Machine Language

This lecture is about
- machine language (in general)
- AARCH64 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:
- ARM
- Intel x86 series
- IBM PowerPC
- RISC-V
- MIPS
  (and, in the old days, dozens more)

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

The Build Process

- Preprocess
- Compile
- Assemble
- Link

Covered in COS 320: Compiling Techniques

Covered here

Agenda

AARCH64 Machine Language
  - AARCH64 Machine Language after Assembly
  - AARCH64 Machine Language after Linking
  - Buffer overrun vulnerabilities

Assembly Language: \texttt{add x1, x2, x3}
Machine Language: \texttt{1000 1011 0000 0011 0000 0000 0100 0001}

AARCH64 Machine Language

AARCH64 machine language
- All instructions are 32 bits long, 4-byte aligned
- Some bits allocated to opcode: what kind of instruction is this?
- Other bits specify register(s)
- Depending on instruction, other bits may be used for an immediate value, a memory offset, an offset to jump to, etc.

Instruction formats
- Variety of ways different instructions are encoded
- We'll go over quickly in class, to give you a flavor
- Refer to slides as reference for Assignment 5!
  (Every instruction format you'll need is in the following slides... we think...)
AARCH64 Instruction Format

**Operation group**
- Encoded in bits 25-28
  - x101: Data processing – 3-register
  - 100x: Data processing – immediate + register(s)
  - 101x: Branch
  - x1x0: Load/store

**Data processing – 3-register**
- Instruction width in bit 31: 0 = 32-bit, 1 = 64-bit
- Whether to set condition flags (e.g. ADD vs ADDS) in bit 29
- Second source register in bits 16-20
- First source register in bits 5-9
- Destination register in bits 0-4
- Remaining bits encode additional information about instruction

**Example:** `add x1, x2, x3`
- opcode = add
- Instruction width in bit 31: 1 = 64-bit
- Whether to set condition flags in bit 29: no
- Second source register in bits 16-20: 3
- First source register in bits 5-9: 2
- Destination register in bits 0-4: 1
- Additional information about instruction: none

**Data processing – immediate + register(s)**
- Instruction width in bit 31: 0 = 32-bit, 1 = 64-bit
- Whether to set condition flags (e.g. ADD vs ADDS) in bit 29
- Immediate value in bits 10-21 for 2-register instructions, bits 5-20 for 1-register instructions
- Source register in bits 5-9
- Destination register in bits 0-4
- Remaining bits encode additional information about instruction

**Example:** `subs w1, w2, 42`
- opcode: subtract immediate
- Instruction width in bit 31: 0 = 32-bit
- Whether to set condition flags in bit 29: yes
- Immediate value in bits 10-21: 101010 = 42
- First source register in bits 5-9: 2
- Destination register in bits 0-4: 1
- Additional information about instruction: none

**Example:** `mov x1, 42`
- opcode: move immediate
- Instruction width in bit 31: 1 = 64-bit
- Immediate value in bits 5-20: 101010, = 42
- Destination register in bits 0-4: 1
### AARCH64 Instruction Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>Instruction format bytes</td>
</tr>
</tbody>
</table>

#### Branch
- **Relative address of branch target in bits 0-25** for unconditional branch (`b`) and function call (`bl`).
- **Relative address of branch target in bits 5-23** for conditional branch.

#### Load / store
- **Instruction width in bits 30-31**: 00 = 8-bit, 01 = 16-bit, 10 = 32-bit, 11 = 64-bit.
- **For [Xn,Xm] addressing mode**: second source register in bits 16-20.
- **For [Xn,offset] addressing mode**: offset in bits 10-21, shifted left by 3 bits for 64-bit, 2 bits for 32-bit, 1 bit for 16-bit.
- **First source register in bits 5-9**.
- **Destination register in bits 0-4**.
- **Remaining bits encode additional information about instruction**.

---

Example: `b` someLabel
- This depends on where `someLabel` is relative to this instruction!
  - For this example, `someLabel` is 3 instructions (12 bytes) later.
  - **opcode**: unconditional branch.

Example: `bl` someLabel
- This depends on where `someLabel` is relative to this instruction!
  - For this example, `someLabel` is 3 instructions (12 bytes) earlier.
  - **opcode**: branch and link (function call).

Example: `b` someLabel
- This depends on where `someLabel` is relative to this instruction!
  - For this example, `someLabel` is 3 instructions (12 bytes) earlier.
  - **opcode**: conditional branch.
  - **Relative address in bits 5-23**: `11b`. Shift left by 2: `1100b = 12`.

---

Example: `ldr x0, [x1, x2]
- **opcode**: load, register+register.
  - **Instruction width in bits 30-31**: 11 = 64-bit.
  - **Second source register in bits 16-20**: 2.
  - **First source register in bits 5-9**: 1.
  - **Destination register in bits 0-4**: 0.
  - **Additional information about instruction**: no LSL.

---
AARCH64 Instruction Format

Example: `str x0, [sp, 24]`
- opcode: store, register+offset
- Instruction width in bits 30-31: 11 = 64-bit
- Offset value in bits 12-20: 11, shifted left by 3 = 11000₂ = 24
- "Source" (really destination!) register in bits 5-9: 31 = sp
- "Destination" (really source!) register in bits 0-4: 0
- Remember that store instructions use the opposite convention from every other instruction: "source" and "destination" are flipped!

AARCH64 Instruction Format

Example: `strb x0, [sp, 24]`
- opcode: store, register+offset
- Instruction width in bits 30-31: 00 = 8-bit
- Offset value in bits 12-20: 11000₂ = 24
- "Source" (really destination!) register in bits 5-9: 31 = sp
- "Destination" (really source!) register in bits 0-4: 0
- Remember that store instructions use the opposite convention from every other instruction: "source" and "destination" are flipped!

AARCH64 Instruction Format

ADR instruction
- Specifies relative position of label (data location)
- 19 High-order bits of offset in bits 5-23
- 2 Low-order bits of offset in bits 29-30
- Destination register in bits 0-4

AARCH64 Instruction Format

Example: `adr x19, someLabel`
- This depends on where `someLabel` is relative to this instruction!
- For this example, `someLabel` is 50 bytes later
- opcode: generate address
- 19 High-order bits of offset in bits 5-23: 1100
- 2 Low-order bits of offset in bits 29-30: 10
- Relative data location is 110010₁₀ = 50 bytes after this instruction
- Destination register in bits 0-4: 19

Agenda

AARCH64 Machine Language
AARCH64 Machine Language after Assembly
AARCH64 Machine Language after Linking
Buffer overrun vulnerabilities

An Example Program

A simple (nonsensical) program, in C and assembly:

```c
#include <stdio.h>

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

Let's consider the machine language equivalent...
Examine Machine Lang: RDATA

Assemble program; run objdump

Disassemble of section .rodata:

type a char: 

Machine language

Assembler does not know addresses
Assembler knows only offsets
"Hi\n" starts at offset 0

Examine Machine Lang: TEXT

Run objdump to see instructions

Disassemble of section .text:

detecta.o:     file format elf64

Assembly

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

sub sp, sp, #0x10
**sub sp, sp, #0x10**

- **opcode**: subtract immediate
- Instruction width in bit 31: 1 = 64-bit
- Whether to set condition flags in bit 29: no
- Immediate value in bits 10-21: 10000 = 0x10 = 16
- First source register in bits 5-9: sp
- Destination register in bits 0-4: sp
- Additional information about instruction: none

**str x30, [sp]**

- **opcode**: store, register + offset
- Instruction width in bits 30-31: 11 = 64-bit
- Offset value in bits 12-20: 0
- "Source" (really destination) register in bits 5-9: sp
- "Destination" (really source) register in bits 0-4: x30
- Additional information about instruction: none

**adr x0, 0 <main>**

- **opcode**: generate address
- 19 High-order bits of offset in bits 5-23: 0
- 2 Low-order bits of offset in bits 29-30: 0
- Relative data location is 0 bytes after this instruction
- Destination register in bits 0-4: x0
- Additional information about instruction: none
Dear Linker,

Please patch the TEXT section at offset 0x8.
Patch in a 21-bit signed offset of an address, relative to the PC, as appropriate for the instruction format. When you determine the address of .rodata, use that to compute the offset you need to do the patch.

Sincerely,
Assembler

Huh? That's not where printf lives!

• Assembler had to calculate [addr of printf] – [addr of this instr]
• But assembler didn't know address of printf – it's off in some library (libc.a) and isn't present yet!
• So, assembler couldn't generate this instruction completely, left a placeholder, and will request help from the linker

Sincerely,
Assembler
Dear Linker,

Please patch the TEXT section at offset `0x10`.

Patching in a 26-bit signed offset relative to the PC, appropriate for the function call (bl) instruction format. When you determine the address of `getchar`, use that to compute the offset you need to do the patch.

Sincerely,
Assembler

---

Recall that `cmp` is really an assembler alias:

- this is the same instruction as `subs wzr, w0, 0x41`
- `opcode: subtract immediate`
- Instruction width in bit 31: 0 = 32-bit
- Whether to set condition flags in bit 29: yes
- Immediate value in bits 10-21: 1000001 = 'A'
- First source register in bits 5-9: 0
- Destination register in bits 0-4: 31 = wzr

Note that register `1111_1` is used to mean either `sp` or `xzr/wzr`, depending on the instruction.
This instruction is at address 0x18, and skip is at address 0x24, which is 0x24 – 0x18 = 0xc = 12 bytes later.

- opcode: conditional branch
- Relative address in bits 5-23: 11b. Shift left by 2: 1100b = 12
- Conditional branch type in bits 0-4: NE
- No need for relocation record!
- Assembly had to calculate [addr of skip] – [addr of this instr]
- Assembly did know address of skip
- So, assembler could generate this instruction completely, and does not need to request help from the linker.

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

Detecta.o:     file format elf64-littleaarch64
Disassembly of section .text:

```
0000000000000000 <main>:
 0: d10043ff    sub   sp, sp, #0x10
 4: f90003fe    str   x30, [sp]
 8: 10000000    adr   x0, 0 <main>
 8: R_AARCH64_ADR_PREL_LO21    .rodata
 c: 94000000    bl    0 <printf>
 c: R_AARCH64_CALL26     printf
10: 94000000    bl    0 <getchar>
10: R_AARCH64_CALL26    getchar
14: 7101041f    cmp   w0, #0x41
18: 54000061    b.ne  24 <skip>
1c: 10000000    adr   x0, 0 <main>
1c: R_AARCH64_ADR_PREL_LO21   .rodata+0xe
20: 94000000    bl    0 <printf>
20: R_AARCH64_CALL26    printf
0000000000000024 <skip>:
24: 52800000    mov   w0, #0x0
28: f94003fe    ldr   x30, [sp]
2c: 910043ff    add   sp, sp, #0x10
30: d65f03c0    ret
```

Relocation Record 4

```
1c: R_AARCH64_ADR_PREL_LO21    .rodata+0xe
```

Dear Linker,
Please patch the TEXT section at offset 0x1c.
Patch in a 21-bit signed offset of an address, relative to the PC, as appropriate for the instruction format. When you determine the address of .rodata, add 0xe and use that to compute the offset you need to do the patch.

Sincerely,
Assembler

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

Detecta.o:     file format elf64-littleaarch64
Disassembly of section .text:

```
0000000000000000 <main>:
 0: d10043ff    sub   sp, sp, #0x10
 4: f90003fe    str   x30, [sp]
 8: 10000000    adr   x0, 0 <main>
 8: R_AARCH64_ADR_PREL_LO21    .rodata
 c: 94000000    bl    0 <printf>
 c: R_AARCH64_CALL26     printf
10: 94000000    bl    0 <getchar>
10: R_AARCH64_CALL26    getchar
14: 7101041f    cmp   w0, #0x41
18: 54000061    b.ne  24 <skip>
1c: 10000000    adr   x0, 0 <main>
1c: R_AARCH64_ADR_PREL_LO21   .rodata+0xe
20: 94000000    bl    0 <printf>
20: R_AARCH64_CALL26    printf
0000000000000024 <skip>:
24: 52800000    mov   w0, #0x0
28: f94003fe    ldr   x30, [sp]
2c: 910043ff    add   sp, sp, #0x10
30: d65f03c0    ret
```

Another printf, with relocation record...

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

Detecta.o:     file format elf64-littleaarch64
Disassembly of section .text:

```
0000000000000000 <main>:
 0: d10043ff    sub   sp, sp, #0x10
 4: f90003fe    str   x30, [sp]
 8: 10000000    adr   x0, 0 <main>
 8: R_AARCH64_ADR_PREL_LO21    .rodata
 c: 94000000    bl    0 <printf>
 c: R_AARCH64_CALL26     printf
10: 94000000    bl    0 <getchar>
10: R_AARCH64_CALL26    getchar
14: 7101041f    cmp   w0, #0x41
18: 54000061    b.ne  24 <skip>
1c: 10000000    adr   x0, 0 <main>
1c: R_AARCH64_ADR_PREL_LO21   .rodata+0xe
20: 94000000    bl    0 <printf>
20: R_AARCH64_CALL26    printf
0000000000000024 <skip>:
24: 52800000    mov   w0, #0x0
28: f94003fe    ldr   x30, [sp]
2c: 910043ff    add   sp, sp, #0x10
30: d65f03c0    ret
```

mov w0, #0x0

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

Detecta.o:     file format elf64-littleaarch64
Disassembly of section .text:

```
0000000000000000 <main>:
 0: d10043ff    sub   sp, sp, #0x10
 4: f90003fe    str   x30, [sp]
 8: 10000000    adr   x0, 0 <main>
 8: R_AARCH64_ADR_PREL_LO21    .rodata
 c: 94000000    bl    0 <printf>
 c: R_AARCH64_CALL26     printf
10: 94000000    bl    0 <getchar>
10: R_AARCH64_CALL26    getchar
14: 7101041f    cmp   w0, #0x41
18: 54000061    b.ne  24 <skip>
1c: 10000000    adr   x0, 0 <main>
1c: R_AARCH64_ADR_PREL_LO21   .rodata+0xe
20: 94000000    bl    0 <printf>
20: R_AARCH64_CALL26    printf
0000000000000024 <skip>:
24: 52800000    mov   w0, #0x0
28: f94003fe    ldr   x30, [sp]
2c: 910043ff    add   sp, sp, #0x10
30: d65f03c0    ret
```

mov w0, #0x0

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

Detecta.o:     file format elf64-littleaarch64
Disassembly of section .text:

```
0000000000000000 <main>:
 0: d10043ff    sub   sp, sp, #0x10
 4: f90003fe    str   x30, [sp]
 8: 10000000    adr   x0, 0 <main>
 8: R_AARCH64_ADR_PREL_LO21    .rodata
 c: 94000000    bl    0 <printf>
 c: R_AARCH64_CALL26     printf
10: 94000000    bl    0 <getchar>
10: R_AARCH64_CALL26    getchar
14: 7101041f    cmp   w0, #0x41
18: 54000061    b.ne  24 <skip>
1c: 10000000    adr   x0, 0 <main>
1c: R_AARCH64_ADR_PREL_LO21   .rodata+0xe
20: 94000000    bl    0 <printf>
20: R_AARCH64_CALL26    printf
0000000000000024 <skip>:
24: 52800000    mov   w0, #0x0
28: f94003fe    ldr   x30, [sp]
2c: 910043ff    add   sp, sp, #0x10
30: d65f03c0    ret
```

- opcode: move immediate
- Instruction width in bit 31: 0 = 32-bit
- Immediate value in bits 5-20: 0
- Destination register in bits 0-4: 0
Everything Else is Similar...

Exercise for you:
- Using information from these slides, create a bitwise breakdown of these instructions, and convince yourself that the hex values are correct!

Agenda
- AARCH64 Machine Language
- AARCH64 Machine Language after Assembly
- AARCH64 Machine Language after Linking
- Buffer overrun vulnerabilities

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
- Fetches 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

Examining Machine Lang: RODATA

Link program; run objdump

Addresses, not offsets

RODATA is at 0x400720
- Starts with some header info
- Real start of RODATA is at 0x400720
- "Type a char: " starts at 0x400720
- "Hi\n" starts at 0x40072a
• opcode: generate address
• 19 High-order bits of offset in bits 5-23: 110010
• 2 Low-order bits of offset in bits 29-30: 00
• Relative data location is 1101000b = 0xc8 bytes after this instruction
• Destination register in bits 0-4: 0
• msg1 is at 0x400720; this instruction is at 0x400658
• 0x400720 – 0x400658 = 0xc8
• opcode: branch and link
• Relative address in bits 0-25: 26-bit two's complement of 1011111111111111
  But remember to shift left by two bits (see earlier slides)!
This gives −1011111111111111 = −0x17c
  • printf is at 0x4004e0; this instruction is at 0x40065c
  • 0x4004e0 − 0x40065c = −0x17c

1001 01 1111 1111 1111 1111 1111 1111 1111 1111 1010 0001

#include <stdio.h>
int main(void)
{
    char name[12], c;
    int i = 0, magic = 42;
    printf("What is your name?\n");
    while ((c = getchar()) != '\n')
        name[i++] = c;
    name[i] = '\0';
    printf("Thank you, %s.\n", name);
    printf("The answer to life, the universe, and everything is %d\n", magic);
    return 0;
}

Explanation: Stack Frame Layout

When there are too many characters, program carelessly writes beyond space "belonging" to name.
• Overwrites other variables
• This is a buffer overrun, or stack smash
• The program has a security bug!
It Gets Worse…

Buffer overrun can overwrite return address of a previous stack frame!
- Value can be an invalid address, leading to a segfault,…

```
#include <stdio.h>

int main(void)
{
    char name[12], c;
    int i = 0, magic = 42;
    printf("What is your name?\n");
    while ((c = getchar()) != '\n')
        name[i++] = c;
    name[i] = '\0';
    printf("Thank you, %s.\n", name);
    printf("The answer to life, the universe, and everything is %d\n", magic);
    return 0;
}
```

It Gets Much, Much Worse…

Buffer overrun can overwrite return address of a previous stack frame!
- Value can be an invalid address, leading to a segfault, or it can cleverly point to malicious code

```
#include <stdio.h>

int main(void)
{
    char name[12], c;
    int i = 0, magic = 42;
    printf("What is your name?\n");
    while ((c = getchar()) != '\n')
        name[i++] = c;
    name[i] = '\0';
    printf("Thank you, %s.\n", name);
    printf("The answer to life, the universe, and everything is %d\n", magic);
    return 0;
}
```

Attacking a Web Server

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

```
Client PC

Web Server
```

Attacking a Web Browser

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

```
Client PC

Web Server
```

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

```
Client PC

The Internet
```

```
#include <stdio.h>

int main(void)
{
    char name[12], c;
    int i = 0, magic = 42;
    printf("What is your name?\n");
    while ((c = getchar()) != '\n')
        name[i++] = c;
    name[i] = '\0';
    printf("Thank you, %s.\n", name);
    printf("The answer to life, the universe, and everything is %d\n", magic);
    return 0;
}
```
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 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

None of these would have prevented the “Heartbleed” attack

Asgt. 5: Attack the “Grader” Program

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

Summary

AARCH64 Machine Language
- 32-bit instructions
- Formats have conventional locations for opcodes, registers, etc.

Assembler
- Reads assembly language file
- Generates TEXT, RODATA, DATA, BSS sections
- Contains 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