Binary Analysis II

Assembly, calling conventions, and dynamic analysis

Start Week 4 →


```
7a85 dabd 8b48 892c a7c3 4cb4 e24c 3b40
8e66 2eb8 7ac1 a36d 95dc b150 8b84 3d02
782e 32bf d9d7 f400 f1ad 7fac b258 6fc6
e966 c004 d7d1 d16b 024f 5805 ff7c b47c
7a85 dabd 8b48 892c a7ad 7fac b258 6fc6
7a85 dabd 8b48 892c a7ad 7fac b258 6fc6
e966 c004 d7d1 d16b 024f 5805 ff7c b47c
371b f798 90fb 1861 2d53 e282 bb5e 8cd0
7aea 31e9 9659 d7d9 f6ad 7fac b258 6fc6
```

Assembly Review

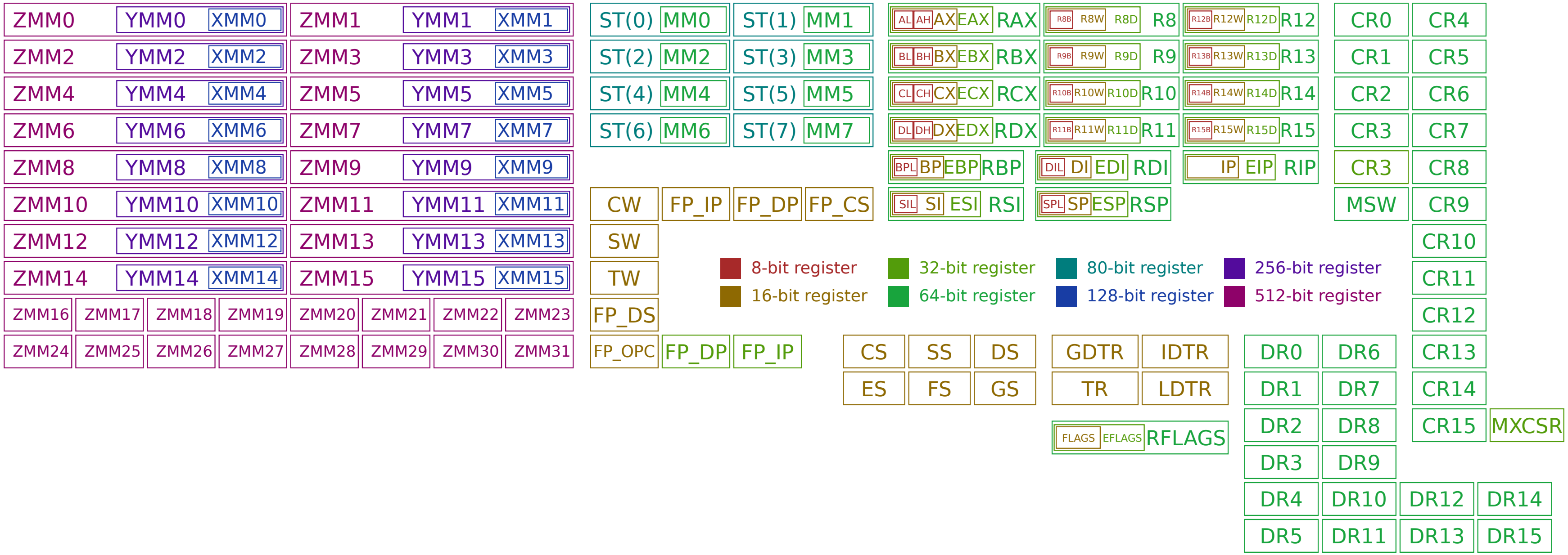
Low-level programming language that is translated into the the architecture's byte-code. Here we will use the x86_64 architecture.

Common Architectures

- Intel x86/x86_64: Used by most desktop computers
- ARM: Used by mobile devices, IOT devices, and (increasingly) desktop computers
- MIPS: Used mostly by IOT devices

Registers

- General purpose: `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `rsp`, `rbp`, `r8` - `r15`
- Special purpose: program counter, counters, flags, floating point arithmetic, etc.



General Purpose Registers

These registers may be used for any purpose but generally follow these conventions.

Name	64-bit	32-bit	16-bit	Conventional Use
Accumulator	`rax`	`eax`	`ax`	Used for the return value of functions
Base	`rbx`	`ebx`	`bx`	Used as a pointer to data
Counter	`rcx`	`ecx`	`cx`	Used in loops and shift operations
Data	`rdx`	`edx`	`dx`	Used in I/O operations and arithmetic
Stack Pointer	`rsp`	`esp`	`sp`	Points to the top of the stack
Base Pointer	`rbp`	`ebp`	`bp`	Points to the base of the current stack frame

Moving Data Around

Move immediates and other data between registers and the stack.

```
mov eax, 0x01          ;put 1 into eax
mov [eax], 0x01        ;put 1 into the address in eax
mov eax, [esi]         ;put contents of address (esi)
```

Push data on onto the stack.

```
push eax               ;put contents of eax on top of stack
push 0x01              ;put 1 on top of stack
                       ; and inc the stack pointer

pop eax                ;put contents top of the stack into eax,
                       ; and dec the stack pointer
```

Displacements.

```
; [base + index*size + offset]
; size can only be 1,2,4,8
mov eax, [arr + esi*4 + 0]
```

Moving Data Around

Load effective address does not access memory with the displacement operator! It only does the pointer arithmetic with no dereference!

```
lea eax, ecx   ;invalid
lea eax, [ecx] ;valid, equivalent to mov eax, ecx

lea eax, [ecx + edx]   ;mov eax, ecx + edx*1 (implicit 1)
lea eax, [ecx + edx*3] ;invalid, valid numbers are 1,2,4,8

lea eax, [eax + edx*4] ;can be thought of as
                       ; eax = (DWORD *)eax[edx] why?
```

Comparisons and Branching

Assembly provides instructions for comparing values and controlling program flow based on the results.

Comparison Instructions

```
cmp eax, ebx    ;compare eax with ebx (sets flags)
test eax, eax   ;bitwise AND of eax with itself (sets flags)
```

Conditional Jumps

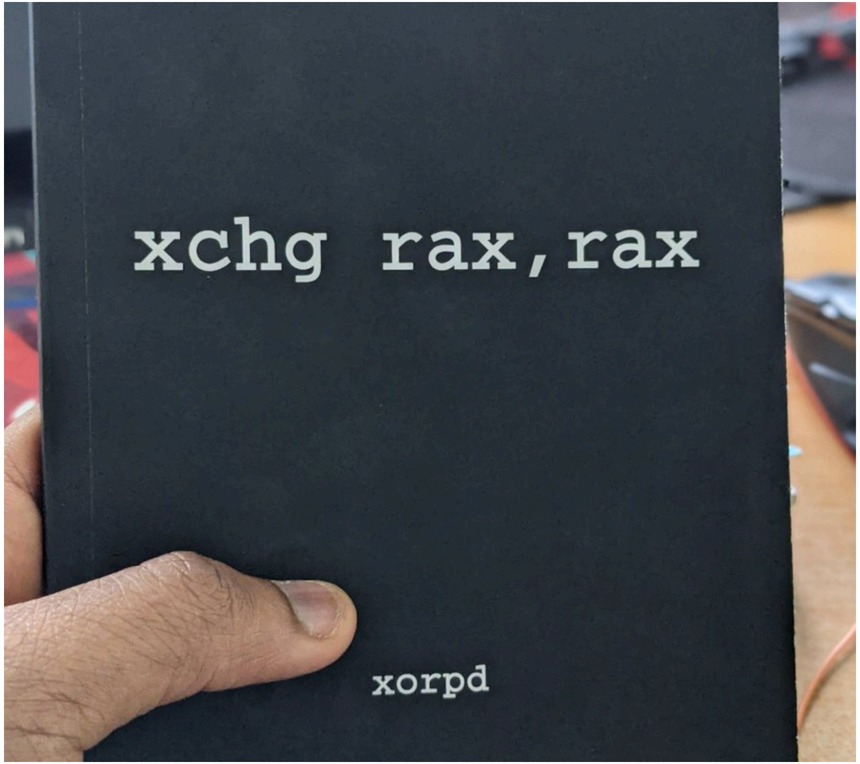
```
je addr  ; or jz  -- if zero flag is set (equal)
jne addr ; or jnz -- if zero flag is not set (not equal)
jg addr  ; or ja  -- if greater - signed or unsigned
jl addr  ; or jb  -- if less    - signed or unsigned
jge addr ;        -- if greater or equal to
jle addr ;        -- if less or equal to
```

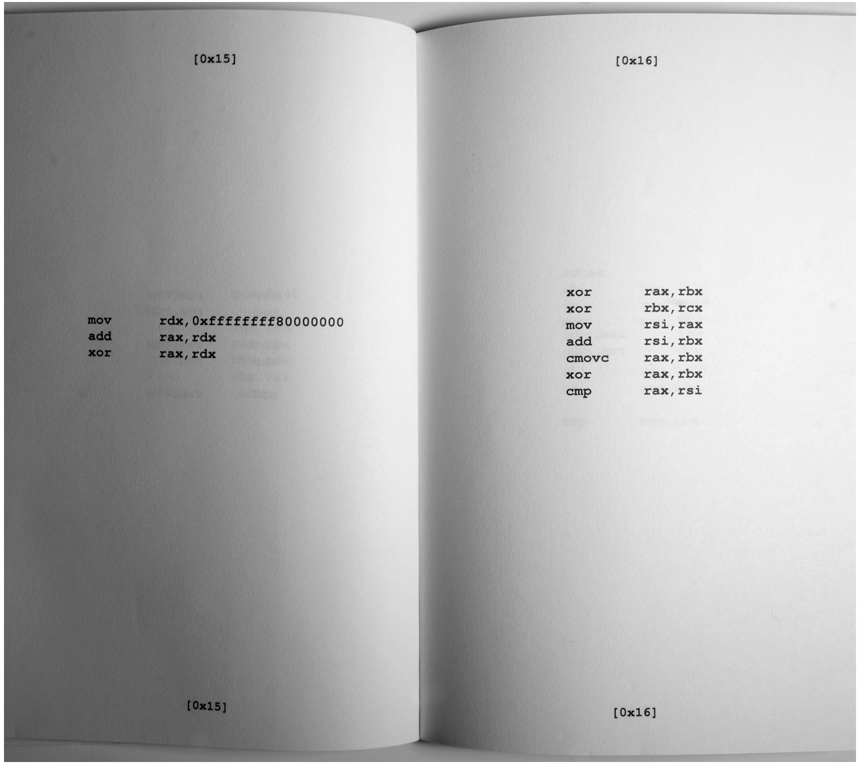
Common flags used by comparisons

```
CF (Carry Flag)    -- used to indicate carry in arithmetic operation
ZF (Zero Flag)     -- if a value is zero or comparison equals 0
SF (Sign Flag)     -- if negative
```

Assembly Practice

The following examples are adapted from this book of assembly riddles.





Assembly Practice 1

Let’s walk through each line.

```
mov     rdx,0
xor     eax,eax
and     esi,0
sub     edi,edi
push    0
pop     rbp
```

Different ways of setting a register to zero.

Assembly Practice 2

What value ends up in `rax` after this code executes?

```
main:
  mov rax, 0x100
  mov rbx, 0x200
  cmp rax, rbx
  je equal_label

  mov rax, 0x300
  jmp end_label

equal_label:
    mov rax, 0x400
end_label:
```

```
rax = 0x300
```

Assembly Practice 3

What value ends up in `rax` after this code executes?

```
mov rax, 0x0
mov rbx, 0x1
mov rcx, 0x3
loop_start:
    test rcx, rcx
    jle  loop_end
    xchg rax, rbx
    add  rax, rbx
    dec  rcx
    jg   loop_start
loop_end:
```

Computes Fibonacci numbers by swapping `rax` and `rbx` each loop, adding them to produce the next term in `rax`, decrementing `rcx` until zero. The loop executes three times, so `rax` = 1, 1, 2.

Application Binary Interface (ABI)

An ABI defines how compiled code interacts at runtime. This includes:

- How general purpose registers are used
- Calling conventions
- Data types sizes and layouts
- Symbols and linkage rules
- System call interfaces
- etc

System V ABI

Defines the calling convention for x86_64 architecture, specifying how functions are called and how registers are used. It's the standard for Linux and includes the ELF format.

www.sco.com/developers/gabi/latest/contents.html

System V Application Binary Interface - DRAFT - 10 June 2013

Contents

Revision History

Chapter 4 - Object Files

- Introduction
- File Format
- Data Representation

System V ABI

The ABI defines how calling conventions and return values are passed.

Function Arguments

Arguments are passed in registers in this order:

1. `rdi` - 1st argument
2. `rsi` - 2nd argument
3. `rdx` - 3rd argument
4. `rcx` - 4th argument
5. `r8` - 5th argument
6. `r9` - 6th argument

Additional arguments are passed on the stack.

Return Values

- `rax` - Integer return value
- `rdx:rax` - 128-bit return value
- `xmm0` - Floating point return value

Comparing different calling conventions

Comparison of 32-bit and 64-bit calling conventions.

Aspect	cdecl (32-bit)	System V (64-bit)
Arguments	All on stack (right-to-left)	First 6 in registers, others on stack
Return Values	Integer: `eax` 64-bit: `edx:eax` 128-bit: Not supported Floating point: Not supported	Integer: `rax` 64-bit: `rdx:rax` 128-bit: `rdx:rax` Floating point: `xmm0`

Why might newer calling conventions use registers instead of the stack for passing arguments?

Register Volatility

By convention, some registers are expected to be preserved across function calls and some are not.

Volatile Registers

Caller-saved - Values may be modified by called functions

- `rax` - Return value
- `rcx` - 4th argument
- `rdx` - 3rd argument
- `rsi` - 2nd argument
- `rdi` - 1st argument
- `r8-r11` - other arguments

Non-Volatile Registers

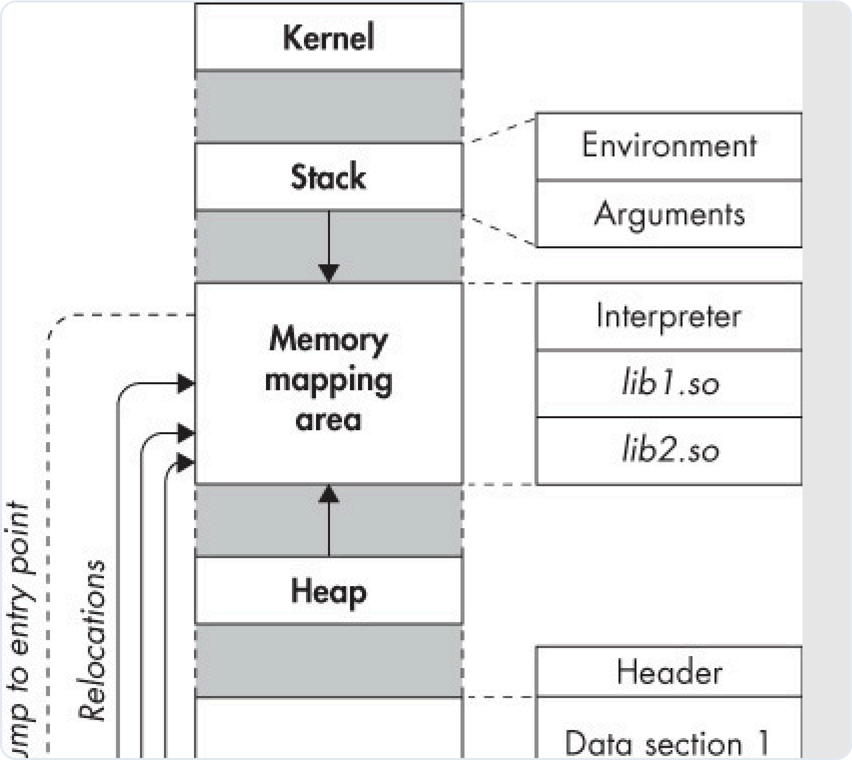
Callee-saved - Values must be preserved by called functions

- `rbx` - Must be preserved
- `rbp` - Frame pointer
- `rsp` - Stack pointer
- `r12` - Must be preserved
- `r13` - Must be preserved
- `r14` - Must be preserved
- `r15` - Must be preserved

Virtual Memory Layout

Last week we talked about the sections and segments. Here's a broader view of the memory layout.

- **Kernel** - Usually situated at a high address
- **Stack** - Grows downward
- **Heap** - Managed by allocator but roughly grows upward
- **Data** - At a low address
- **Code** - At a low address



Dynamic Analysis

Last week we introduced static analysis. This can be laborious for large binaries, so it's useful to analyze a program when it's running. Debugging with GDB is one strategy.

Command	Description
`gdb ./program`	Start GDB with program
`run`	Start execution
`quit`	Exit GDB
`help`	Show help
`info registers`	Show all registers
`info breakpoints`	List breakpoints
`break main`	Set breakpoint at main

GDB Commands

Do we remember GDB commands?

1. What command shows the current instruction pointer?

   `info registers rip` or `x/i $rip`

2. How do you set a breakpoint at the `main` function?

   `break main` or `b main`

3. How do you set a breakpoint at `0x400000`?

   `break *0x400000` or `b *0x400000`

4. How do you continue execution after hitting a breakpoint?

   `continue` or `c`

GDB Commands

Do we remember GDB commands?

5. What's the difference between `step` and `next`?

   `step` goes into function calls, `next` steps over them

6. What does `x/s $rdi` do?

   Examines the memory pointed to by RDI as a string

7. How do you examine memory at address `0x401000` as hex?

   `x/x 0x401000`

8. What command sets the value of RAX to 0?

   `set $rax = 0`

GEF Plugin

To aid debugging for reverse engineering we've set up the `gef` plugin for GDB. It provides a number of new commands and a new view.

```
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ────
$rax   : 0x0
$rbx   : 0x0
$rcx   : 0x00007ffff7ffcca0  →  0x0004095d00000000
$rdx   : 0x0
$rsp   : 0x00007fffffffe530  →  0x0000000000000000
$rbp   : 0x00007fffffffe560  →  0x00000000004007f0  →  <__libc_csu_init+0> push r15
$rsi   : 0x00007ffff7dd1b78  →  0x0000000000602000  →  0x0000000000000000
$rdi   : 0x20000
$rip   : 0x0000000000400799  →  <main+64> mov QWORD PTR [rbp-0x28], rax
$r8    : 0x00007ffff7fec700  →  0x00007ffff7fec700  →  [loop detected]
$r9    : 0x1
$r10   : 0x0
$r11   : 0x246
$r12   : 0x0000000000400580  →  <_start+0> xor ebp, ebp
$r13   : 0x00007fffffffe640  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume virtualx86 identification]
$ss: 0x002b  $cs: 0x0033  $ds: 0x0000  $gs: 0x0000  $es: 0x0000  $fs: 0x0000
──────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffe530│+0x0000: 0x0000000000000000   ← $rsp
0x00007fffffffe538│+0x0008: 0x0000000000000000
0x00007fffffffe540│+0x0010: "myfile.txt"
0x00007fffffffe548│+0x0018: 0x0000000000007478 ("xt"?)
```

Lab 1

Using GDB.

https://hacs408e.umd.edu/labs/week-03/lab-1/

86

MINUTES

− +

Ends at 6:15 PM

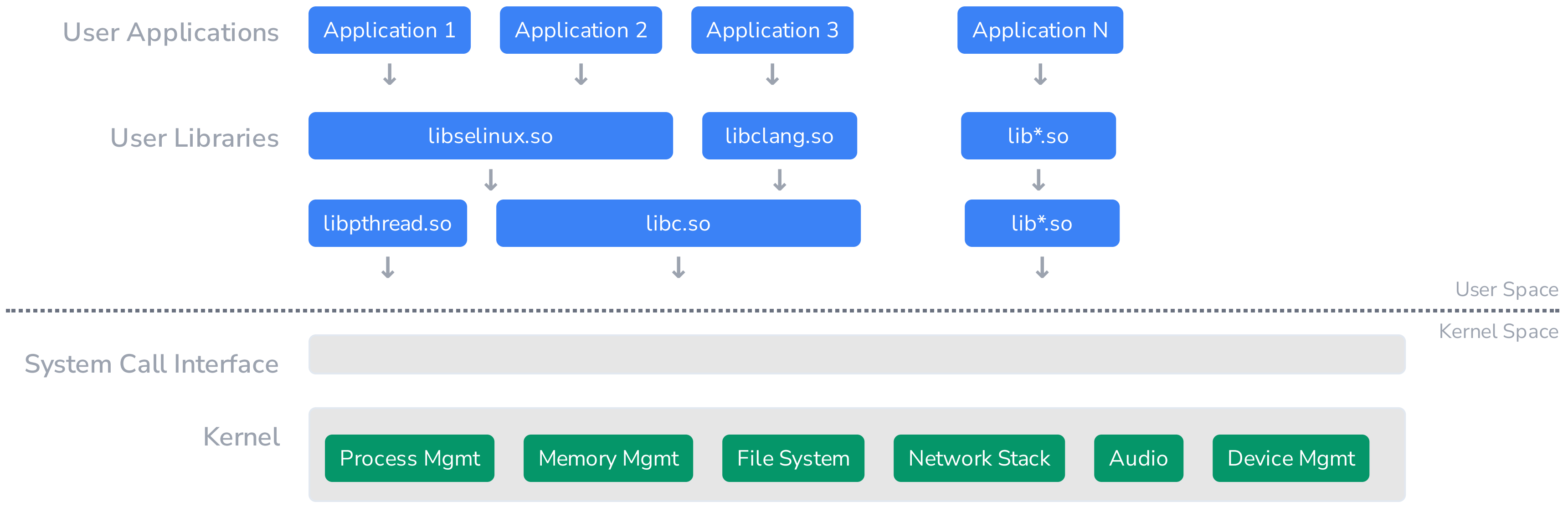
Tracing

Debugging is an appropriate solution for fine-grained or precise analysis. For faster results it's sometimes useful to trace only specific aspects of your program's behavior.

- Data written to `stdout` / `stderr`
- Changes to the system file system
- Library function calls
- System calls

Library Loading

Processes may load dynamic libraries that provide shared functionality.



Library and System Call Tracing

Tracing the library and system calls within processes can provide useful information. On linux the following utilities support this.

- `ltrace` tracing library calls
- `strace` tracing system calls

Lab 2

https://hacs408e.umd.edu/schedule/week-04/lab-2/

171

MINUTES

− +

Ends at 7:40 PM

Homework

Homework 2 is due next week.

Quiz 1 is next week at the beginning of class.