Reverse Engineering

Mainack Mondal

CS60112
Spring 2023-24

Outline

- Reverse Engineering?
- x86_64 Assembly : Instructions, Registers, Memory
- Reading the Instruction Set Architecture
- Programming in x86 : Examples
- ELF

Assembly

- Programming constructs at the Instruction Set Architecture (ISA)
  - Mainly x86_64 in this course
- Main components
  - Instructions : Form *opcode <operand 1>, <operand 2> …*
  - Registers : mapped to the register file in hardware
  - Memory : stack, heap, shared memory, other mapped memory

Instructions

- General form: *opcode <operand 1>, <operand 2> …*
  - opcode: Uniquely identifies the instruction being executed
  - operands: Data being operated upon by the instruction
- A good source for x86 instruction set: https://www.felixcloutier.com/x86/

Register File

- "Volatile" memory *closest* to the processor
  - Extremely fast access
  - Expensive to place upon a core (hence is limited)

Register	Accumulator	Counter	Data	Base	Stack Pointer	Stack Base Pointer	Source	Destination
64-bit	RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI
32-bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
16-bit	AX	CX	DX	BX	SP	BP	SI	DI
8-bit	AH, AL	CH, CL	DH, DL	BH, BL	SPL	BPL	SIL	DIL

- Several registers usable in x86: RAX, RCX, RBX. Other registers like R9, R10, R11…
- Allows addressing 64-bits (entire width), 32-bits, 16-bits, 8-bits
  - Ex: Using **eax** accesses lower 32-bits; using **ah** accesses penultimate byte
- **RIP** handles the instruction pointer
- Flags status register (interpreted as individual *bits* wrt. ALU events)

Memory

- (For this course) Other volatile/non-volatile hardware than registers

- Organized as hierarchy trading off access times VS storage capacity
    - Less expensive, high storage memory is *slow* to access

- Memory can be accessed through
    - *offsetting* **rbp** or **rsp** in x86 for local variables (examples ahead)
        - **DWORD PTR** [**rbp-4**] to read a double-word (4-bytes) offset at **rbp-4**
        - **QWORD PTR [rsp-32]** to read a quad-word (8-bytes) offset at **rsp-32**
    - *offsetting* **rip** for global/static variables

Instructions - Reading the ISA

- Example: **add**
- **Step 1:** Understand instruction semantics (operand types, sign extensions etc.)

Description

Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

- **Step 2:** Understand x86 semantics for that instruction

ADD — Add

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
04 ib	ADD AL, imm8	I	Valid	Valid	Add imm8 to AL.
05 iw	ADD AX, imm16	I	Valid	Valid	Add imm16 to AX.
05 id	ADD EAX, imm32	I	Valid	Valid	Add imm32 to EAX.

Ex. Assume you need to *add* a constant 0x1000 (> 8 bit integer representation) to some variable **x**, and that **x** is stored within **rax**. One possible way: **add ax, 0x1000**.

Instructions - Executing (Local)

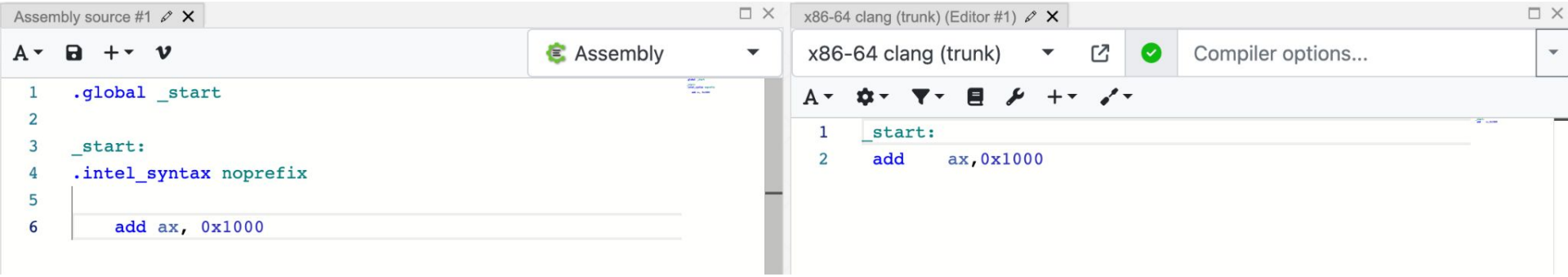
- One possible way of executing the simple logic discussed in previous slide
- In **code.S**, define a label (say **_start**) which becomes the starting point of execution

```
.global _start

_start:
.intel_syntax noprefix

    add ax, 0x1000
```

- Within **_start**, program logic as a series of x86 instructions
- Compile with **gcc -nostdlib -static code.S -o code-elf**
    - Flag **-static** disables any dynamic linking
    - Flag **-nostdlib** disables use of system libraries while linking

Instructions - Executing (godbolt)

- Play around with https://godbolt.org/
  - Define your source language (Assembly or C or C++)
  - Choose your compiler (along with compilation flags)



Programming in x86 - Arithmetic

C

```c
int source1;
int dummy() {
    static int source2;
    int destination, source3, source4;

    destination = source3 + source4;
    __asm__ __volatile__("nop");
    destination = source1 / source3;
    __asm__ __volatile__("nop");
    destination = source2 >> 2;
}
```

x86-64 gcc 13.2 -nostdlib -static -O0

```
source1:
        .zero   4
dummy:
        push    rbp
        mov     rbp, rsp
        mov     edx, DWORD PTR [rbp-4]
        mov     eax, DWORD PTR [rbp-8]
        add     eax, edx
        mov     DWORD PTR [rbp-12], eax
        nop
        mov     eax, DWORD PTR source1[rip]
        cdq
        idiv    DWORD PTR [rbp-4]
        mov     DWORD PTR [rbp-12], eax
        nop
        mov     eax, DWORD PTR source2.0[rip]
        sar     eax, 2
        mov     DWORD PTR [rbp-12], eax
        nop
        pop     rbp
        ret
```

- Points to note
  - Use of **mov** instructions to exchange data between memory/registers
  - Semantics of **add**, **idiv**, **sar**

Programming in x86 - Arithmetic

C

```
1  int source1;
2  int dummy() {
3      static int source2;
4      int destination, source3, source4;
5
6      destination = source3 + source4;
7      __asm__ __volatile__("nop");
8      destination = source1 / source3;
9      __asm__ __volatile__("nop");
10     destination = source2 >> 2;
11 }
12
```

x86-64 gcc 13.2 -nostdlib -static -O0

```
1  source1:
2          .zero   4
3  dummy:
4          push    rbp
5          mov     rbp, rsp
6          mov     edx, DWORD PTR [rbp-4]
7          mov     eax, DWORD PTR [rbp-8]
8          add     eax, edx
9          mov     DWORD PTR [rbp-12], eax
10         nop
11         mov     eax, DWORD PTR source1[rip]
12         cdq
13         idiv    DWORD PTR [rbp-4]
14         mov     DWORD PTR [rbp-12], eax
15         nop
16         mov     eax, DWORD PTR source2.0[rip]
17         sar     eax, 2
18         mov     DWORD PTR [rbp-12], eax
19         nop
20         pop     rbp
21         ret
```

- **idiv:** *Signed divide EDX:EAX by r/m32, with result stored in EAX := Quotient, EDX := Remainder*.

Line 11 brings *source1* into eax. **idiv** then divides EDX:EAX **32-bit** register pair, with 4-byte memory at location [**rbp-4**], which is *source3* (see line 6; it is the **first** local variable of this function). Finally, quotient, contained in **eax**, is stored at [**rbp-12**] (which is *destination*; i.e. the **third** local variable of this function).

Programming in x86 - Control Flow

C

```
1  int dummy() {
2      int number, temp;
3      for(int i = 0; i < 100; i++){
4          if(number % 2)
5              temp = number * 2;
6      }
7  }
8
```

x86-64 gcc 13.2 -nostdlib -static -O0

```
1   dummy:
2           push    rbp
3           mov     rbp, rsp
4           mov     DWORD PTR [rbp-4], 0
5           jmp     .L2
6   .L4:
7           mov     eax, DWORD PTR [rbp-8]
8           and     eax, 1
9           test    eax, eax
10          je      .L3
11          mov     eax, DWORD PTR [rbp-8]
12          add     eax, eax
13          mov     DWORD PTR [rbp-12], eax
14  .L3:
15          add     DWORD PTR [rbp-4], 1
16  .L2:
17          cmp     DWORD PTR [rbp-4], 99
18          jle     .L4
19          nop
20          pop     rbp
21          ret
```

- **Loop**: Notice the *yellow* shaded portions. In line 4-5, the loop variable at **[rbp-4]** is initialized to 0. Then a **jmp .L2** executes line 17-18 to **cmp** the value at **[rbp-4]** against 99 (loop termination condition). If this comparison yields "less-than" result, perform a **jump-on-less-or-equal** (jle) jump to **.L4** which executes loop body.

- **Condition:** In line numbers 7-10, **[rbp-8]** is loaded into eax. A logical AND and subsequent test checks the LSB. Should the LSB be 1, jump to line number 15 (increment loop variable), and continue looping. Otherwise, do an additional add to **[rbp-12].**

Programming in x86 - Calling convention

- Function calls are performed by the **call** instruction
- Semantics:
  - Arguments are passed through specific registers
  - Return value is stored in **rax**

arch	syscall NR	return	arg0	arg1	arg2	arg3	arg4	arg5
x86	eax	eax	ebx	ecx	edx	esi	edi	ebp
x86_64	rax	rax	rdi	rsi	rdx	r10	r8	r9

Programming in x86 - Calling convention

C source #1

```c
int function(int a, int b){
    return a + b - 4;
}

int dummy() {
    int x, y;
    int result = function(x, y);
    return result * 2;
}
```

x86-64 gcc 13.2 (Editor #1) — x86-64 gcc 13.2 — -static -O0

```
1   function:
2           push    rbp
3           mov     rbp, rsp
4           mov     DWORD PTR [rbp-4], edi
5           mov     DWORD PTR [rbp-8], esi
6           mov     edx, DWORD PTR [rbp-4]
7           mov     eax, DWORD PTR [rbp-8]
8           add     eax, edx
9           sub     eax, 4
10          pop     rbp
11          ret
12  dummy:
13          push    rbp
14          mov     rbp, rsp
15          sub     rsp, 16
16          mov     edx, DWORD PTR [rbp-8]
17          mov     eax, DWORD PTR [rbp-4]
18          mov     esi, edx
19          mov     edi, eax
20          call    function
21          mov     DWORD PTR [rbp-12], eax
22          mov     eax, DWORD PTR [rbp-12]
23          add     eax, eax
```

- Notice the *blue* shaded part. The function call **function(x, y)** involves **x** as the first argument (which is also **[rbp-4]**. Likewise, second argument **y** is **[rbp-8]**. Notice how **esi** takes **edx** (where **y** is stored). Likewise, **edi** takes **eax**, where **x** is stored. Finally, inside the function, notice how the final expression is stored in **eax** (line 9). When control flow returns to line 21 (from line 11), the value in **eax** is stored at **[rbp-12]**, pointing to **result.**

Programming in x86 - Calling convention

C source #1

```c

int function(int a, int b, int c, int d,
             int e, int f, int g, int x, int y){
    return - 4 + x;
}


int dummy() {
    int a, b, c, d, e, f, g, x, y;
    int result = function(a, b, c, d, e, f, g, x, y);
    return result * 2;
}

```

x86-64 gcc 13.2 (Editor #1) — x86-64 gcc 13.2 — -static -O0

```
function:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     DWORD PTR [rbp-16], ecx
        mov     DWORD PTR [rbp-20], r8d
        mov     DWORD PTR [rbp-24], r9d
        mov     eax, DWORD PTR [rbp+24]
        sub     eax, 4
        pop     rbp
        ret
```

- **Passing more than 6 arguments:** If more than 6 arguments are passed, then additional arguments are picked directly from the stack **from the previous function's function frame**.

In this note, observe the *green* portion. Note that the first 6 arguments are moved within the *called function's frame* (indicated by the negative offsets in **rbp**). However, **x** is picked directly from function **dummy**'s function frame, and hence is accessed by **[rbp+24]** (notice the positive offset). The return value is in **rax**, as before.

Programming in x86 - System Calls

- Function calls which have well defined interface and role
    - Example: read/write syscalls, used for opening a input/output file stream in C
- A good source: https://chromium.googlesource.com/chromiumos/docs/+/HEAD/constants/syscalls.md

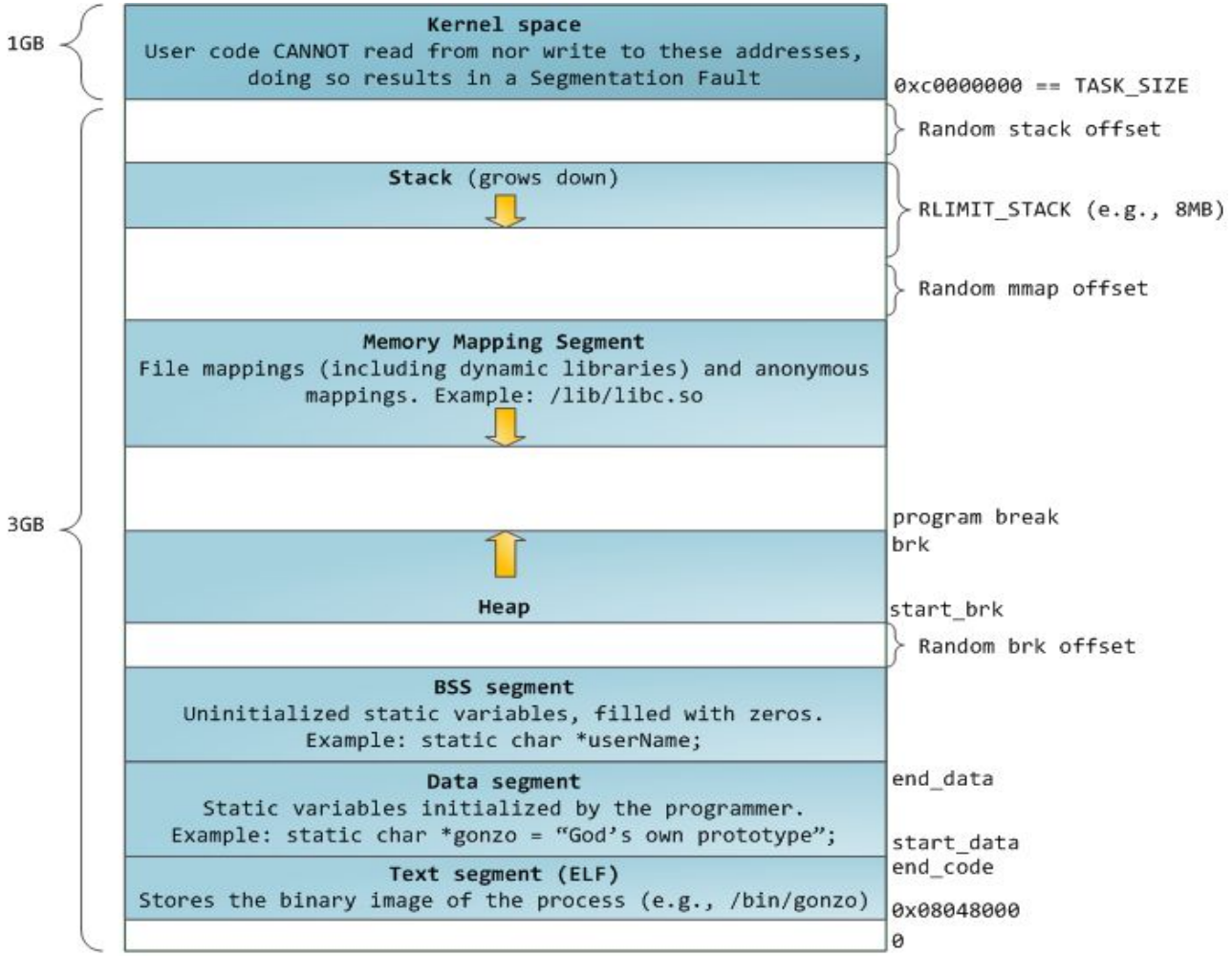
NR	syscall name	references	%rax	arg0 (%rdi)	arg1 (%rsi)	arg2 (%rdx)	arg3 (%r10)	arg4 (%r8)	arg5 (%r9)
0	read	man/ cs/	0x00	unsigned int fd	char *buf	size_t count	-	-	-
1	write	man/ cs/	0x01	unsigned int fd	const char *buf	size_t count	-	-	-

To execute a syscall, mov the syscall identifier in **rax**. Arguments go into other registers as specified by the syscall interface. Finally, execution of a **syscall** instruction executes the system call. **Example:** print “hello world” can be performed by

- Shift **0x01** to **rax** and **0x1** to **rdi** (which is the file identifier for stdout in Linux)
- Shift the starting address of **char*** buffer containing “hello world” to **rsi**
- Shift the number of characters to print into **rdx**
- Execute **syscall** instruction

ELF : Executable and Linkable Format

- The format for storing binaries on Unix based systems



ELF : Executable and Linkable Format

- Example

```c
#include <stdio.h>

int global_var = 0;
int main(){
        int local_var = 10;
        static int static_local_var = 20;

        // some computation
        int temp = global_var + local_var + static_local_var;
        printf("%d\n", temp);

        return 0;
}
```

Things to note: initialized global variable, static local variable, local variable, printf call ….

Compile with: gcc -static <filename>.c -o <executable_name>

ELF : Executable and Linkable Format

- ELF Header

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401b90
  Start of program headers:          64 (bytes into file)
  Start of section headers:          869864 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         10
  Size of section headers:           64 (bytes)
  Number of section headers:         32
  Section header string table index: 31
```

ELF header has info like magic bytes (uniquely identifying the binary as ELF), architecture (32-bit vs 64-bit), encoding of data, details of program/section headers.

Command: readelf -h <executable_name>

ELF : Executable and Linkable Format

- Use **objdump -D -M intel <executable_name>** to parse ELF into readable format

```
0000000000401cb5 <main>:
  401cb5:       f3 0f 1e fa             endbr64
  401cb9:       55                      push   rbp
  401cba:       48 89 e5                mov    rbp,rsp
  401cbd:       48 83 ec 10             sub    rsp,0x10
  401cc1:       c7 45 f8 0a 00 00 00    mov    DWORD PTR [rbp-0x8],0xa
  401cc8:       8b 15 a2 05 0c 00       mov    edx,DWORD PTR [rip+0xc05a2]        # 4c2270 <global_var>
  401cce:       8b 45 f8                mov    eax,DWORD PTR [rbp-0x8]
  401cd1:       01 c2                   add    edx,eax
  401cd3:       8b 05 17 e4 0b 00       mov    eax,DWORD PTR [rip+0xbe417]        # 4c00f0 <static_local_var.2317>
  401cd9:       01 d0                   add    eax,edx
  401cdb:       89 45 fc                mov    DWORD PTR [rbp-0x4],eax
  401cde:       8b 45 fc                mov    eax,DWORD PTR [rbp-0x4]
  401ce1:       89 c6                   mov    esi,eax
  401ce3:       48 8d 3d 1a 33 09 00    lea    rdi,[rip+0x9331a]        # 495004 <_IO_stdin_used+0x4>
  401cea:       b8 00 00 00 00          mov    eax,0x0
  401cef:       e8 6c ec 00 00          call   410960 <_IO_printf>
  401cf4:       b8 00 00 00 00          mov    eax,0x0
  401cf9:       c9                      leave
  401cfa:       c3                      ret
  401cfb:       0f 1f 44 00 00          nop    DWORD PTR [rax+rax*1+0x0]
```

Things to note: global and static data is being picked from addresses 0x4c2270 and 0x4c00f0 respectively. These come in **.bss** section and **.data** section respectively (detailed in next slide). Likewise, **IO_stdin_used** resides in **.rodata** (read-only data section)

ELF : Executable and Linkable Format

- Disassembly of **.bss**, **.data**, and **.rodata** sections, showing **global_var, static_local_var**, and **IO_stdin_used** respectively

```
Disassembly of section .bss:

00000000004c2220 <completed.7507>:
        ...

00000000004c2240 <object.7512>:
        ...

00000000004c2270 <global_var>:
        ...
```

```
Disassembly of section .data:

00000000004c00e0 <__data_start>:
        ...

00000000004c00e8 <__dso_handle>:
        ...

00000000004c00f0 <static_local_var.2317>:
  4c00f0:       14 00                   adc    al,0x0
  4c00f2:       00 00                   add    BYTE PTR [rax],al
  4c00f4:       00 00                   add    BYTE PTR [rax],al
        ...
```

```
Disassembly of section .rodata:

0000000000495000 <_IO_stdin_used>:
  495000:       01 00                   add    DWORD PTR [rax],eax
  495002:       02 00                   add    al,BYTE PTR [rax]
  495004:       25 64 0a 00 78          and    eax,0x78000a64
  495009:       65 6f                   outs   dx,DWORD PTR gs:[rsi]
  49500b:       6e                      outs   dx,BYTE PTR ds:[rsi]
  49500c:       5f                      pop    rdi
```