Machine Language

This lecture is about
- Machine language (in general)
- x86-64 machine language (in particular)
- The assembly and linking processes
- Amusing and important applications to computer security
  (and therefore, Programming Assignment 5, Buffer Overrun)

Instruction Set Architecture (ISA)

There are many kinds of computer chips out there:

Intel x86 series
IBM PowerPC
ARM
RISC-V
MIPS

(and, in the old days, dozens more)

CISC and RISC styles of machine language

<table>
<thead>
<tr>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex, powerful</td>
<td>Simple do-only-one-thing</td>
</tr>
<tr>
<td>instructions</td>
<td>instructions</td>
</tr>
<tr>
<td>Many memory addressing</td>
<td>Few memory addressing</td>
</tr>
<tr>
<td>modes (direct, indirect,</td>
<td>modes (typically only</td>
</tr>
<tr>
<td>base-displacement, indexed,</td>
<td>base+displacement)</td>
</tr>
<tr>
<td>scaled indexed)</td>
<td></td>
</tr>
<tr>
<td>Hardware interpretation is complex</td>
<td>Hardware interpretation is simple</td>
</tr>
<tr>
<td>Need relatively few instructions to accomplish a given job</td>
<td>Need more instructions to accomplish a given job</td>
</tr>
<tr>
<td>Example: x86-64</td>
<td>Examples: ARM, PowerPC</td>
</tr>
</tbody>
</table>

A paradox

```c
enum {BUFSIZE = 48};
char grade = 'D';
char name[BUFSIZE];

/* Read a string into s */
void readString(char *s) {
    char buf[BUFSIZE];
    int i = 0, c;

    /* Read string into buf[] */
    for (;;) {
        c = fgetc(stdin);
        if (c == EOF || c == '\n')
            break;
        buf[i] = c;
        i ++;
    }

    /* Copy buf[] to s[] */
    for (i = 0; i < BUFSIZE; i++)
        s[i] = buf[i];
}

int main(void) {
    printf("What is your name?\n");
    readString(name);
    if (strcmp(name, "Andrew") == 0)
        grade = 'B';
    printf("%c is your grade, %s.\n", grade, name);
    return 0;
}
```

What is your name? Bob
D is your grade, Bob.

What is your name? Andrew
B is your grade, Andrew.

What is your name? [fill in something here]
A is your grade, Susan.
Agenda

x86-64 Machine Language
- Buffer overrun vulnerabilities
- x86-64 Machine Language after Assembly
- x86-64 Machine Language after Linking

Assembly Language: addq %rax, %rbx
Machine Language: 01001000 00000001 11000011

x86-64 Instruction Format
- Difficult to generalize about x86-64 instruction format; many instructions use this format

<table>
<thead>
<tr>
<th>Instruction prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to 4 prefixes of 1 byte each</td>
<td>1, 2, or 3 bytes</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>1, 2, 4, or 8 bytes (if required)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mod R/M</th>
<th>Scale</th>
<th>Index</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mod</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Reg/Opcode</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>R/M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction prefix
- Sometimes a repeat count
- Rarely used; don't be concerned

x86-64 Instruction Format (cont.)

<table>
<thead>
<tr>
<th>Instruction prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to 4 prefixes of 1 byte each</td>
<td>1, 2, or 3 bytes</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>1, 2, 4, or 8 bytes (if required)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mod R/M</th>
<th>Scale</th>
<th>Index</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mod</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Reg/Opcode</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>R/M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ModR/M (register mode, register/opcode, register/memory)
- Specifies types of operands (immediate, register, memory)
- Specifies sizes of operands (byte, word, long)
- Sometimes contains an extension of the opcode

x86-64 Instruction Format (cont.)

<table>
<thead>
<tr>
<th>Extra ModR/M Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000 RAX</td>
</tr>
<tr>
<td>0 001 RCX</td>
</tr>
<tr>
<td>0 010 RDX</td>
</tr>
<tr>
<td>0 011 RBX</td>
</tr>
<tr>
<td>0 100 RSP</td>
</tr>
<tr>
<td>0 101 RBP</td>
</tr>
<tr>
<td>0 110 RSI</td>
</tr>
<tr>
<td>0 111 RDI</td>
</tr>
<tr>
<td>1 000 R8</td>
</tr>
<tr>
<td>1 001 R9</td>
</tr>
<tr>
<td>1 010 R10</td>
</tr>
<tr>
<td>1 011 R11</td>
</tr>
<tr>
<td>1 100 R12</td>
</tr>
<tr>
<td>1 101 R13</td>
</tr>
<tr>
<td>1 110 R14</td>
</tr>
<tr>
<td>1 111 R15</td>
</tr>
</tbody>
</table>

SIB (scale, index, base)
- Used when one of the operands is a memory operand that uses a scale, an index register, and/or a base register

x86-64 Instruction Format (cont.)

Sometimes 3 bits in ModR/M byte, along with extra bit in another field, specify a register
- For 8-byte registers:
  - Similar mappings exist for 4-byte, 2-byte and 1-byte registers

Opcode
- Specifies which operation should be performed
  - Add, move, call, etc.
- Sometimes specifies additional (or less) information

x86-64 Instruction Format (cont.)

SIB (scale, index, base)
- Used when one of the operands is a memory operand that uses a scale, an index register, and/or a base register

SIB (scale, index, base)
- Used when one of the operands is a memory operand that uses a scale, an index register, and/or a base register
x86-64 Instruction Format (cont.)

### Instruction prefixes
- Opcode: 0, 1, or 2 bytes
- ModR/M: 1 byte (if required)
- SIB: 1 byte, 1, 2, or 4 bytes (if required)
- Displacement: 1, 2, 4, or 8 bytes (if required)

### Immediate
- Specifies an immediate operand
- Uses little-endian byte order

### Displacement
- Part of memory operand, or...
- In jump and call instructions, indicates the displacement between the destination instruction and the jump/call instruction
  - More precisely, indicates: \[\text{addr of destination instr} - \text{addr of instr following the jump/call}\]
  - Uses little-endian byte order

---

Example 1

**Assembly lang:**  
```
addq %rax, %rbx
```

**Machine lang:**  
```
4801c3
```

**Explanation:**
- Opcode: This is an add instruction whose src operand is an 8-byte register or memory operand and whose dest operand is an 8-byte register
- ModR/M: The ModR/M byte designates a register
  - ModR/M: The src register is RAX
  - ModR/M: The dest register is RBX

**Observation:** Sometimes opcode specifies operation (e.g. add) and format(s) of operand(s)

---

Example 2

**Assembly lang:**  
```
movl $1, %ebx
```

**Machine lang:**  
```
bb01000000
```

**Explanation:**
- Opcode: This is a mov instruction whose src operand is an immediate
- Opcode: The destination operand is the EBX register
- Immediate: The immediate operand is 1

**Observation:** Sometimes opcode specifies operation and operand(s)

**Observation:** Immediate operands are in little-endian byte order

---

Examples 3, 4

**Assembly lang:**  
```
pushq %rax
```

**Machine lang:**  
```
50
```

**Explanation:**
- Opcode: This is a pushq instruction

**Assembly lang:**  
```
pushq %rcx
```

**Machine lang:**  
```
51
```

**Explanation:**
- Opcode: This is a pushq instruction

**Observation:** Sometimes opcode specifies operation and operand(s)

**Observation:** pushq is used often, so is optimized into 1 byte

---

Example 5

**Assembly lang:**  
```
movl -8(%eax,%ebx,4), %edx
```

**Machine lang:**  
```
678b5498f8
```

**Explanation:**
- Opcode: This is a mov instruction whose src operand is a 4-byte register or memory operand and whose dest operand is a 4-byte register
- ModR/M: The src operand is a register, the dest operand is of the form disp(base,index, scale), the base and index registers are 4-byte registers, and the disp is one-byte
- ModR/M: The dest register is EDX
- SIB: The scale is 4
- SIB: The index register is EBX
- SIB: The base register is EAX
- Displacement: The disp is -8

**Observation:** Two's complement notation

**Observation:** Complicated!!!
Agenda

x86-64 Machine Language
Buffer overrun vulnerabilities
x86-64 Machine Language after Assembly
x86-64 Machine Language after Linking

A program

```c
#include <stdio.h>
int main(int argc, char **argv) {
  char name[12];  int i;
  printf("What is your name?\n");
  for (i=0; i++;
    int c = getchar();
    if (c==\n' | c ==EOF) break;
  name[i] = c;
} name[i]=\0;
printf("Thank you, %s.\n", name);
return 0;
}
```

Why did this program crash?

```c
#include <stdio.h>
int main(int argc, char **argv) {
  char name[12];  int i;
  printf("What is your name?\n");
  for (i=0; i++;
    int c = getchar();
    if (c==\n' | c ==EOF) break;
  name[i] = c;
} name[i]=\0;
printf("Thank you, %s.\n", name);
return 0;
}
```

Stack frame layout

```c
#include <stdio.h>
int main(int argc, char **argv) {
  char name[12];  int i;
  printf("What is your name?\n");
  for (i=0; i++;
    int c = getchar();
    if (c==\n' | c ==EOF) break;
  name[i] = c;
} name[i]=\0;
printf("Thank you, %s.\n", name);
return 0;
}
```

Buffer overrun

```c
#include <stdio.h>
int main(int argc, char **argv) {
  char name[12];  int i;
  printf("What is your name?\n");
  for (i=0; i++;
    int c = getchar();
    if (c==\n' | c ==EOF) break;
  name[i] = c;
} name[i]=\0;
printf("Thank you, %s.\n", name);
return 0;
}
```

Innocuous? buffer overrun

```c
#include <stdio.h>
int main(int argc, char **argv) {
  char name[12];  int i;
  printf("What is your name?\n");
  for (i=0; i++;
    int c = getchar();
    if (c==\n' | c ==EOF) break;
  name[i] = c;
} name[i]=\0;
printf("Thank you, %s.\n", name);
return 0;
}
```
**Buffer overrun**

```
% a.out
What is your name?
abcdefgijklm???
Executable-machine-code...
How may I serve you, master?
%
#include <stdio.h>
int main(int argc, char **argv) {
    char name[12];  int i;
    printf("What is your name?\n");
    for (i=0; ; i++) {
        int c = getchar();
        if (c == EOF || c == 'n') break;
        name[i] = c;
    }
    name[i]=\0;
    printf("Thank you, %s.\n", name);
    return 0;
}
```

**Attacking a web server**

- URLs
- Input in web forms
- Crypto keys for SSL
- etc.

**Attacking a web browser**

- HTML keywords
- Images
- Image names
- URLs
- etc.

**Attacking everything in sight**

- E-mail client
- PDF viewer
- Operating-system kernel
- TCP/IP stack
- Any application that ever sees input directly from the outside

**Defenses against this attack**

**Best:** program in languages that make array-out-of-bounds impossible (Java, C#, ML, python, ...)

If you must program in C: use discipline and software analysis tools in C programming always to check bounds of array subscripts

Otherwise, stopgap security patches:
- Operating system randomizes initial stack pointer
- "No-execute" memory permission
- "Canaries" at end of stack frames

**Your programming assignment:**

```
enum {BUFSIZE = 48};
char grade = 'D';
char name[BUFSIZE];
/* Read a string into s */
void readString(char *s) {
    char buf[BUFSIZE];
    int i = 0;  int c;
    /* Read string into buf[] */
    for (;;) {
        c = fgetc(stdin);
        if (c == EOF || c == 'n') break;
        buf[i] = c;
        i++;
    }
    /* Copy buf[] to s[] */
    for (i = 0; i < BUFSIZE; i++)
        s[i] = buf[i];
}
```
Agenda

x86-64 Machine Language
Buffer overrun vulnerabilities
x86-64 Machine Language after Assembly
x86-64 Machine Language after Linking

An Example Program

A simple (nonsensical) program:

```c
#include <stdio.h>

int main(void)
{
    printf("Type a char: ");
    if (getchar() == 'A')
        printf("Hi\n");
    return 0;
}
```

Let's consider the machine lang equivalent after assembly...

Examining Machine Lang: RODATA

Assemble program; run objdump

```
$ gcc217 -o detecta.s
$ objdump -full-contents --section .rodata detecta.o
```

```
detecta.o:     file format elf64-x86-64
Contents of section .rodata:
0000 54797065 20612063 6861723a 2000
4869 Type a char: .
```

• Assembler does not know addresses
• Assembler knows only offsets
• "Type a char" starts at offset 0
• "Hi\n" starts at offset 0e

Examining Machine Lang: TEXT

Assemble program; run objdump

```
$ gcc217 -o detecta.s
$ objdump -disassemble --reloc detecta.o
```

```
detecta.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
0: b8 00 00 00 00       mov    $0x0,%eax
5: 48 c7 c7 00 00 00 00 mov    $0x0,%rdi
8: R_X86_64_32S .rodata
c: e8 00 00 00 00       callq  11 <main+0x11>
d: R_X86_64_PC32 printf-0x4
e: 83 f8 41             cmp    $0x41,%eax
11: 75 11                jne    2c <skip>
15: b8 00 00 00 00       mov    $0x0,%eax
18: 48 c7 c7 00 00 00 00 mov    $0x0,%rdi
21: R_X86_64_32S .rodata+0xe
c: e8 00 00 00 00       callq  2c <skip>
d: R_X86_64_PC32 printf-0x4
16: e8 00 00 00 00       callq  16 <main+0x16>
12: R_X86_64_PC32 getchar-0x4
10: 83 f8 41             cmp    $0x41,%eax
19: 89 11                jne    2c <skip>
14: b8 00 00 00 00       mov    $0x0,%eax
13: 88 00 00 00 00       mov    $0x0,%edi
20: 48 c7 c7 00 00 00 00 mov    $0x0,%rdi
23: R_X86_64_32S .rodata-0x4
c: e8 00 00 00 00       callq  2c <skip>
d: R_X86_64_PC32 printf-0x4
```

movl $0, %eax

```
$ gcc217 -o detecta.s
$ objdump -disassemble --reloc detecta.o
```

```
Disassembly of section .text:
0000000000000000 <main>:
5: 48 c7 c7 00 00 00 00 mov    $0x0,%rdi
0: b8 00 00 00 00       mov    $0x0,%eax
c: 88 00 00 00 00       mov    $0x0,%edi
d: R_X86_64_32B .rodata
c: e8 00 00 00 00       callq  11 <main+0x11>
d: R_X86_64_PC32 printf-0x4
e: 83 f8 41             cmp    $0x41,%eax
11: 75 11                jne    2c <skip>
15: b8 00 00 00 00       mov    $0x0,%eax
18: 48 c7 c7 00 00 00 00 mov    $0x0,%rdi
21: R_X86_64_32B .rodata-0x4
c: e8 00 00 00 00       callq  2c <skip>
d: R_X86_64_PC32 printf-0x4
```

movl $0, %eax

Assembly lang: movl $0, %eax
Machine lang: b800000000

Explanation:

Opcode: This is a mov instruction whose src operand is a 4-byte immediate

Immediate: The immediate operand is 0

10111000 00000000 00000000 00000000 00000000 00000000
movq $msg1, %rdi

Assembly lang: movq $msg1, %rdi
Machine lang: 48 C7 C7 00 00 00 00

Explanation:

- movq must contain an address
- Assembler knew offset marked by msg1
  - msg1 marks offset 0 relative to beginning of RODATA section
- But assembler did not know address of RODATA section!
- So assembler didn't know address marked by msg1
- So assembler couldn’t generate this instruction completely

Relocation Record 1

8: R_X86_64_32S .rodata

Dear Linker,
Please patch the TEXT section at offset 08. Patch in a 32-bit, Signed value. When you determine the addr of the RODATA section, place that address in the TEXT section at the prescribed place.

Sincerely,
Assembler

call printf

Assembly lang: call printf
Machine lang: e8 00 00 00 00

Explanation:

- call must contain a displacement
- Assembler had to generate the displacement: [addr of printf] − [addr after call instr]
- But assembler didn’t know addr of printf
  - printf isn’t even present yet!
- So assembler couldn’t generate this instruction completely
Relocation Record 2

```
$ gcc217 -c detecta.s
$ objdump -d --reloc detecta.o
detecta.o: file format elf64-x86-64

Disassembly of section .text:
```

```
0000000000000000 <main>:
  0: b8 00 00 00 00       mov    $0x0,%eax
  5: 48 c7 c7 00 00 00 00 mov    $0x0,rdi
  8: R_X86_64_32S         rodata
c: e8 00 00 00 00       callq  11 <main+0x11>
 11: e8 00 00 00 00       callq  12 <main+0x12>
 14: e8 00 00 00 00       callq  17 <main+0x17>
 17: e8 00 00 00 00       callq  22 <main+0x22>

Relocation Record 2
```

```
de: R_X86_64_PC32 printf-0x4
```

```
Dear Linker,
Please patch the TEXT section at offset 0d. Patch in a 32-bit "PC-relative" value. When you determine the addr of printf, compute [addr of printf] – [addr after call] and place the result at the prescribed place.
Sincerely,
Assembler
```

Relocation Record 3

```
call getchar
```

```
$ gcc217 -c detecta.s
$ objdump -d --reloc detecta.o
detecta.o: file format elf64-x86-64

Disassembly of section .text:
```

```
0000000000000000 <main>:
  0: b8 00 00 00 00       mov    $0x0,%eax
  5: 48 c7 c7 00 00 00 00 mov    $0x0,rdi
  8: R_X86_64_32S         rodata
c: e8 00 00 00 00       callq  11 <main+0x11>
 11: e8 00 00 00 00       callq  16 <main+0x16>
 12: R_X86_64_PC32 printf-0x4
 16: e8 00 00 00 00       callq  21 <main+0x21>
 19: e8 00 00 00 00       callq  26 <main+0x26>
 22: e8 00 00 00 00       callq  31 <main+0x31>

Relocation Record 3
```

```
12: R_X86_64_PC32 getchar-0x4
```

```
Dear Linker,
Please patch the TEXT section at offset 12. Do a 32-bit PC-relative patch. When you determine the addr of getchar, compute [offset of getchar] – [addr after call] and place the result at the prescribed place.
Sincerely,
Assembler
```

Relocation Record 2

```
$ gcc217 -c detecta.s
$ objdump -d --reloc detecta.o
detecta.o: file format elf64-x86-64

Disassembly of section .text:
```

```
0000000000000000 <main>:
  0: b8 00 00 00 00       mov    $0x0,%eax
  5: 48 c7 c7 00 00 00 00 mov    $0x0,rdi
  8: R_X86_64_32S         rodata
c: e8 00 00 00 00       callq  11 <main+0x11>
 11: e8 00 00 00 00       callq  16 <main+0x16>
 12: R_X86_64_PC32 printf-0x4
 16: e8 00 00 00 00       callq  21 <main+0x21>
 19: e8 00 00 00 00       callq  26 <main+0x26>
 22: e8 00 00 00 00       callq  31 <main+0x31>

call getchar
```

```
Assembly lang: call getchar
Machine lang: e8 00 00 00 00
Explanation:
```
```
11101000 00000000 00000000 00000000 00000000
Opcode: This is a call instruction with a 4-byte displacement
Disp: The displacement is 00000000 H (0)
```

Relocation Record 3

```
call getchar
```

```
$ gcc217 -c detecta.s
$ objdump -d --reloc detecta.o
detecta.o: file format elf64-x86-64

Disassembly of section .text:
```

```
0000000000000000 <main>:
  0: b8 00 00 00 00       mov    $0x0,%eax
  5: 48 c7 c7 00 00 00 00 mov    $0x0,rdi
  8: R_X86_64_32S         rodata
c: e8 00 00 00 00       callq  11 <main+0x11>
 11: e8 00 00 00 00       callq  16 <main+0x16>
 12: R_X86_64_PC32 printf-0x4
 16: e8 00 00 00 00       callq  21 <main+0x21>
 19: e8 00 00 00 00       callq  26 <main+0x26>
 22: e8 00 00 00 00       callq  31 <main+0x31>

Relocation Record 3
```

```
12: R_X86_64_PC32 getchar-0x4
```

```
Dear Linker,
Please patch the TEXT section at offset 12. Do a 32-bit "PC-relative" value. When you determine the addr of printf, compute [addr of printf] – [addr after call] and place the result at the prescribed place.
Sincerely,
Assembler
```
cmpl $'A', %eax

```
$ gcc217 -c detecta.s
$ objdump --disassemble --reloc detecta.o
detecta.o:     file format elf64-x86-64

Disassembly of section .text:
0000000000000000 <main>:
 0: b8 00 00 00 00       mov    $0x0,%eax
 5: 48 c7 c7 00 00 00 00 mov    $0x0,%rdi
 c: e8 00 00 00 00       callq  11 <main+0x11>
11: e8 00 00 00 00       callq  16 <main+0x16>
16: e8 00 00 00 00       callq  28 <skip>
28: e8 00 00 00 00       callq  2c <skip>

Assembly lang:  cmpl $'A', %eax
Machine lang:  83 f8 41

Explanation:
Opcode: This is an instruction whose source operand is a one-byte immediate and whose destination operand is a register or memory
ModR/M: This is a cmpl instruction, and the last three bytes of the ModR/M field specify the destination register
ModR/M: The dest register is EAX
The immediate operand is 41H ("A")
```

cmpl $'A', %eax

```
$ gcc217 -c detecta.s
$ objdump --disassemble --reloc detecta.o

detecta.o:     file format elf64-x86-64

Disassembly of section .text:
0000000000000000 <main>:
 0: b8 00 00 00 00       mov    $0x0,%eax
 5: 48 c7 c7 00 00 00 00 mov    $0x0,%rdi
 c: e8 00 00 00 00       callq  11 <main+0x11>
11: e8 00 00 00 00       callq  16 <main+0x16>
16: e8 00 00 00 00       callq  12 <char-0x4>
28: e8 00 00 00 00       callq  2c <skip>
2c: b8 00 00 00 00       mov    $0x0,%eax
31: c3                   retq

Assembly lang:  cmpl $'A', %eax
Machine lang:  83 f8 41

Explanation:
Opcode: This is an instruction whose source operand is a one-byte immediate and whose destination operand is a register or memory
ModR/M: This is a cmpl instruction, and the last three bytes of the ModR/M field specify the destination register
ModR/M: The dest register is EAX
The immediate operand is 41H ("A")
```

jne skip

```
$ gcc217 -c detecta.s
$ objdump --disassemble --reloc detecta.o
detecta.o:     file format elf64-x86-64

Disassembly of section .text:
0000000000000000 <main>:
 0: b8 00 00 00 00       mov    $0x0,%eax
 5: 48 c7 c7 00 00 00 00 mov    $0x0,%rdi
 c: e8 00 00 00 00       callq  11 <main+0x11>
11: e8 00 00 00 00       callq  16 <main+0x16>
16: e8 00 00 00 00       callq  28 <skip>
28: e8 00 00 00 00       callq  2c <skip>

Assembly lang:  jne skip
Machine lang:  75 11

Explanation:
Opcode: This is a jne instruction with a one-byte displacement
Disp: The displacement is 11H (17)

• jne must contain a displacement
• Assembler had to generate the displacement:
  [addr of skip] [addr after jne instr]
• Assembler did not know addr of skip
• So assembler could generate this instruction completely
  2cH = 1BH = 11H = 17H
```

movl $0, %eax

```
$ gcc217 -c detecta.s
$ objdump --disassemble --reloc detecta.o
detecta.o:     file format elf64-x86-64

Disassembly of section .text:
0000000000000000 <main>:
 0: b8 00 00 00 00       mov    $0x0,%eax
 5: 48 c7 c7 00 00 00 00 mov    $0x0,%rdi
 c: e8 00 00 00 00       callq  11 <main+0x11>
11: e8 00 00 00 00       callq  16 <main+0x16>
16: e8 00 00 00 00       callq  28 <skip>
28: e8 00 00 00 00       callq  2c <skip>

Assembly lang:  movl $0, %eax
Machine lang:  83 f8 41

Explanation:
Opcode: This is an instruction whose source operand is a one-byte immediate and whose destination operand is a register or memory
ModR/M: This is a movl instruction, and the last three bytes of the ModR/M field specify the destination register
ModR/M: The dest register is EAX
The immediate operand is 0H (0)
```
movl $0, %eax

Assembly lang:  movl $0, %eax
Machine lang:   b800000000
Explanation:

1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Opcode: This is a mov instruction whose src operand is a 4-byte immediate
Opcode: the destination operand is the EAX register
Immediate: The immediate operand is 0

movq $msg2, %rdi

Assembly lang:  movq $msg2, %rdi
Machine lang:   48 C7 C7 00 00 00 00
Explanation:

0 1 0 0 1 0 0 0
Opcode: This is a movq instruction with a 4-byte immediate source operand and a 8 byte register destination operand
Opcode: The destination register is RDI
Disp: The immediate(memory address) is 0

• movq must contain an address
• Assembler knew offset marked by msg2
  • msg2 marks offset 0eH relative to beginning of RODATA section
• But assembler did not know address of RODATA section!
• So assembler didn’t know address marked by msg2
• So assembler couldn’t generate this instruction completely

Relocation Record 4

23: R_X86_64_32S .rodata+0xe

Dear Linker,

Please patch the TEXT section at offset 23H. Patch in a 32-bit Signed value. When you determine the addr of the RODATA section, add 0eH to that address, and place the result in the TEXT section at the prescribed place.

Sincerely,
Assembler
call printf

Assembly lang: call printf
Machine lang: e8 00 00 00 00

Explanations:

- call must contain a displacement
- Assembler must generate the displacement: [addr of printf] - [addr after call instr]
- But assembler didn't know addr of printf
- printf isn't even present yet!
- So assembler couldn't generate this instruction completely

Machine lang:

```
e8 00 00 00 00
```

Explanation:

`11101000 00000000 00000000 00000000 00000000`
 Opcode: This is a call instruction with a 4-byte displacement

`Disp: The displacement is 00000000 (0)`

Relocation Record 5

28: R_X86_64_PC32 printf-0x4

Dear Linker,

Please patch the TEXT section at offset 28. Patch in a 32-bit PC-relative address. When you determine the addr of printf, compute [addr of printf] - [addr after call instr] and place the result at the prescribed place.

Sincerely,
Assembler

movl $0, %eax

Assembly lang: movl $0, %eax
Machine lang: b8 00 00 00 00

Explanations:

10111000 00000000 00000000 00000000 00000000
 Opcode: This is a mov instruction whose source operand is a four-byte immediate and whose destination is EAX

The immediate operand is 0

movl $0, %eax

Machine lang:

```
b8 00 00 00 00
```

Explanation:

11101100 00000000 00000000 00000000 00000000
 Opcode: This is a mov instruction whose source operand is a four-byte immediate and whose destination is EAX

The immediate operand is 0

ret

Machine lang:

```
b8 00 00 00 00
```

Explanation:

11101100 00000000 00000000 00000000 00000000
 Opcode: This is a mov instruction whose source operand is a four-byte immediate and whose destination is EAX

The immediate operand is 0
ret

Assembly lang: ret
Machine lang: c3
Explanation:

11000011
Opcode: This is a ret (alias retq) instruction

Agenda

x86-64 Machine Language
Buffer overrun vulnerabilities
x86-64 Machine Language after Assembly
x86-64 Machine Language after Linking

From Assembler to Linker

Assembler writes its data structures to .o file

Linker:
- Reads .o file
- Writes executable binary file
- Works in two phases: resolution and relocation

Linker Resolution

Resolution
- Linker resolves references

For this program, linker:
- Notes that labels getchar and printf are unresolved
- Fetched machine language code defining getchar and printf from libc.a
- Adds that code to TEXT section
- Adds more code (e.g. definition of _start) to TEXT section too
- Adds code to other sections too

Linker Relocation

Relocation
- Linker patches ("relocates") code
- Linker traverses relocation records, patching code as specified

Examining Machine Lang: RODATA

Link program; run objdump

$ gcc217 detecta.o
$ objdump -full-contents --section .rodata detecta

detecta: file format elf64-x86-64
Contents of section .rodata:
400638 01000200 00000000 00000000 ................
400648 54797065 20612063 6861723a 2000
        4869
Type a char: .

Hi

400658 0a00 ..

(Partial) addresses, not offsets

RODATA is at _00400638_
Starts with some header info
Real start of RODATA is at _00400648_
"Type a char: " starts at _00400648_
"Hi\n" starts at _00400656_
**Examining Machine Lang: TEXT**

```
movq $msg1, %rdi
```

**Additional Code**

```
movq $msg2, %rdi
```

**movq $msg1, %rdi**

Let's examine one line at a time... Addresses, not offsets

**movq $msg2, %rdi**

Recall: Real addr of RODATA = \( \ldots 00400648_\text{h} \)

Linker replaced 00000000\_h with real addr of RODATA + 0

\( \ldots 00400648_\text{h} + 0 = \ldots 00400648_\text{h} \)

= addr denoted by msg1

**movq $msg2, %rdi**

Recall: Real addr of RODATA = \( \ldots 00400648_\text{h} \)

Linker replaced 00000000\_h with [addr of printf] - [addr after call]

\( \ldots 00400648_\text{h} - \ldots 00400525_\text{h} = \_f f f f d 3_\text{h} \)

\( = -301_\text{h} \)

**call printf**

Recall: Real addr of RODATA = \( \ldots 00400648_\text{h} \)

Linker replaced 00000000\_h with [addr of printf] - [addr after call]

\( \ldots 00400648_\text{h} - \ldots 00400525_\text{h} = \_f f f f d 3_\text{h} \)

\( = -301_\text{h} \)

**call getchar**

```
```

**movq $msg2, %rdi**

Recall: Real addr of RODATA = \( \ldots 00400648_\text{h} \)

Linker replaced 00000000\_h with [addr of getchar] - [addr after call]

\( \ldots 00400418_\text{h} - \ldots 00400525_\text{h} = \_f f f f e 0_\text{h} \)

\( = -274_\text{h} \)

**call getchar**

```
```

**call printf**

```
```

**movq $msg2, %rdi**

Recall: Real addr of RODATA = \( \ldots 00400648_\text{h} \)

Linker replaced 00000000\_h with [addr of printf] - [addr after call]

\( \ldots 00400648_\text{h} - \ldots 00400525_\text{h} = \_f f f f e 0_\text{h} \)

\( = -274_\text{h} \)

**movq $msg2, %rdi**

Recall: Real addr of RODATA = \( \ldots 00400648_\text{h} \)

Linker replaced 00000000\_h with [addr of printf] - [addr after call]

\( \ldots 00400648_\text{h} - \ldots 00400525_\text{h} = \_f f f f e 0_\text{h} \)

\( = -274_\text{h} \)

**movq $msg2, %rdi**

Recall: Real addr of RODATA = \( \ldots 00400648_\text{h} \)

Linker replaced 00000000\_h with [addr of printf] - [addr after call]

\( \ldots 00400648_\text{h} - \ldots 00400525_\text{h} = \_f f f f e 0_\text{h} \)

\( = -274_\text{h} \)
$ gcc217 detecta.o -o detecta
$ objdump --disassemble --reloc detecta
detecta: file format elf64-x86-64
... Disassembly of section .text:
... 0000000000400514 <main>:
  400514:       b8 00 00 00 00          mov    $0x0,%eax
  400519:       48 c7 c7 48 06 40 00    mov    $0x400648,%rdi
  400520:       e8 d3 fe ff ff          callq  4003f8 <printf@plt>
  400525:       e8 ee fe ff ff          callq  400418 <getchar@plt>
  40052a:       83 f8 41                cmp    $0x41,%eax
  40052d:       75 11                   jne    400540 <skip>
  40052f:       b8 00 00 00 00          mov    $0x0,%eax
  400534:       48 c7 c7 56 06 40 00    mov    $0x400656,%rdi
  40053b:       callq  4003f8 <printf@plt>
0000000000400540 <skip>:
  400540:       b8 00 00 00 00          mov    $0x0,%eax
  400545:       c3                      retq
... 79

Linker replaced 00000000H with [addr of printf] - [addr after call]
= ...004003f8H - ...00400540H
= ...fffffff8H
= -328H

Summary

x86-64 Machine Language
- CISC: many instructions, complex format
- Fields: prefix, opcode, modR/M, SIB, displacement, immediate

Assembler
- Reads assembly language file
- Generates TEXT, RODATA, DATA, BSS sections
- Containing machine language code
- Generates relocation records
- Writes object (.o) file

Linker
- Reads object (.o) file(s)
- Does resolution: resolves references to make code complete
- Does relocation: traverses relocation records to patch code
- Writes executable binary file