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

Each of these different "machine architectures" understands a different machine language

(and, in the old days, dozens more)

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;
    int c;
    /* Read string into buf[] */
    for (;;) {
        c = fgetc(stdin);
        if (c == EOF || c == '\n')
            break;
        buf[i] = c;
        i ++;
    }
    /* Copy buf[] to s[] */
    buf[i] = '\0';
    for (i = 0; i < BUFSIZE; i++)
        s[i] = buf[i];
}
int main(void) {
    printf("What is your name? \\
    ");
    readString(name);
    if (strcmp(name, "Andrew") == 0)
        grade = 'B';
    printf("%c is your grade, %s. \\
    
", 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.

The Build Process

```
mypgm.c

Preprocess
mypgm.i
mypgm.s

Compile
mypgm.s

Assemble
mypgm.s
libc.a

Link
mypgm
```

Covered in COS 320: Compiling Techniques
Covered here

CISC and RISC styles of machine language

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

CISC and RISC styles of machine language
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

Instruction prefixes
- Up to 4 prefixes of 1 byte each (optional)
  - Modifiable prefix
  - Negated opcode
  - R/M

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

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

Sometimes 3 bits in ModR/M byte, along with extra bit in another field, specify a register
- For 8-byte registers:
  - Modifiable register: 101
  - Negated register: 111

Similar mappings exist for 4-byte, 2-byte and 1-byte registers

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

<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 (optional)</td>
<td>1.2, or 1 byte (optional)</td>
<td>1.2, or 1 byte (optional)</td>
<td>1.2, or 1 byte (optional)</td>
<td>1.2, or 1 byte (optional)</td>
<td>1.2, or 1 byte (optional)</td>
</tr>
</tbody>
</table>

#### 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:**

```
01001000 00000001 11000011
```

- **Opcode:** This is an add instruction whose src operand is an 8-byte register or memory operand and whose dest operand is a 8-byte register  
- **ModR/M:** the M field of 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:**

```
10111011 00000001 00000000 00000000 00000000
```

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

### Example 3, 4

**Assembly lang:** `pushq %rax`  
**Machine lang:** `50`  
**Explanation:**

```
01010000
```

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

**Assembly lang:** `pushq %rcx`  
**Machine lang:** `51`  
**Explanation:**

```
01010001
```

- **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:**

```
01010011 10001011 00001000 00001000 11111000
```

- **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 destination register is EDX  
- **SIB:** The scale is 4  
- **SIB:** The index reg is EAX  
- **SIB:** The base reg 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

```
#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?

```
#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

```
#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

```
#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

```
#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;
}
```
### Attacking a web server

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

**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

Not a single one of these would have prevented the "Heartbleed" attack

---

### Attacking everything in sight

- Email client
- PDF viewer
- Operating system kernel
- TCP/IP stack

**Any** application that ever sees input directly from the outside

---

### Agenda

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

---

### Your programming assignment: Attack the "grader" 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;
}
```

**What is your name?**
- Bob
- Andrew
- Susan

**grade**
- D
- B
- A

---

### Buffer overrun

**a.out**

```
#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;
}
```

---

### Cleverly malicious? Maliciously clever? Buffer overrun

Attacking a web server

- URLs
- Crypto keys for SSL etc.

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

Not a single one of these would have prevented the "Heartbleed" attack
An Example Program

A simple (nonsensical) program:

```
#include <stdio.h>
int main(void)
    int c;
    if (getchar() == 'A')
        return 0;
```

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

Examining Machine Lang: TEXT

```
movl $0, %eax
jne 2c <skip>
```

Let's examine one line at a time...

Examining Machine Lang: RODATA

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

Assembler knows only addresses
- Assembler knows only offsets
- "Type a char" starts at offset 0
- "Hi
n" starts at offset 0e

movl $0, %eax

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

Explanation:
- 0010 0000 0000 0000 0000 0000 0000 0000
- Opcodes: This is a mov instruction whose src operand is a 4-byte immediate
- Opcodes: the destination operand is the EAX register
- Immediate: The immediate operand is 0

movl $0, %eax

movq $msg1, %rdi

Assembly lang:  movq $msg1, %rdi
Machine lang: b800000000

Explanation:
- 00000000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
- Opcodes: This is a movq instruction whose src operand is a 8-byte immediate
- Opcodes: the destination operand is the RDI register
- Immediate: The immediate operand is 0
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 couldn’t generate this instruction completely

Relocation Record 1

8: .R_X86_64_32S .rodata

Dear Liner,

Please patch the TEXT section at offset 0x2c. 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:  48 00 00 00 00
Explanation:

00000000 00000000 00000000 00000000
Opcode: This is a call instruction with a 4-byte displacement
Disp: The displacement is 00000000

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

Relocation Record 1

0: .R_X86_64_32S .rodata

1: 48 C7 C7 00 00 00 00
2: e8 00 00 00 00
3: b8 00 00 00 00
4: callq  16 <main+0x16>
5: callq  11 <main+0x11>
6: mov    $0x0,%eax
7: callq  27 <skip>
8: retq

Disassembly of section .text:

call printf
Relocation Record 2

d: R_X86_64_PC32 printf-0x4

Dear Linker,

Please patch the TEXT section at offset 0x4. 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

12: R_X86_64_PC32 getchar-0x4

Dear Linker,

Please patch the TEXT section at offsets 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

call getchar

Assembly lang: call getchar
Machine lang: 80 00 00 00 00
Explanations:

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

cmpl $'A', %eax
cmpl $'A', %eax

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

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 41 (H)

movl $0, %eax

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

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

Disassembly of section .text:
detecta.o:     file format elf64
$ objdump
$ gcc217

movq $msg2, %rdi

Disassembly of section .text:
detecta.o:     file format elf64
$ objdump
$ gcc217

Disassembly of section .text:
detecta.o:     file format elf64
$ objdump
$ gcc217

Relocation Record 4

23: R_X86_64_32S .rodata+0xe

Dear Linker,

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

Sincerely,
Assembler

Relocation Record 4

23: R_X86_64_32S .rodata+0xe

call printf

Disassembly of section .text:
detecta.o:     file format elf64
$ objdump
$ gcc217

Disassembly of section .text:
detecta.o:     file format elf64
$ objdump
$ gcc217

Disassembly of section .text:
detecta.o:     file format elf64
$ objdump
$ gcc217

call printf

Disassembly of section .text:
detecta.o:     file format elf64
$ objdump
$ gcc217

Disassembly of section .text:
detecta.o:     file format elf64
$ objdump
$ gcc217
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] and place the result at the prescribed place.

Sincerely,
Assembler
Agenda

- x86-64 Machine Language
- Buffer overrun vulnerabilities

Preprocess

- `mypgm.c`
- `mypgm.i`
- `mypgm.s`
- `mypgm.o`

Assemble

- `libc.a`

Link

- `myprog`
### 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