Binary Exploitation

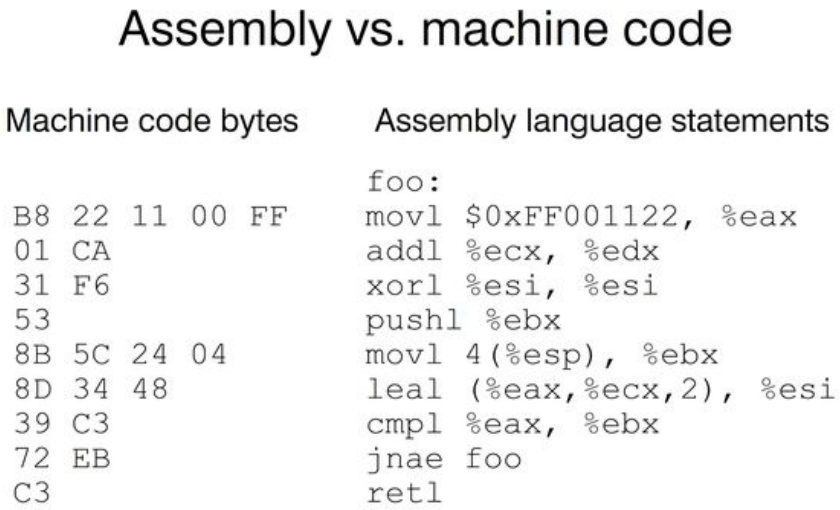
Jake Smith and Mariah Kenny
University of Virginia

Assembly

Credit: Most slides taken from Collin Berman and Cyrus Malekpour's Modern Security Topics class at UVA in Spring 2017

Basics

- Lowest level programming language
- Step by step instructions for CPU
- Results of compiled program
- Types: Intel and AT&T



(Data Movement) Instructions

- mov eax, ebx
  - eax = ebx
- mov eax, 123
  - eax = 123
- mov eax, DWORD PTR [0x123456]
  - eax = *(0x123456)
- mov eax, DWORD PTR [edx+esi*4]
  - eax = *(edx + esi * 4)
- lea esi, [ebp - 16]
  - esi = ebp - 16
  - Commonly used for pointer manipulations

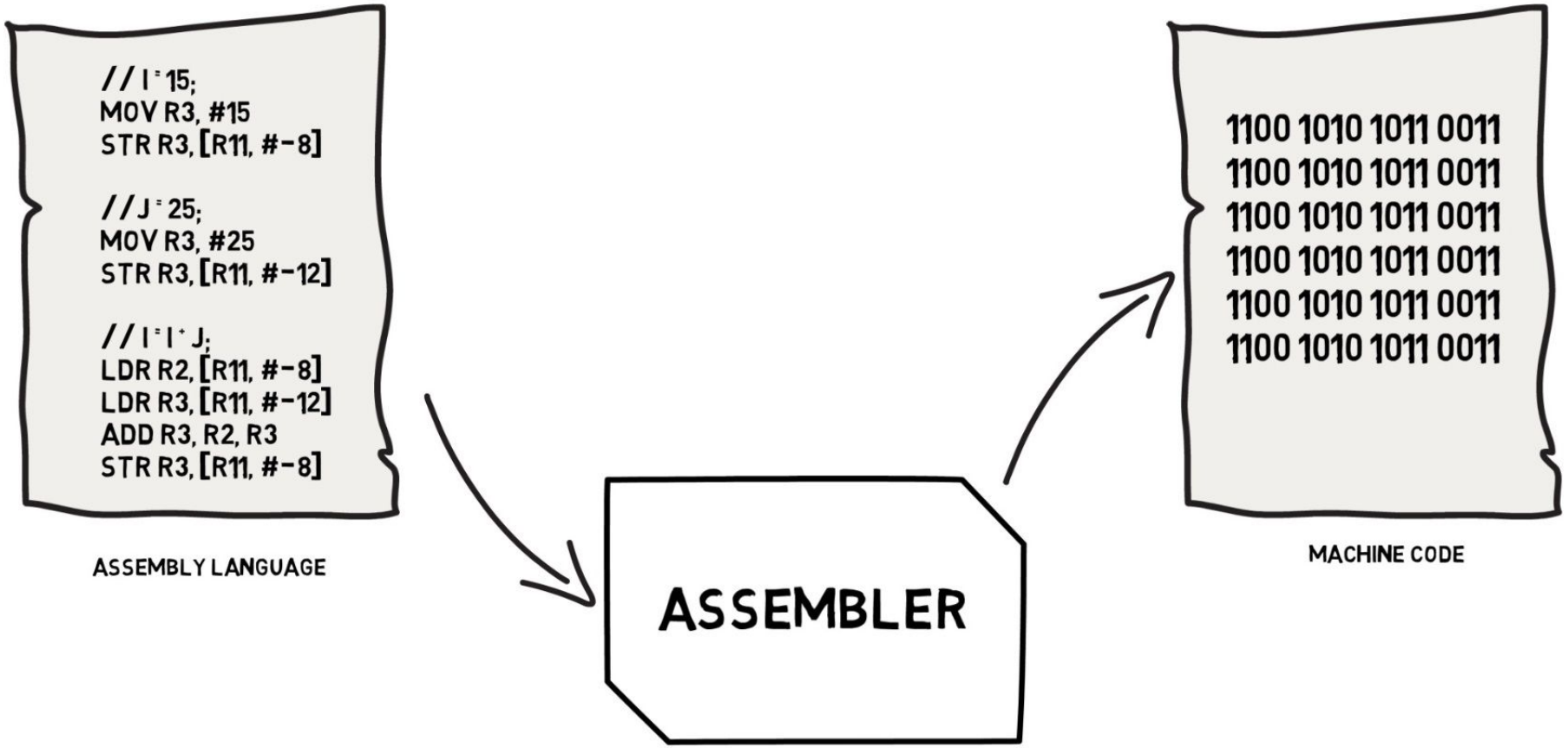
Addressing Modes

[ebx] [ebx - 4]
[ebx + eax] [ebx + eax*4]
[ebx-eax] ; Invalid
(Can only add registers)
[ebx + eax + ecx] ; Invalid
(Can only use 2 registers)

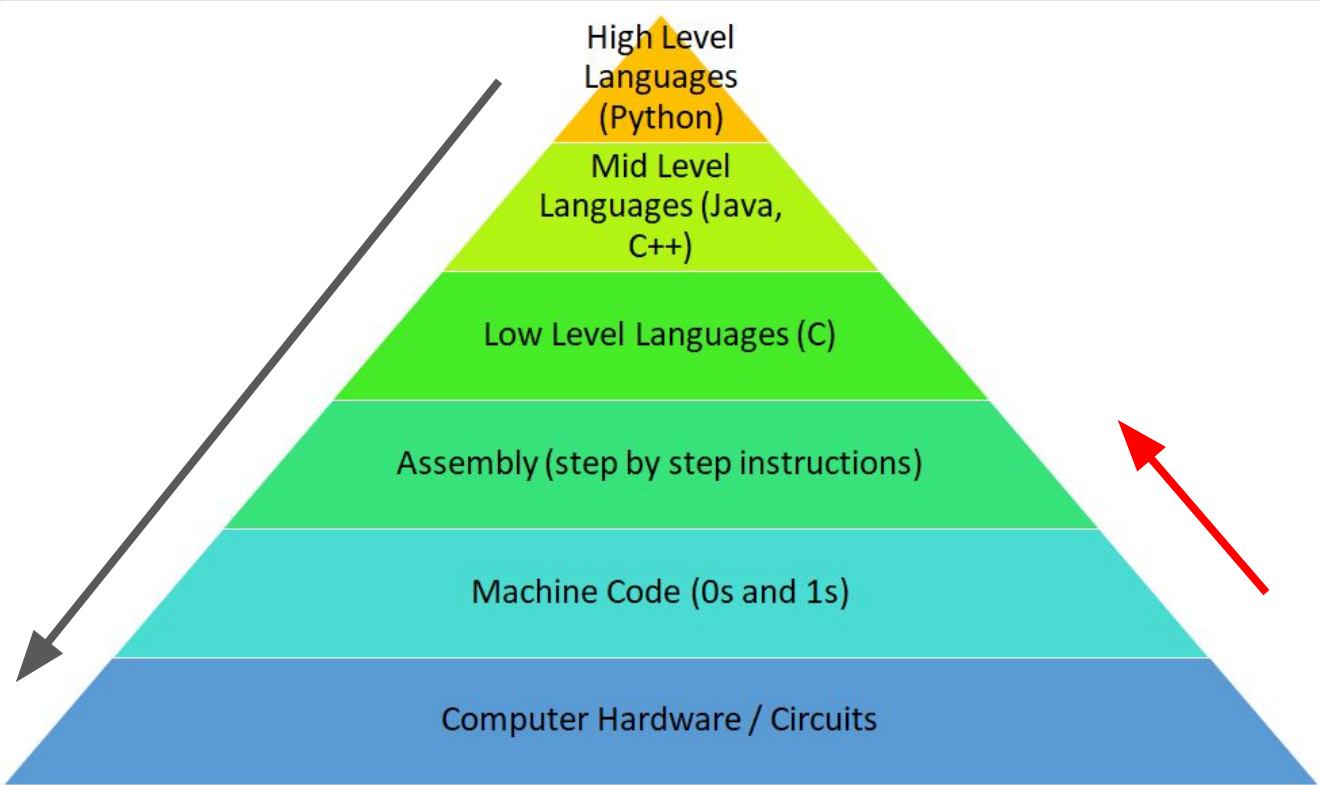
Size Directives

BYTE PTR [edx]
WORD PTR [edx]
DWORD PTR [edx]

Credit: Modern Security Topics, Spring 2017 at UVA

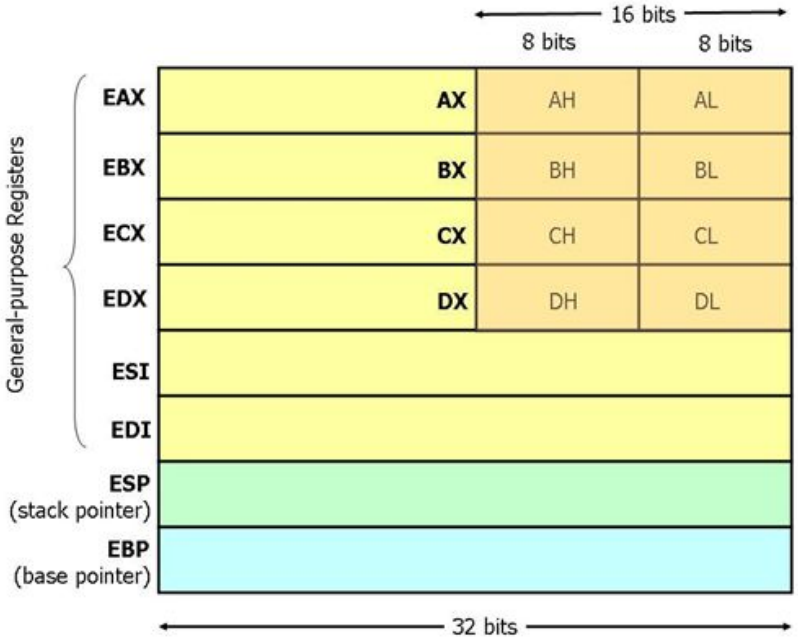


Levels of Abstraction



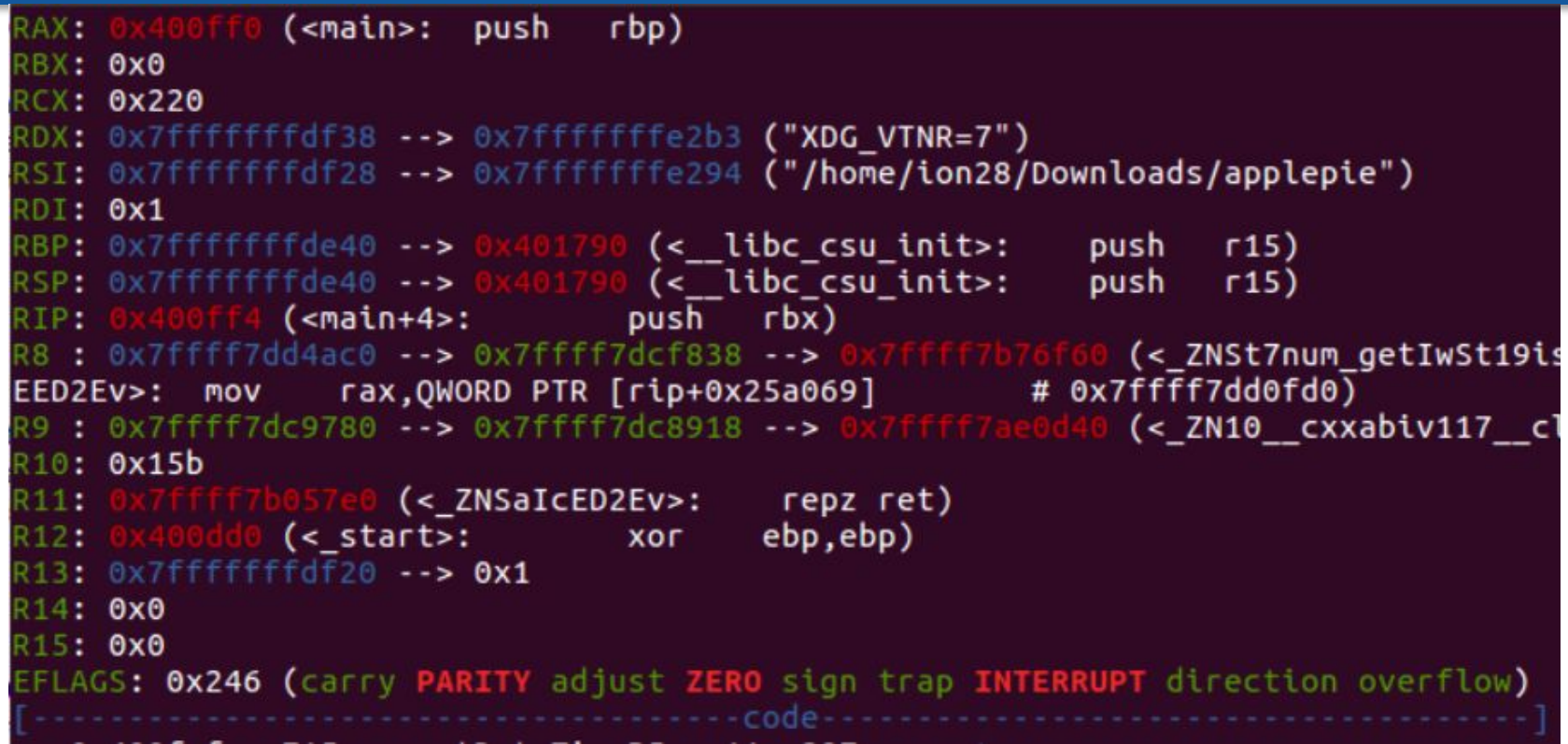
Registers

- **EAX**, **EBX**, **ECX**, **EDX**
  - Common general purpose registers
- **ESP**
  - Points to the “top” of the current stack frame
- **EBP**
  - Stack base pointer, points to the “bottom” of the current stack frame
- **EIP**
  - Points to the location of the current instruction in memory
- **EFLAGS**
  - Contains **flag bits** (zero flag, carry flag, sign flag, etc)



Credit: Modern Security Topics, Spring 2017 at UVA

Registers


RAX: 0x400ff0 (<main>: push rbp)
RBX: 0x0
RCX: 0x220
RDX: 0x7fffffffdf38 --> 0x7fffffffe2b3 ("XDG_VTNR=7")
RSI: 0x7fffffffdf28 --> 0x7fffffffe294 ("/home/ion28/Downloads/applepie")
RDI: 0x1
RBP: 0x7fffffffde40 --> 0x401790 (<__libc_csu_init>: push r15)
RSP: 0x7fffffffde40 --> 0x401790 (<__libc_csu_init>: push r15)
RIP: 0x400ff4 (<main+4>: push rbx)
R8 : 0x7ffff7dd4ac0 --> 0x7ffff7dcf838 --> 0x7ffff7b76f60 (<_ZNSt7num_getIwSt19is
EED2Ev>: mov rax,QWORD PTR [rip+0x25a069] # 0x7ffff7dd0fd0)
R9 : 0x7ffff7dc9780 --> 0x7ffff7dc8918 --> 0x7ffff7ae0d40 (<_ZN10__cxxabiv117__cl
R10: 0x15b
R11: 0x7ffff7b057e0 (<_ZNSaIcED2Ev>: repz ret)
R12: 0x400dd0 (<_start>: xor ebp,ebp)
R13: 0x7fffffffdf20 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[----------code----------]

(Arithmetic) Instructions

- `add eax, 0x123`
  - `eax = eax + 0x123`
- `sub eax, 0x456`
  - `eax = eax - 0x456`
- `and eax, ebx`
  - `eax = eax | ebx`
- `not ecx`
  - `ecx = ~ecx`
- `inc edx`
  - `edx++`
- `shl ecx, al`
  - `ecx = ecx << al`
- `mul ecx`
  - `(edx:eax) = ecx * eax`
- `div ecx`
  - `eax = (edx:eax) / ecx`
  - `edx = (edx:eax) % ecx`

Credit: Modern Security Topics, Spring 2017 at UVA

(Control Flow) Instructions

- jmp eax
  - Unconditional jump
- jz $my_location
  - Jump if zero flag set
- jnz $my_location
  - Jump if zero flag not set
- jg $my_location
  - Jump if greater
- jb $my_location
  - Jump if below

EFLAGS register

- Stores bit flags to indicate the results of operations
  - **Carry Flag**
  - **Zero Flag**
  - **Sign Flag**
- Implicitly set after certain instructions

```
mov eax, 5
cmp eax, 4
jg $cool_place
```

```
mov eax, 5
cmp eax, 5 jz
$also_cool
```

Credit: Modern Security Topics, Spring 2017 at UVA

x86 -> C

```
my_function:
    push ebp
    mov ebp, esp
    mov eax, [ebp+0x8]
    mov edx, [ebp+0xc]
body:
    mov ecx, [eax]
    cmp ecx, edx
    jz found
    cmp ecx, 0
    jz notfound
    inc eax
    jmp body
( rest omitted )
```

```
int my_function(char *d, char ch)
{
    int x = 0;
    while (d[x] != 0) {
        if (d[x] == ch)
            return x;
        x++;
    }
    return 0;
}
```

Credit: Modern Security Topics, Spring 2017 at UVA

Your First Program

```
# return 1;
.globl main
.intel_syntax noprefix

main:
mov eax, 1
ret
```

Run: `gcc -m32 -o simple simple.s`

#compiles program

Run: `./simple`

1

#runs program

Credit: Modern Security Topics, Spring 2017 at UVA

Your Second Program

```
# print the first command line argument to stdout
.intel_syntax noprefix

.text
.globl main
main:
push ebp
mov ebp, esp
mov ecx, [ebp+0xc]
mov ecx, [ecx+0x4]
push ecx
call puts
pop ecx
mov eax, 0
pop ebp
ret
```

Run: `gcc -m32 -o arg arg.s`

#compiles program

Run: `./arg stuff`

stuff

#runs program

Credit: Modern Security Topics, Spring 2017 at UVA

Basic Binary Analysis

file

- Looks at “magic bytes” - first few bytes of file
- Compares byte sequence to see what type of file it is
- ELF = Executable and Linking Format
- Executable/ELF file:
  - Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
- Syntax: `file <filename>`

`file`: CTF Problem

- Run file on `crackme1`
- What information can we deduce about the program?

`strings`

- Outputs all strings in the program
- Useful to see what you can deduce about program / its contents
- Syntax: `strings <filename>`

strings: CTF Problem

- You are given the executable: `crackme1`
- Use strings to find out what the flag probably is.

objdump

- Outputs information from “object files”
- `objdump -d`: prints disassembly of program
- `objdump -t`: Prints out symbol table (headers, function names, etc)
- Example Syntax: `objdump -d <filename>`

`objdump`: CTF Problem

- Run `objdump -d` and `objdump -t` on game
- What are some interesting function calls?
- What can we infer from these?

GDB

Basics

- Command-line debugger
- Specifically the GNU debugger
  - “GNU” is the compiler framework that contains a lot of things including the debugger gdb
- It allows you to see what is going on ‘inside’ the program while it executes

Basics

- Allows us to control the execution of the program
- Ability to pause, resume, determine the values of variables, reset variable values, etc.
- If the program crashes, the debugger can tell you exactly where the program crashed

Commands

- To open a file in gdb: gdb <filename>
- Once in gdb, to run the program: run or r
- To see current and surrounding lines: list
- To see list of function calls that led to current point in program: backtrace or bt (important command!)

Frames

- To move to a higher frame: up
- To move down a frame: down
- ^these let you move up and down the calling stack (of nested function calls)

Breakpoints

- Pausing the program at a specific place (specific line or start of a function)
- These locations in a program where execution pauses are called “breakpoints”
- You must use code that executes, cannot be a comment, etc.

Breakpoints

- break or b followed by what you want to pause
  - function name: b my_function
  - Line number: b 13
- To see info about breakpoints: info breakpoints or info break

Breakpoints

- To remove a breakpoint: delete or d
- To remove a specific breakpoint: d 2 or d my_func
- Temp breakpoint: tbreak

Controlling Execution

- Execute line by line
- step command steps into a function; moves into called function (s)
- next command passes over the function call and brings you to the line after the function call (n)
- continue command resumes execution until next breakpoint (c)

Variables

- To see the value of a variable or expression: print or p followed by variable name
- If var is a pointer or address: print *<var> which will print the value that the address references
- To see all args and local variables: info locals

Display

- To auto display variable values: display <var>
- If see all variables on display: display
- To remove a variable from display: undisplay <display var #>

GDB PEDA

- Python Exploit Development Assistance for GDB (more colorful and helpful)
- To download:
  - git clone https://github.com/longld/peda.git ~/peda
  - echo “source ~/peda/peda.py” >> ~/.gdbinit

Practice

- Goal: find the flag
- Open program in gdb: gdb animal1.exe
- Set breakpoints at main or any interesting functions (b main, etc) or use disas main
- Run program (run)
- Use step (s) and next (n) to move through program

Questions?