A Crash Course on X86 Assembly

COMP3703 Software Security

Slides prepared by Alwen Tiu. Based on D. Andriesse's Practical Binary Analysis (Appendix A).

Motivation and scope

- Learn how assembly programs are structured
- Some common patterns assembly output by compilers.
- *Not* a course on how to write assembly programs.
- Focus on the 64-bit variant of x86

Outline

- Registers
- Layout of an assembly program
- Memory operands
- The structure of the stack
- Function calls and function frames
- Conditional branches
- Loops

ISA: RISC vs. CISC

- RISC, e.g., ARM
  - Single-cycle instructions;
  - Small number of instructions.
- CISC, e.g., Intel x86:
  - An instruction can take several clock cycles;
  - May support microcode;
  - Large number of instructions.
  - Variable-length instructions

Assembly x86-64

- Side effect flags: flags N (negative), Z (zero), C (carry), and V (o[illegible] are equal to Intel SF, ZF, CF, and OF bits
- On 32-bits, function arguments are put in the stack.
- On 64-bits, some of the registers are also used for function arguments: rdi, rsi, rcx, rdx, r8, r9, and then the stack.
- eip/rip: program counter/instruction pointer.

Assembly x86-64 – Registers

8-Bytes	Byte 0-3	Byte
rax	eax	ax
rbx	ebx	bx
rcx	ecx	cx
rdx	edx	dx
rsi	esi	si
rdi	edi	di
rsp	esp	sp
rbp	ebp	bp

8-Bytes	Byte 0-3	Byte 0-1
r8	r8d	r8w
r9	r9d	r9w
r10	r10d	r10w
r11	r11d	r11w
r12	r12d	r12w
r13	r13d	r13w
r14	r14d	r14w
r15	r15d	r15w

Assembly x86-64 – Registers

Layout of an assembly program

Each assembly program (produced by gcc) has these components:

- The instructions: actual operations the CPU executes.
- Directives: commands to tell the assembler to produce/place a piece of data in a particular section.
- Labels: symbolic names to refer to instructions/data.
- Comments: human-readable strings for documentation.

Layout of an assembly program

```
#include <stdio.h>

int
❶ main(int argc, char *argv[])
{
  ❷printf(❸"Hello, world!\n");

  return 0;
}
```


```
    .file "hello.c"
    .intel_syntax noprefix
❹   .section .rodata
.LC0:
❺   .string "Hello, world!"
❻   .text
    .globl main
    .type   main, @function
❼ main
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     DWORD PTR [rbp-4], edi
    mov     QWORD PTR [rbp-16], rsi
❽   mov     edi, OFFSET FLAT:.LC0
❾   call    puts
    mov     eax, 0
    leave
    ret
    .size   main, .-main
    .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9)"
    .section .note.GNU-stack,"",@progbits
```

Layout of an assembly program

```
#include <stdio.h>

int
❶ main(int argc, char *argv[])
{
  ❷printf(❸"Hello, world!\n");

  return 0;
}
```

```
    .file "hello.c"
    .intel_syntax noprefix
❹   .section .rodata
.LC0:
❺   .string "Hello, world!"
❻   .text
    .globl main
    .type   main, @function
❼ main
    push   rbp
    mov    rbp, rsp
    sub    rsp, 16
    mov    DWORD PTR [rbp-4], edi
    mov    QWORD PTR [rbp-16], rsi
❽   mov    edi, OFFSET FLAT:.LC0
❾   call   puts
    mov    eax, 0
    leave
    ret
    .size   main, .-main
    .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9)"
    .section .note.GNU-stack,"",@progbits
```

Instructions

Layout of an assembly program

```
#include <stdio.h>

int
❶ main(int argc, char *argv[])
{
  ❷printf(❸"Hello, world!\n");

  return 0;
}
```

```
.file "hello.c"
.intel_syntax noprefix
❹ .section .rodata
.LC0:
❺ .string "Hello, world!"
❻ .text
.globl main
.type   main, @function
❼ main
push   rbp
mov    rbp, rsp
sub    rsp, 16
mov    DWORD PTR [rbp-4], edi
mov    QWORD PTR [rbp-16], rsi
❽ mov    edi, OFFSET FLAT:.LC0
❾ call   puts
mov    eax, 0
leave
ret
.size   main, .-main
.ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9)"
.section .note.GNU-stack,"",@progbits
```

Directives: define a string and place it in section .rodata.

Layout of an assembly program

```
#include <stdio.h>

int
❶ main(int argc, char *argv[])
{
  ❷printf(❸"Hello, world!\n");

  return 0;
}
```


```
.file "hello.c"
.intel_syntax noprefix
❹ .section .rodata
.LC0:
❺ .string "Hello, world!"
❻ .text
.globl main
.type   main, @function
❼ main
push   rbp
mov    rbp, rsp
sub    rsp, 16
mov    DWORD PTR [rbp-4], edi
mov    QWORD PTR [rbp-16], rsi
❽ mov    edi, OFFSET FLAT:.LC0
❾ call   puts
mov    eax, 0
leave
ret
.size   main, .-main
.ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9)"
.section .note.GNU-stack,"",@progbits
```

Labels: to be translated to actual memory locations.

Memory operands

- Memory operand[illegible]y a memory address where the CPU should fetc[illegible] more bytes.
- The x86 supports only one explicit memory operand per instructions.
  - Can't directly copy bytes from one memory location to another – you need a register as intermediate storage.
- A memory operand is specified with

  [base + index*scale + displacement]

Memory operands

[*base* + *index**scale* + *displacement*]

- *base, index*: These are any registers (e.g., rax, rbx, etc).
- scale: An integer value with the value 1,2,4 or 8. Default (if not specified) is one.
- displacement: a 32-bit constant or a symbol. Default is 0.

Memory operands

Example:

mov eax, DWORD PTR [**rax***4 + **arr**]

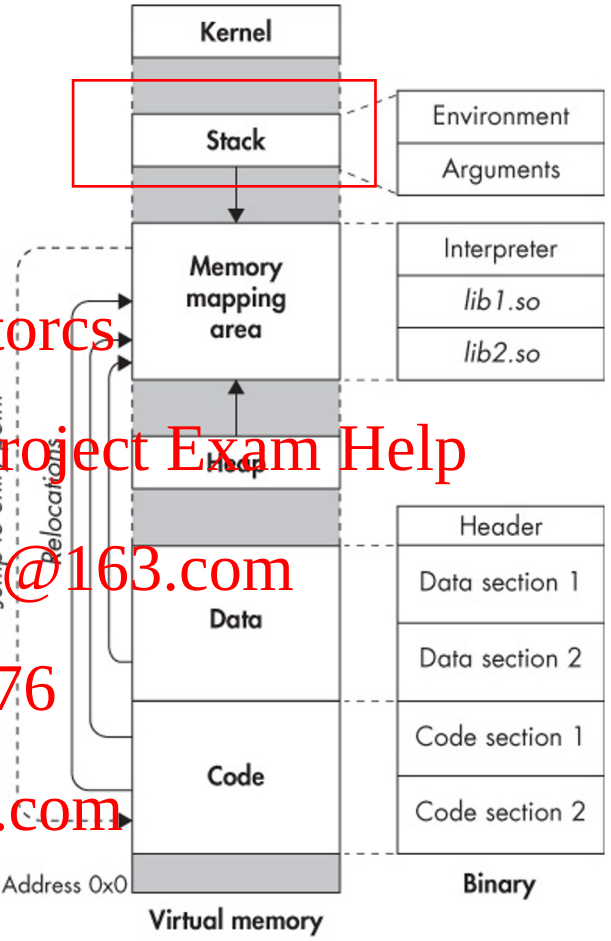
where arr is a symbolic name denoting the displacement.

This could be used, for example, to access an array element, where

- arr denotes the array's starting address,
- rax contains the index of the element,
- and each array element is 4 byte long.

The stack

- The stack is a memory region reserved for storing data related to function calls, such as return addresses, function arguments and local variables.
- It's a last-in-first-out (LIFO) structure, with two operations: push and pop.



The stack

Figure A-3: Pushing the value f *onto the stack and then popping it into* rax

Function calls and function frames

- Each function has an associated *function frame* (also called *stack frame*).
- A stack frame is delimited by values set in registers
  - **rbp** (base pointer) pointing to the base of the frame, and
  - **rsp** (stack pointer) pointing to the top.
- Access to elements in a stack frame is usually specified as an offset relative to rbp.
  - But some compilers may use addressing relative to rsp.

An example

```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
  printf("%s=%s\n",
        argv[1], getenv(argv[1]));

  return 0;
}
```

```
Contents of section .rodata:
 400630 01000200 ❶25733d25 730a00 ....%s=%s..
```

```
Contents of section .text:
 0000000000400566 <main>:
❷ 400566: push   rbp
  400567: mov    rbp,rsp
❸ 40056a: sub    rsp,0x10
❹ 40056e: mov    DWORD PTR [rbp-0x4],edi
  400571: mov    QWORD PTR [rbp-0x10],rsi
  400575: mov    rax,QWORD PTR [rbp-0x10]
  400579: add    rax,0x8
  40057d: mov    rax,QWORD PTR [rax]
❺ 400580: mov    rdi,rax
❻ 400583: call   4004d0 <getenv@plt>
❼ 400588: mov    rdx,rax
  40058b: mov    rax,QWORD PTR [rbp-0x10]
  40058f: add    rax,0x8
  400593: mov    rax,QWORD PTR [rax]
❽ 400596: mov    rsi,rax
  400599: mov    edi,0x400634
  40059e: mov    eax,0x0
❾ 4005a3: call   400440 <printf@plt>
❿ 4005a8: mov    eax,0x0
  4005ad: leave
  4005ae: ret
```

Example of function frames

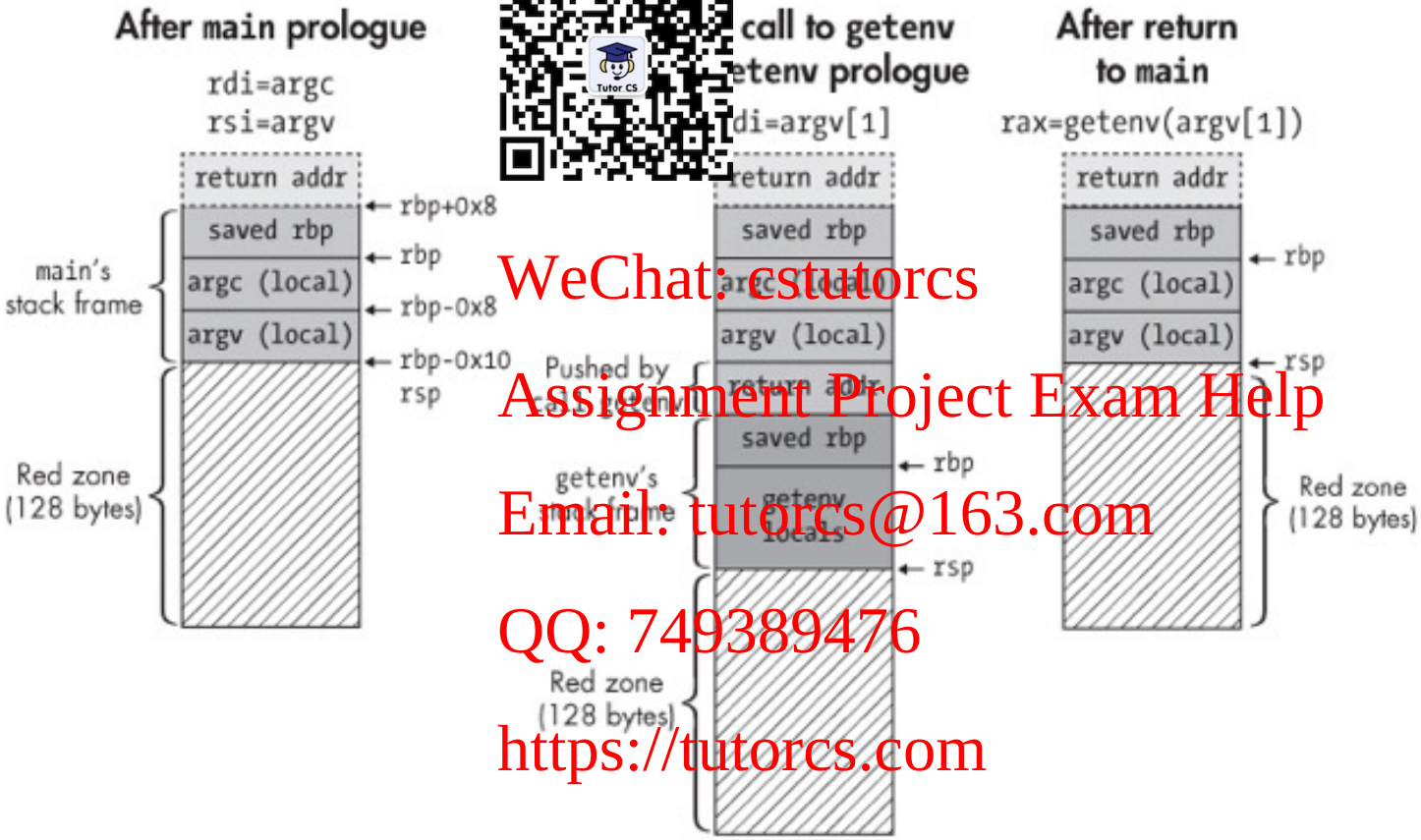


Figure A-4: Example of x86 function frames on a Linux system

Function prologues and local variables

- The first thing a function does is to run a ***prologue*** to set up its function frame.
  - Save the content of rbp register on the stack
  - Copy rsp into rbp.

- After the prologue, space for local variables is reserved by decrementing rsp. For example:

```
sub rsp, 0x10
```

reserves 0x10 (16) bytes for local variable(s).

Example

```
void f()
{
  char buff[16];
  fgets(buff,16,stdin);
  puts(buff);
}
```

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
mov     rdx, QWORD PTR stdin[rip]
lea     rax, [rbp-16]
mov     esi, 16
mov     rdi, rax
call    fgets
lea     rax, [rbp-16]
mov     rdi, rax
call    puts
nop
leave
ret
```

Function calls: preparing arguments

- On x86-64 linux systems, the first six arguments are passed in registers: rdi, rsi, rdx, rcx, r8 and r9, respectively.
- If there are more than 6 arguments, the remaining arguments are pushed to the stack.


```
mov rdi, param1
mov rsi, param2
mov rdx, param3
mov rcx, param4
mov r8, param5
mov r9, param6
push param9
push param8
push param7
```

Function calls: calling a function

- The "call" instructio
used to call a functi
- It automatically pushes the return address (the address of the instruction right after the "call" instruction) onto the stack.

```
push    rbp
mov     rbp, rsp
sub     rsp, 16
mov     rdx, QWORD PTR stdin[rip]
lea     rax, [rbp-16]
mov     esi, 16
mov     rdi, rax
call    fgets
lea     rax, [rbp-16]
mov     rdi, rax
call    puts
nop
leave
ret
```

Function calls: returning from a function

- For linux x86-64 [illegible] convention, return values are stored in registe[illegible]
- To exit a function, an "epilogue" is called to clean up the frame:

```
mov rsp, rbp;  # restore stack pointer
pop rbp;       # restore the base pointer
```

- The instruction "leave" can be used for the same effect; it is a shorthand for "mov rsp, rbp; pop rbp".

Conditional jumps

- Conditional jump branching are implemented using a combination of cmp instruction and one of the conditional jump instructions.
- The "cmp src dst" instruction sets the status flags in the **rflags (eflags)** register based on the outcome of (src-dst).
  - The status can be zero flag (ZF), sign flag (SF), and overflow flag (OF).
- The "jle addr" instruction jumps to address "addr" if src <= dst in the last comparison.

Conditional jumps

```
#include <stdio.h>

int
main(int argc, char *argv[])
{
  if(argc > 5) {
    printf("argc > 5\n");
  } else {
    printf("argc <= 5\n");
  }

  return 0;
}
```

```
0000000000400525 <main>:
400526:  push   rbp
400527:  mov    rbp,rsp
40052a:  sub    rsp,0x10
40052e:  mov    DWORD PTR [rbp-0x4],edi
400531:  mov    QWORD PTR [rbp-0x10],rsi
❸ 400535:  cmp    DWORD PTR [rbp-0x4],0x5
❹ 400539:  jle    400547 <main+0x21>
40053b:  mov    edi,0x4005e4
400540:  call   400400 <puts@plt>
❺ 400545:  jmp    400551 <main+0x2b>
400547:  mov    edi,0x4005ed
40054c:  call   400400 <puts@plt>
400551:  mov    eax,0x0
400556:  leave
400557:  ret
```

Loops

- Loops are specialized forms of conditional branches.
- They can be implemented with cmp/test and conditional jump instructions.

Loops

```
int
main(int argc, char *argv[])
{
  while(argc > 0) {
    printf("%s\n",
          argv[(unsigned)--argc]);
  }

  return 0;
}
```

```
0000000000400526 <main>:
 400526:  push   rbp
 400527:  mov    rbp,rsp
 40052a:  sub    rsp,0x10
 40052e:  mov    DWORD PTR [rbp-0x4],edi
 400531:  mov    QWORD PTR [rbp-0x10],rsi
❶ 400535:  jmp    40055a <main+0x34>
 400537:  sub    DWORD PTR [rbp-0x4],0x1
 40053b:  mov    eax,DWORD PTR [rbp-0x4]
 40053e:  mov    eax,eax
 400540:  lea    rdx,[rax*8+0x0]
 400548:  mov    rax,QWORD PTR [rbp-0x10]
 40054c:  add    rax,rdx
 40054f:  mov    rax,QWORD PTR [rax]
 400552:  mov    rdi,rax
 400555:  call   400400 <puts@plt>
❷ 40055a:  cmp    DWORD PTR [rbp-0x4],0x0
❸ 40055e:  jg     400537 <main+0x11>
 400560:  mov    eax,0x0
 400565:  leave
 400566:  ret
```

Summary

- We have covered basic patterns in assembly code output by a compiler (gcc in this case):
  - memory addressing, function calls, conditionals and loops.
- Understanding patterns in binaries produced by compilers is important for reverse engineering.
  - real-world binaries were most likely produced by compilers.
- Some examples are simplified, in practice compiler optimisations and other features (such as position-independent code) may obscure the logic.
- Some patterns are a result of *calling convention*. For linux, see, e.g., https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf