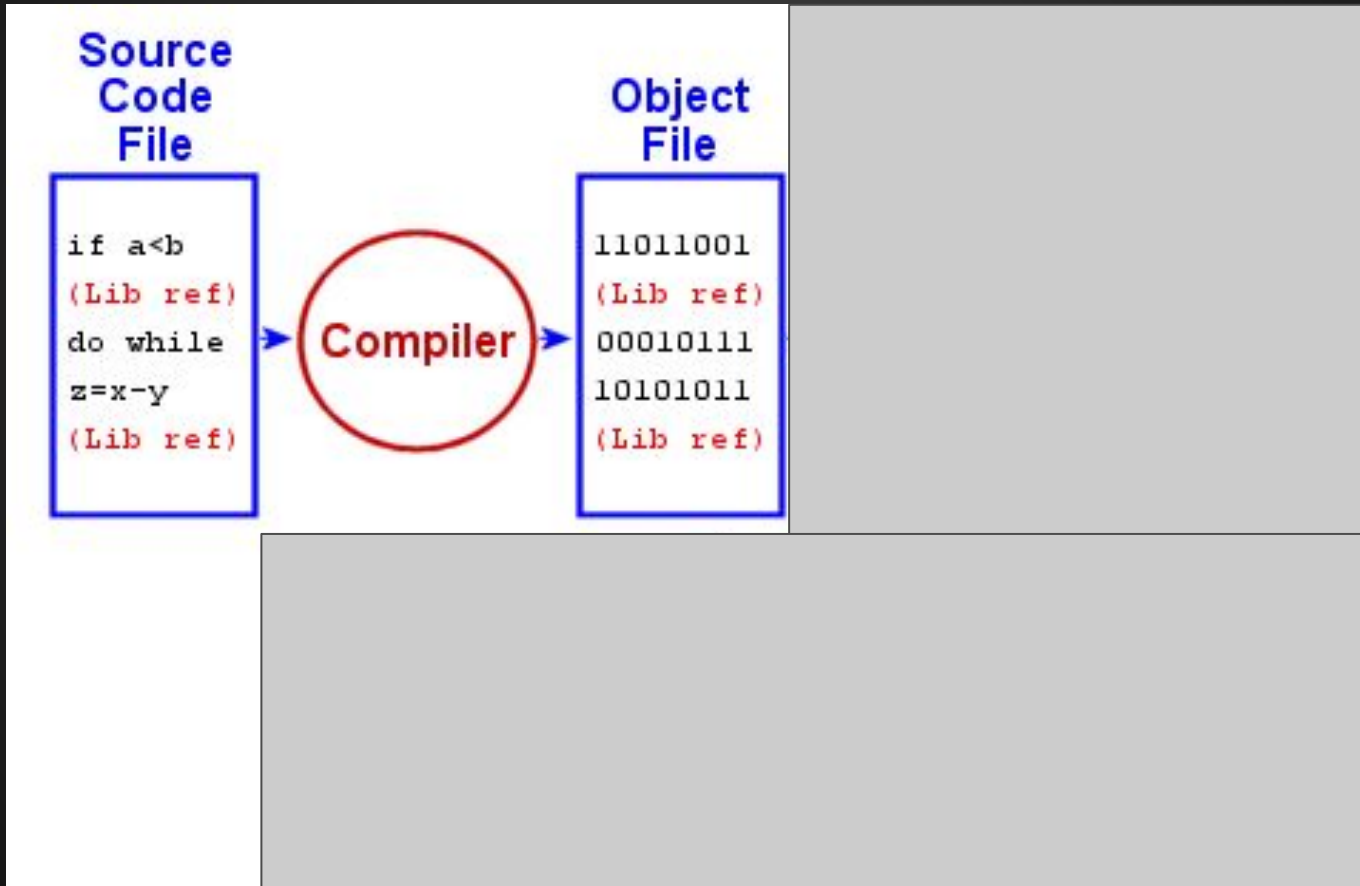
Source
Code
File

```
if a<b  
(Lib ref)  
do while  
z=x-y  
(Lib ref)
```

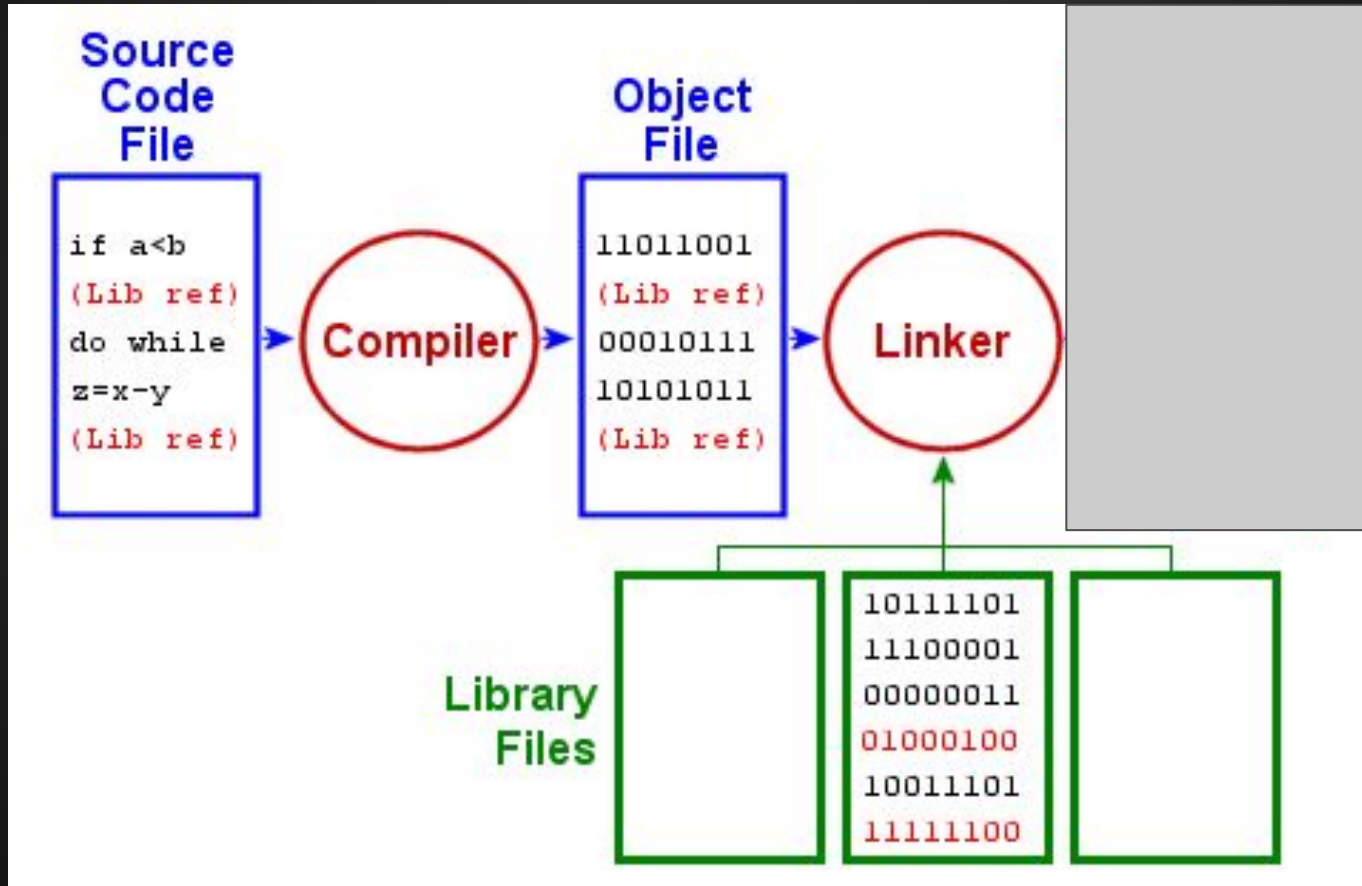
Compiling



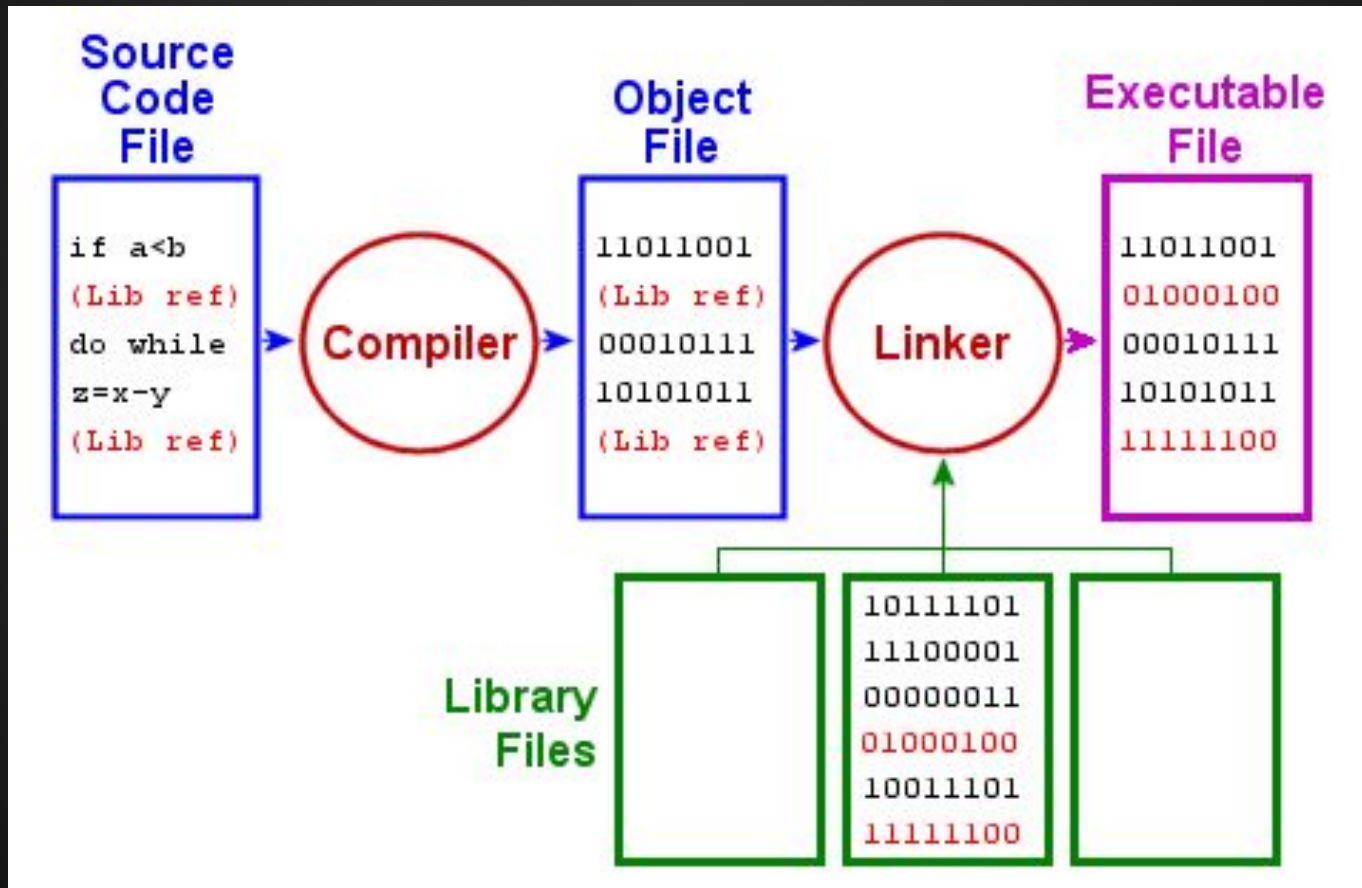
Compiling



Compiling



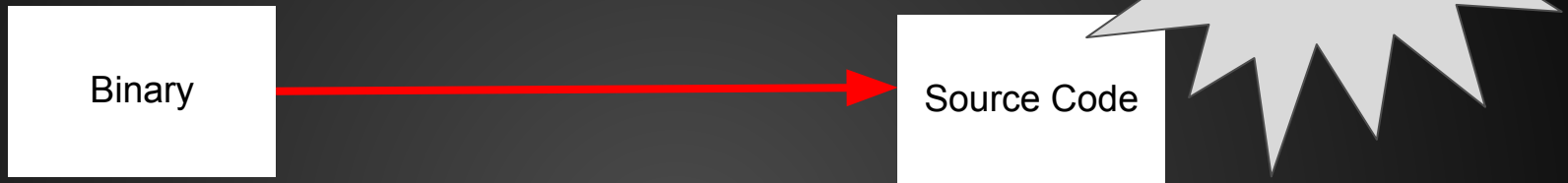
Compiling



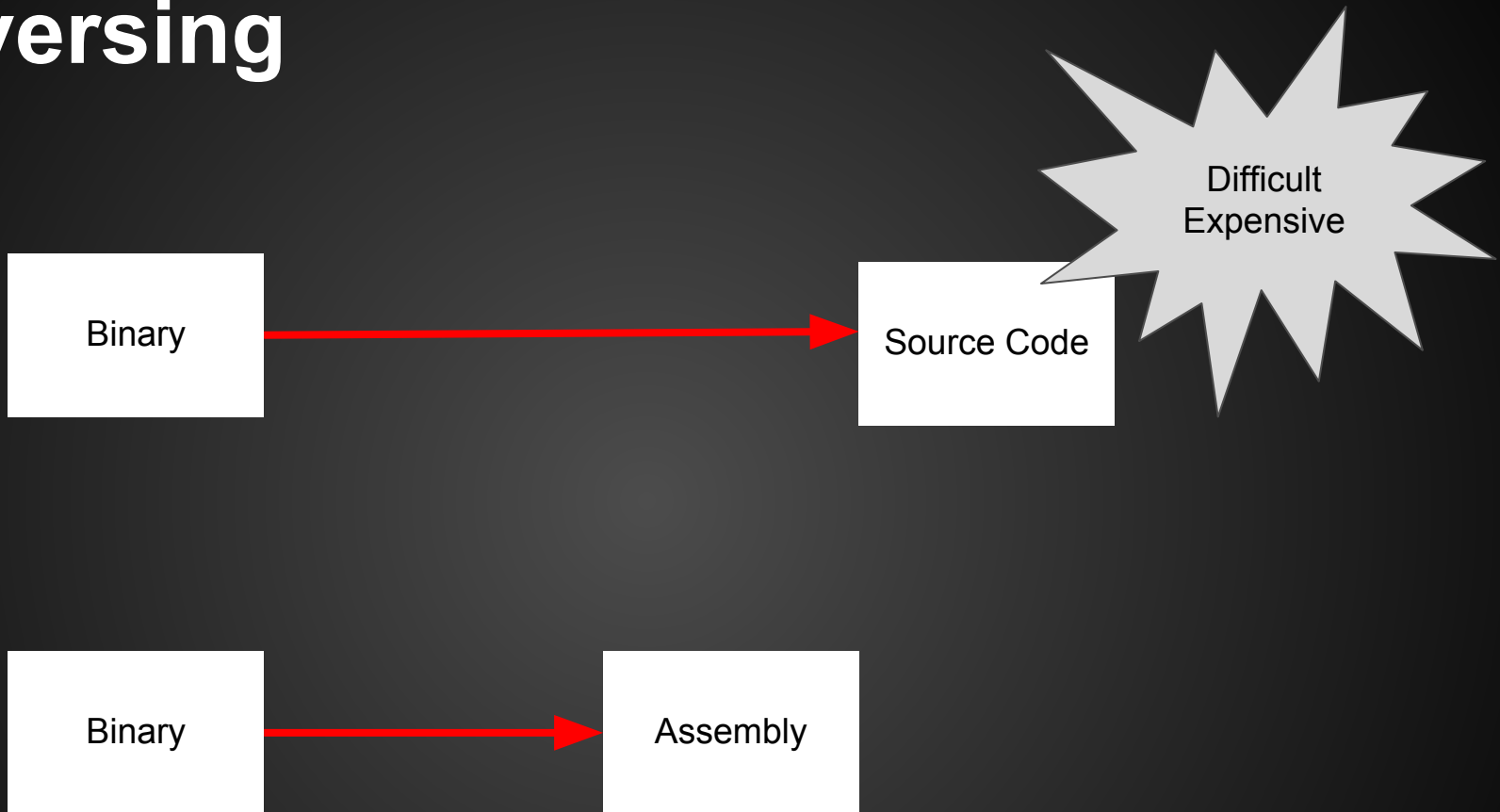
Reversing



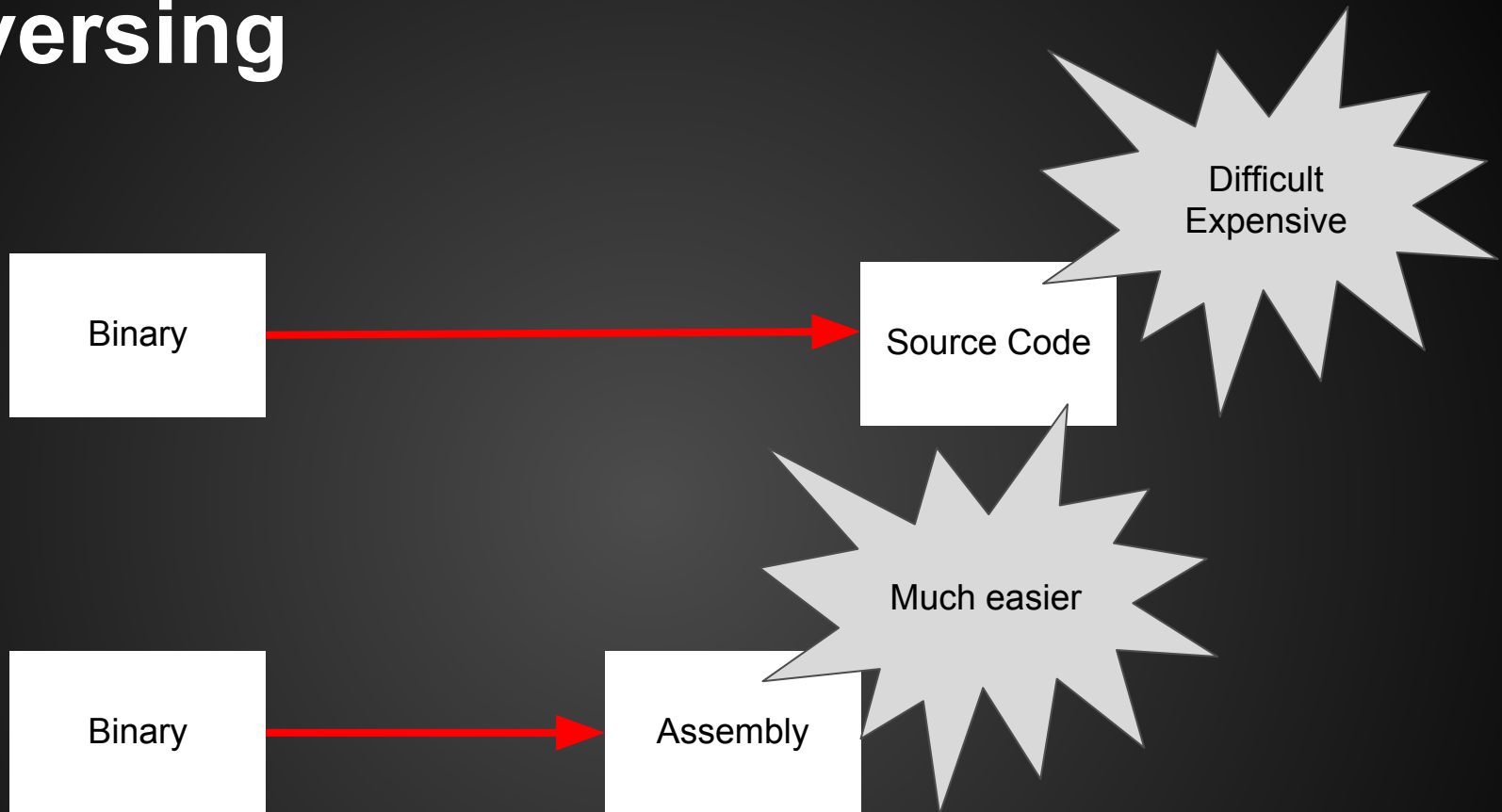
Reversing



Reversing



Reversing



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

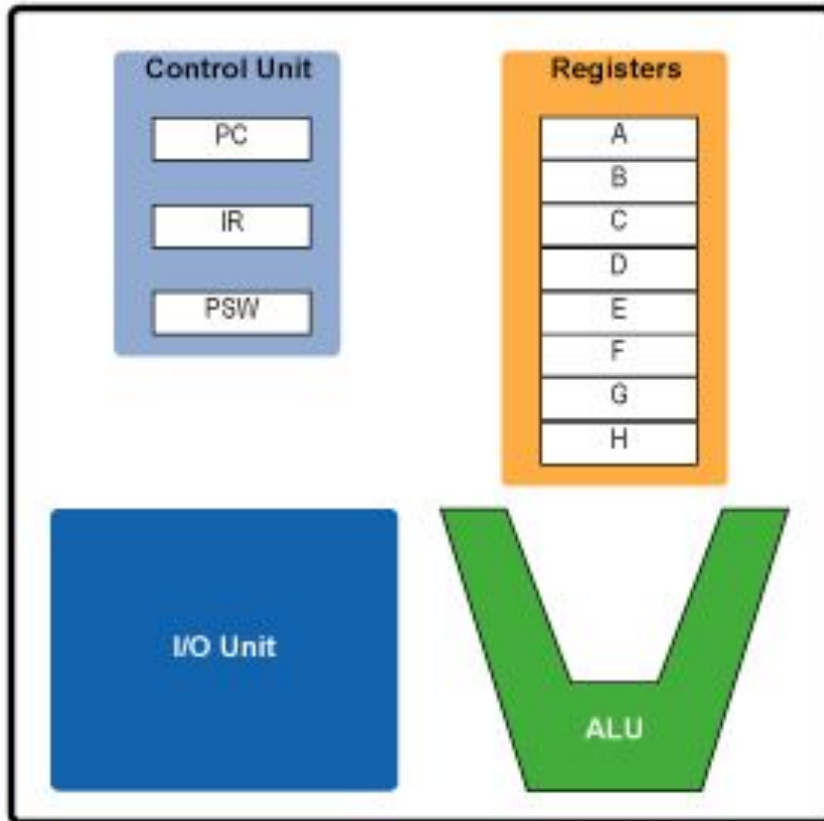
Some Basics

- Registers
- Instructions
- Memory

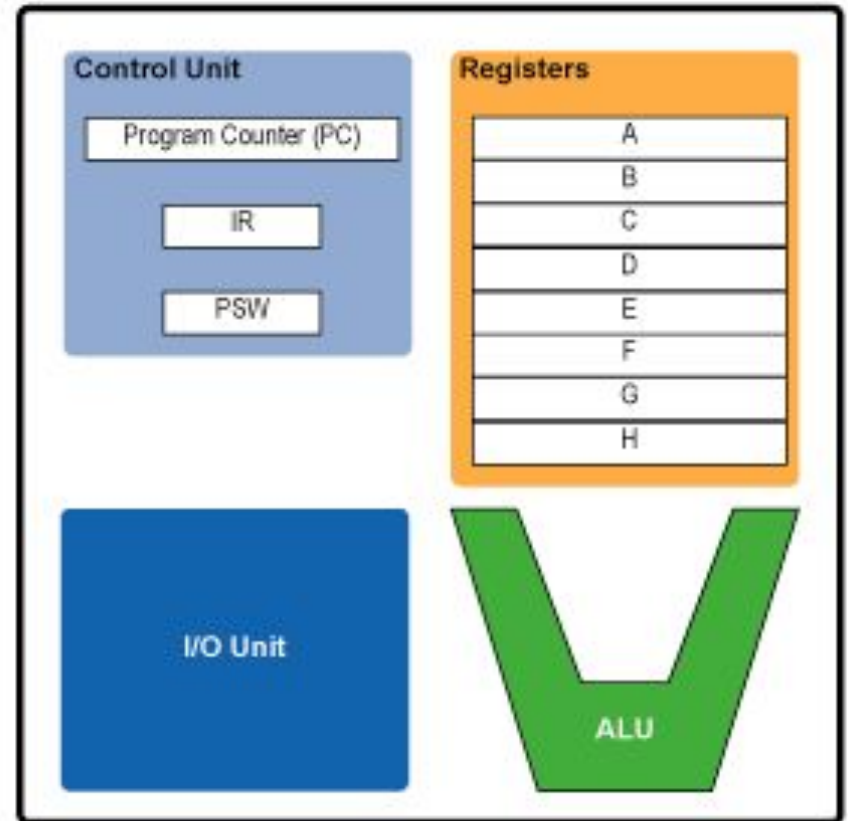
Registers

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

Simplified Microarchitecture

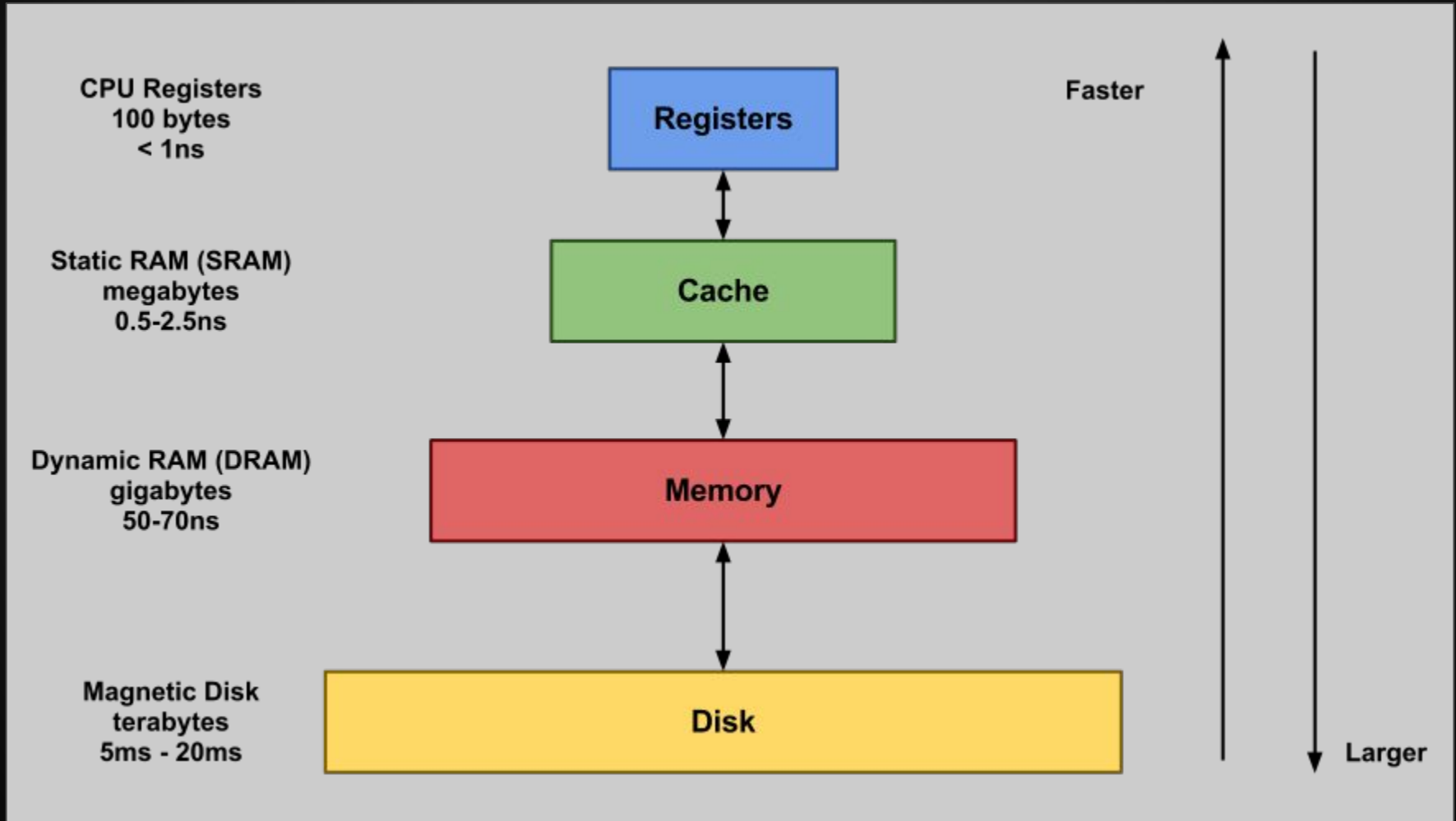


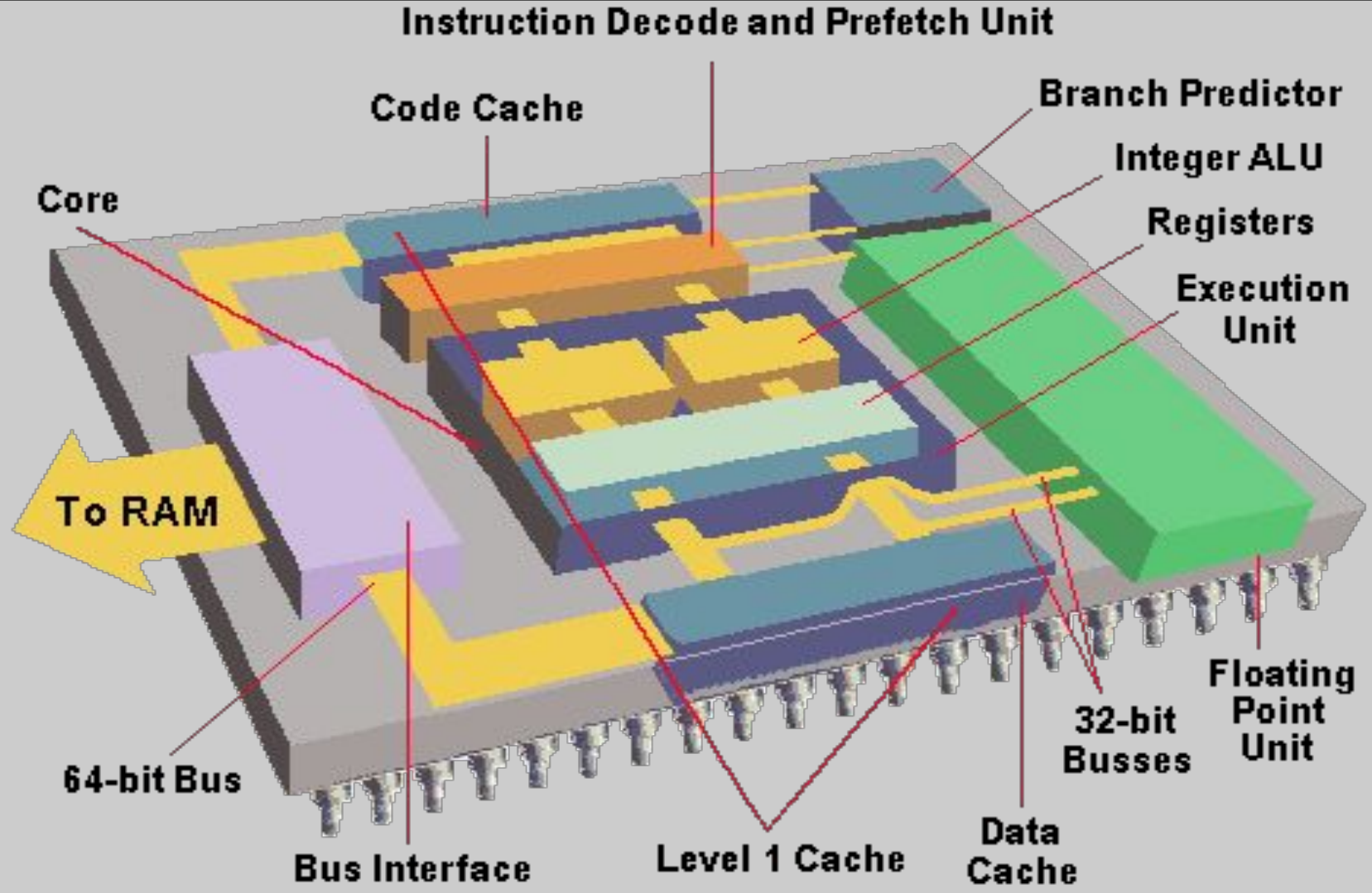
32-bit



64-bit

Memory Levels



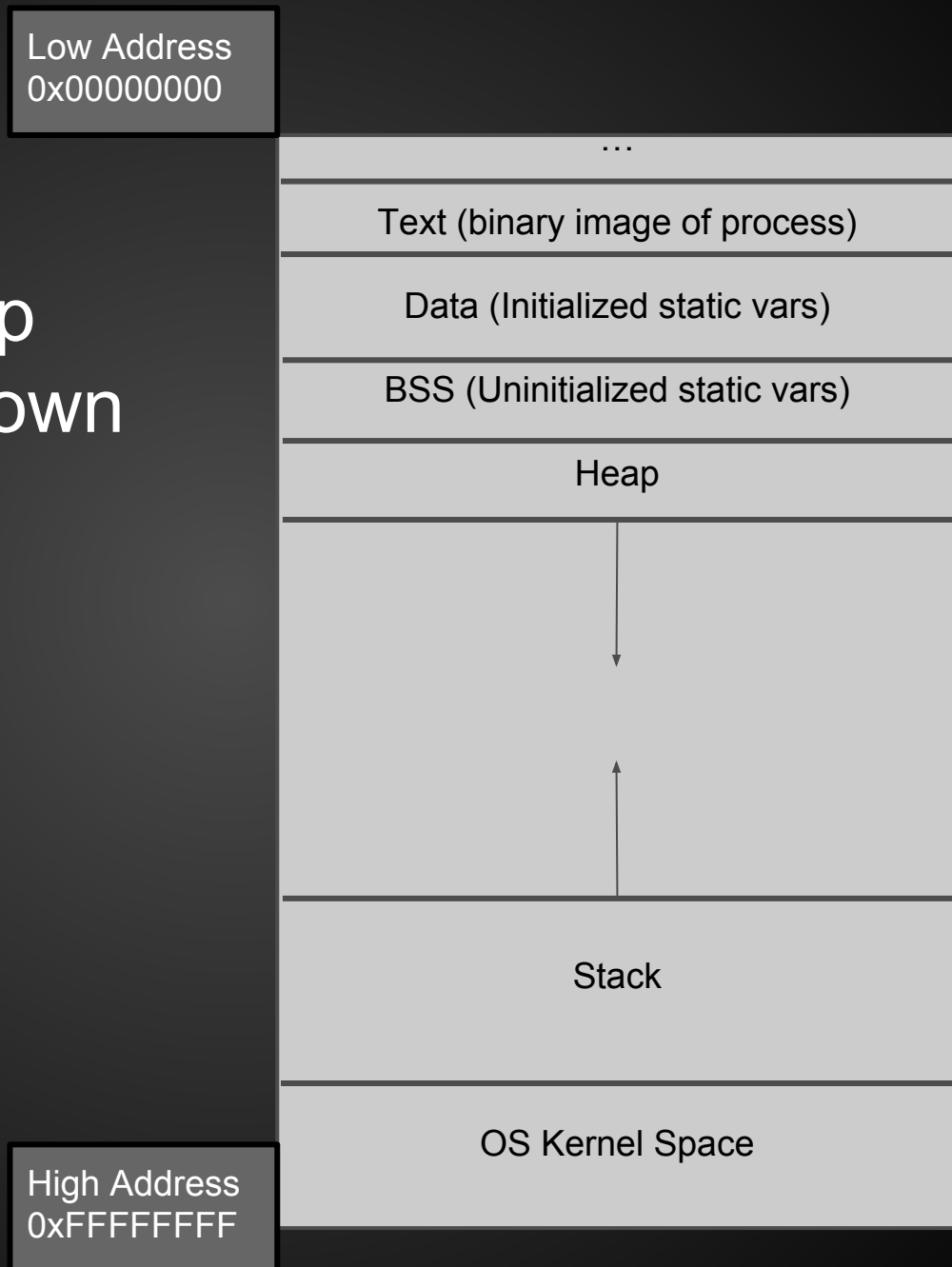


Instructions

- 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

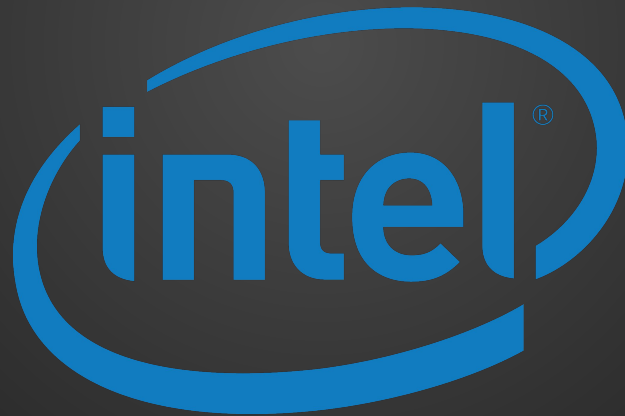
- Stack grows up
- Heap grows down



CHOOSE YOUR OWN ADVENTURE®

- x86
- MicroCorruption

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

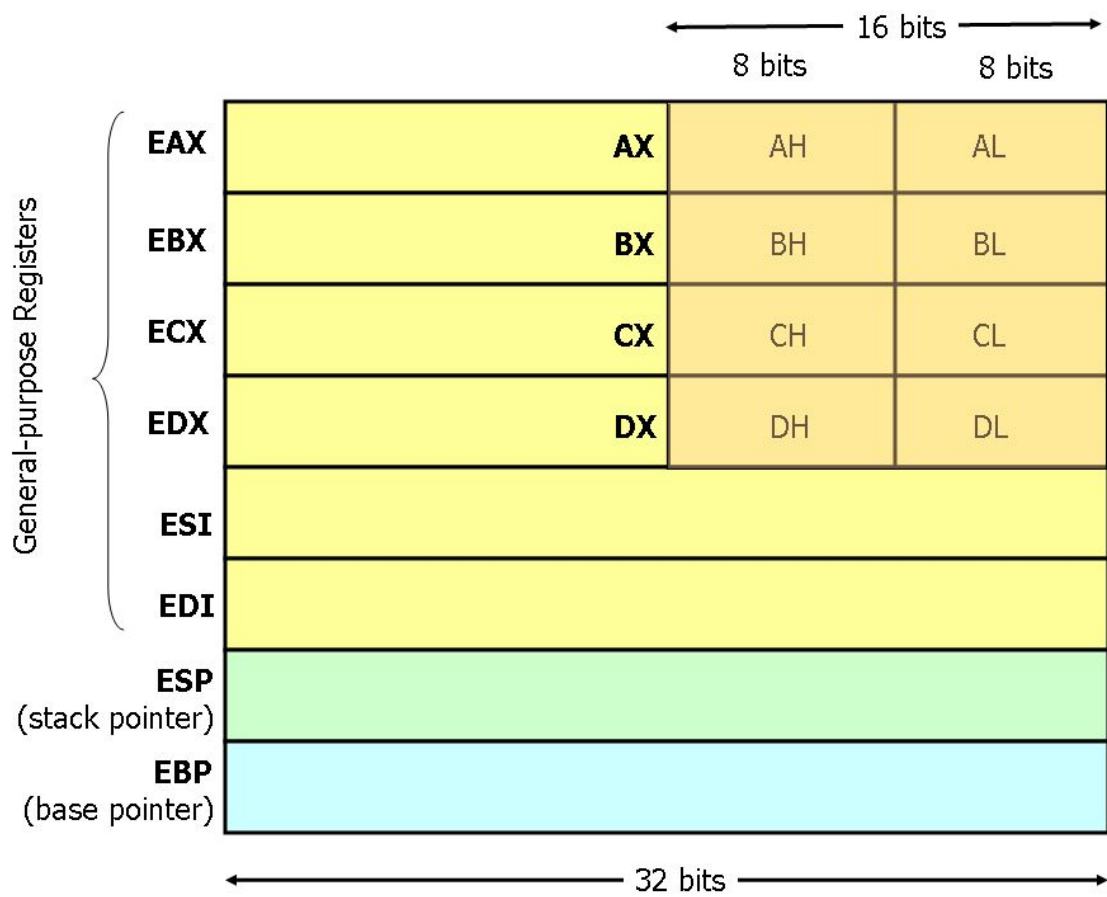
- Registers
- Syntax
- Common Instructions

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



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

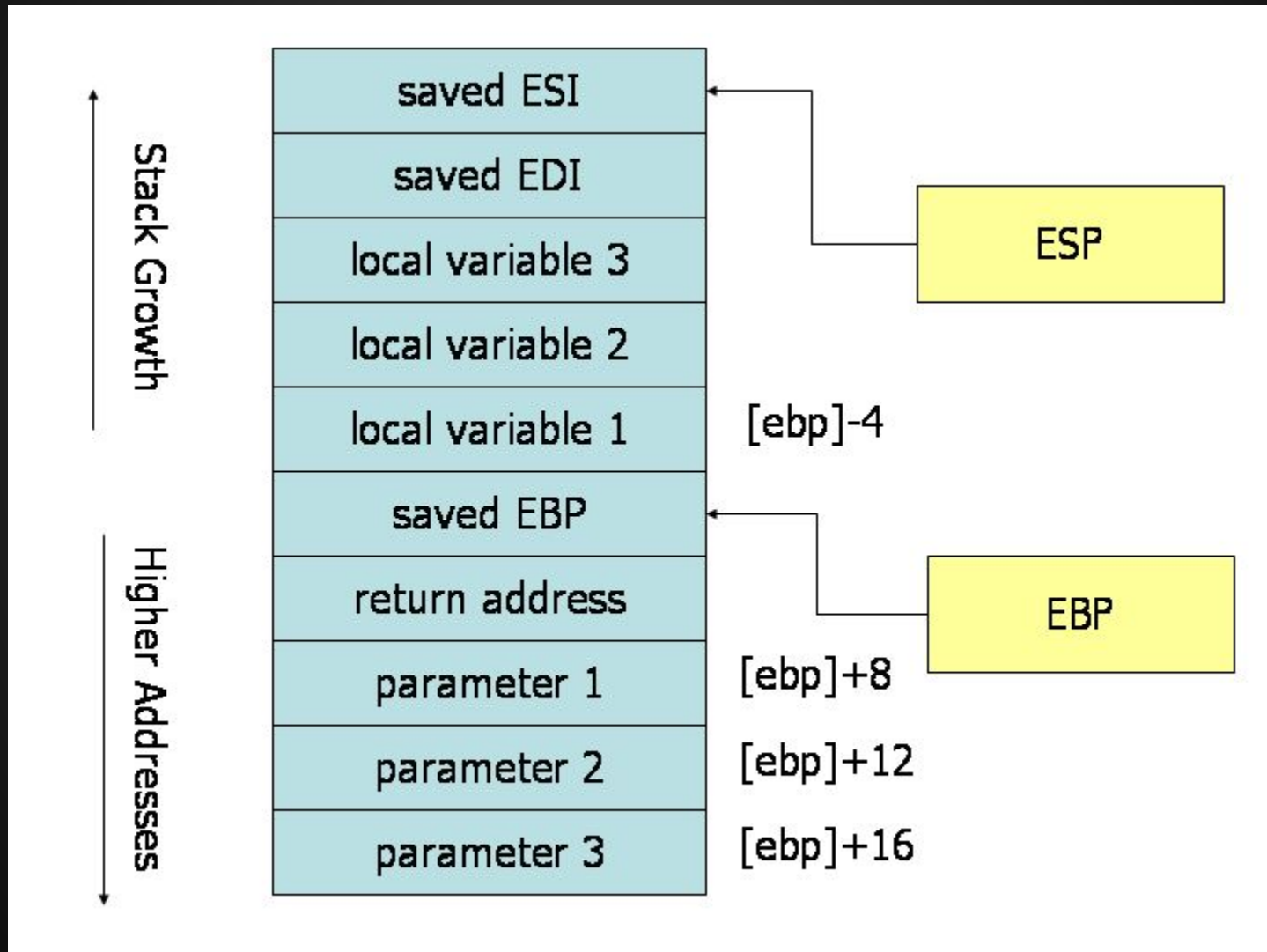
- **CMP**
 - Subtracts operands but discards result
 - Sets flags
- **TEST**
 - ANDs operands but discards result
 - Sets flags
- **JMP/Jxx**
 - JNE, JAE, etc

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; }
```



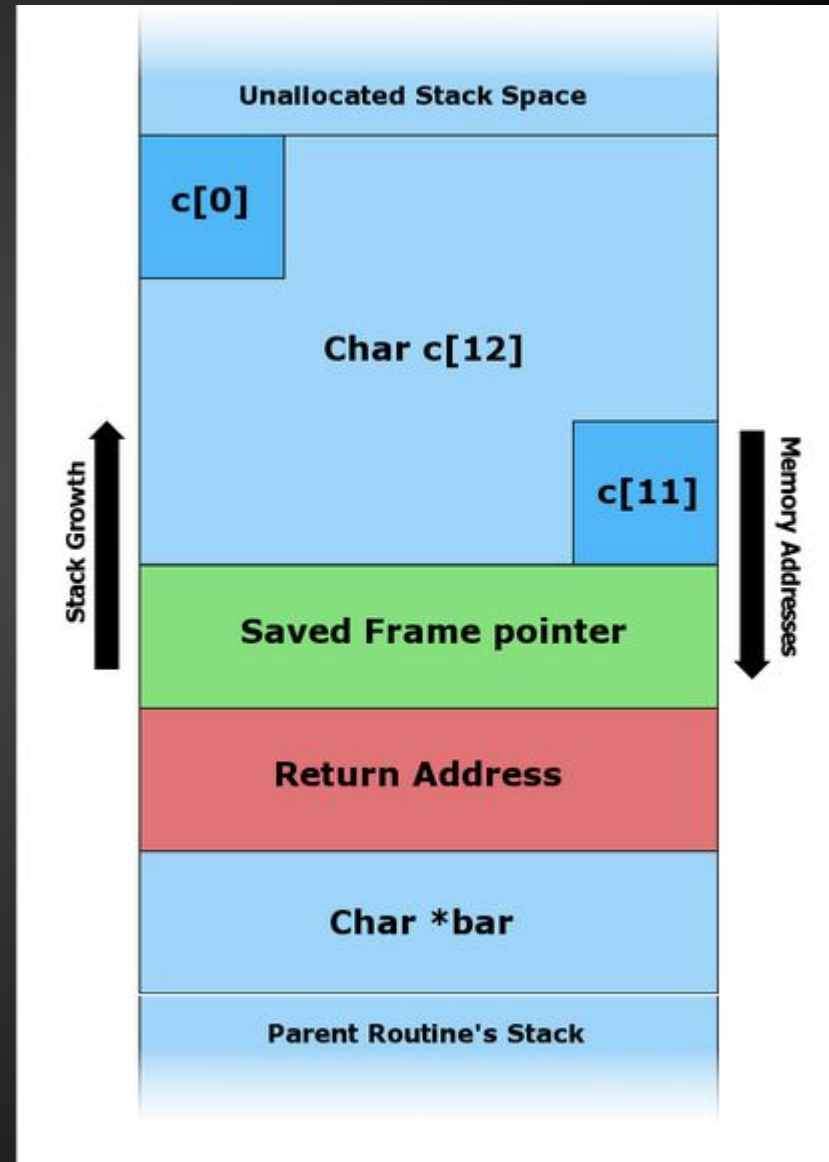
C Call Stack

```
#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

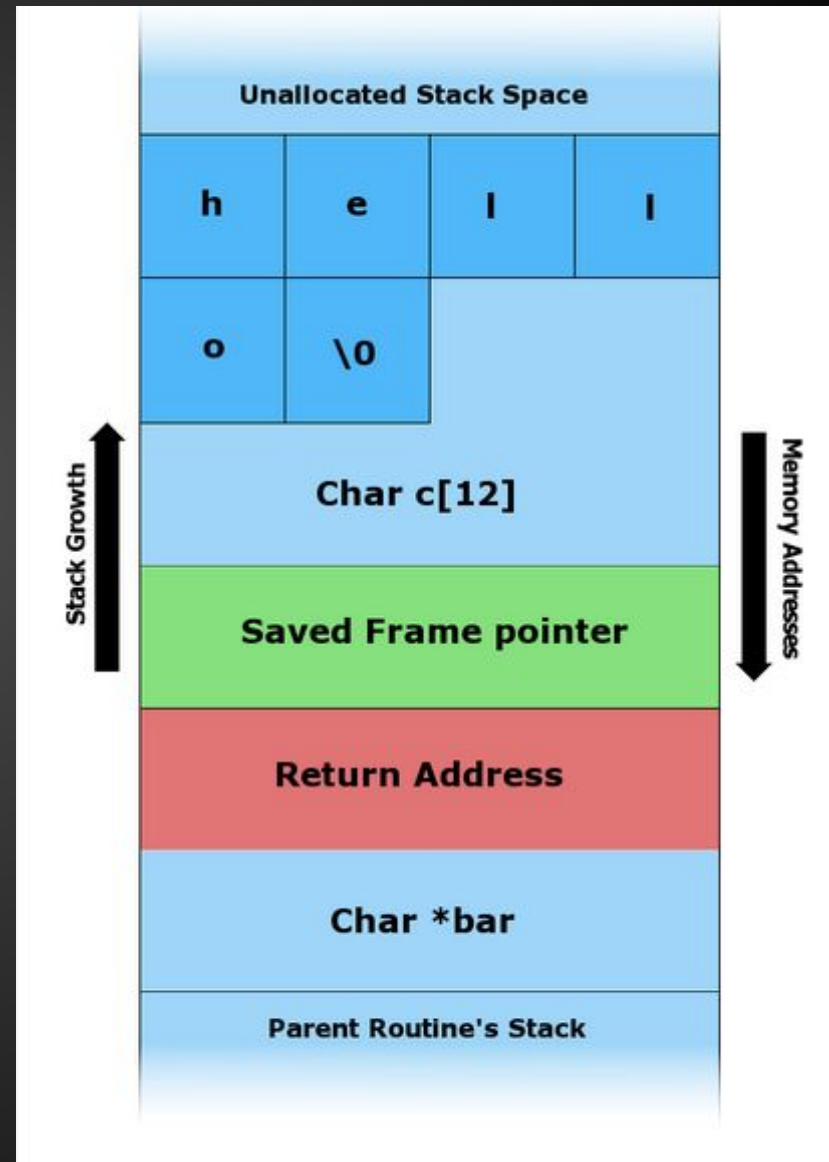
Input: "Hello"

```
#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"

```
#include <string.h>

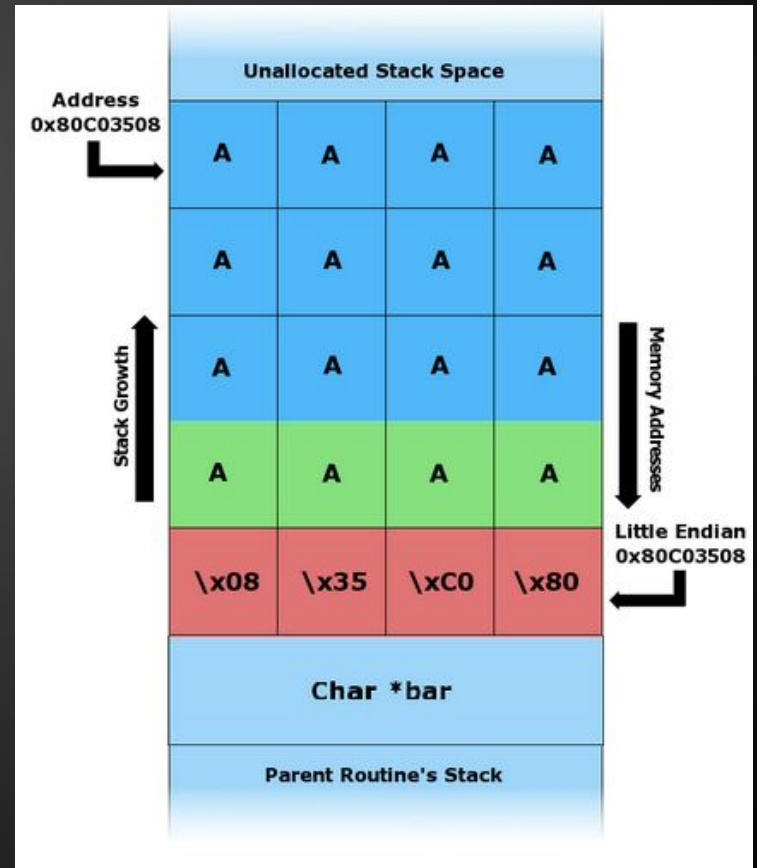
void foo (char *bar)
{
    char c[12];

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

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

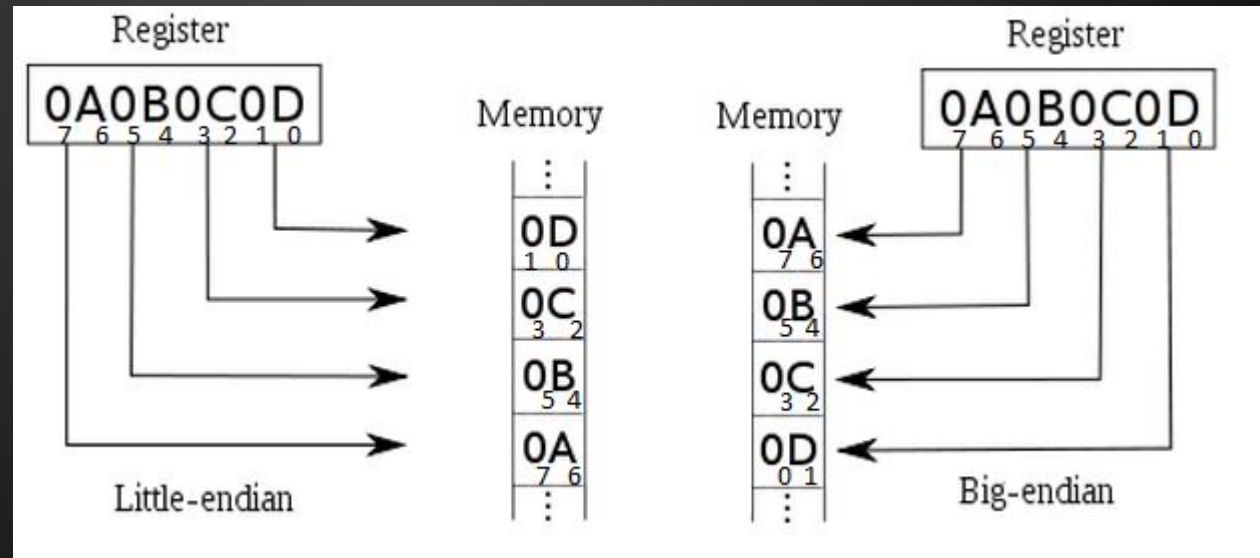
Hex (“\x0”) = 4 Bits = ½ Byte

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`