Machine Language
Machine Language

The first part of this lecture (Thursday) covers
- Machine language (in general)
- A motivating example from Assignment 6: Buffer Overrun
- AARCH64 machine language (in particular)

The second part (Tuesday) covers
- The assembly and linking processes
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
  - mypgm.c
  - mypgm.i

- Compile
  - mypgm.i
  - mypgm.s
  - mypgm.o

- Assemble
  - mypgm.o

- Link
  - libc.a

Covered in COS 320: Compiling Techniques

Covered here
Flashback to last lecture ...

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 cause unintended control flow, or even cause arbitrary malicious code to execute!

```c
#include <stdio.h>
int main(void)
{
    char name[12], c;
    int i = 0, magic = 42;
    printf("What is your name?\n");
    while (i = getchar()) i = '\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;
}
```
Assignment 6: Attack the “Grader” Program

/* Prompt for name and read it */
void getName() {
    printf("What is your name?\n");
    readString();
}

/* Read a string into name */
void readString() {
    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++;
    }
    buf[i] = '\0';

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

Unchecked write to buffer!
Initially, the name array in BSS is blank (all 0 bits).

Initially, the buf array in readString has garbage.
Memory Map of Stack and BSS Section

- SP
- readString's stackframe
- buf[0] buf[1] ...
- buf[47] ???
- getName's stackframe
- main's stackframe
- ??? ‘B’ ‘o’ ‘b’ ‘\0’ ??? ...
- ??? ??? ???
- old X30 somewhere
- ...
- name[0] '0'
- name[1] '0'
- ...
- name[47] '0'

(Nothing is copied to BSS until the loop filling buf finishes.) You will put your name + ‘\0’ into the buf array.
Memory Map of Stack and BSS Section

- **readString’s stackframe**
  - SP
  - buf[0]
  - buf[1]
  - ...
  - ???
  - ‘B’
  - ‘o’
  - ‘b’
  - ‘\0’
  - ???
  - ...

- **buf[47]**
  - ???
  - ???
  - ???

- **getName’s stackframe**
  - ???
  - ???

- **main’s stackframe**
  - ???
  - ???

- **old X30 somewhere**
  - ""

- **name[0]**
  - ‘\0’
  - ‘\0’
  - ‘\0’
  - ‘\0’

- **name[1]**
  - ‘\0’
  - ‘\0’
  - ‘\0’
  - ‘\0’

- **name[47]**
  - ‘\0’

*(Nothing is copied to BSS until the loop filling buf finishes.)*

You will put the instructions for your attack here.

What instructions???
Memory Map of Stack and BSS Section

(readString's stackframe)

(readString)

(...)

(getName's stackframe)

(...)

(Nothing is copied to BSS until the loop filling buf finishes.)

How do we write an instruction into memory?
Memory Map of Stack and BSS Section

SP

readString’s
stackframe

buf[0]

buf[1]

...

getName’s
stackframe

main’s
stackframe

Now stack smash like in ‘B’ attack!

Nothing is copied to BSS until the loop filling buf finishes.)

The rest of the instructions to change grade.

getName’s
stackframe

old X30
somewhere

.Replace with BSS address where adr instruction will be put

name[0]  \0

name[1]  \0

...

name[47]  \0

...
Memory Map of Stack and BSS Section

(readString's stackframe) buf[0] buf[1] ...

getName's stackframe

main's stackframe

The address of our adr instruction in BSS (in this example k=4)

(buf[47] ???)

(name[0] ??? 'B' '0' 'b' '\0'
    adr x0, grade other instructions go here then enough padding to smash the stack to fill in: ...
    &name[k]

(name[1] ??? 'B' '0' 'b' '\0'
    adr x0, grade other instructions go here then as much padding as fills name

(name[47])

(Nothing is copied to BSS until the loop filling buf finishes.)
Agenda

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

Assembly Language: add x1, x2, x3

Machine Language: 1000 1011 0000 0011 0000 0000 0100 0001
AARCH64 Machine Language

INSTRUCTION FORMATS

Remember TOY?
ARM is more complex, but the same ideas!

| Format RR: | . . . . . | . . . . | . . . . | . . . . | . . . . | (0–6, A–B) |
| Format A:  | . . . . . | . . . . | . . . . | . . . . | . . . . | (7–9, C–F) |

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
- \textbf{x101}: Data processing – 3-register
- \textbf{100x}: Data processing – immediate + register(s)
- \textbf{101x}: Branch
- \textbf{x1x0}: Load/store
AARCH64 Instruction Format

Op. Group: 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
AARCH64 Instruction Format

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
AARCH64 Instruction Format

Op. Group: 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

msb: bit 31

wxs1 00xx  xxii iiiii iiirr rrrrr rrrrr

Isb: bit 0

wxx1 0010  1xxi iiiii iiiii iiiii iiirr rrrrr
AARCH64 Instruction Format

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: $101010_2 = 42$
- First source register in bits 5-9: 2
- Destination register in bits 0-4: 1
- Additional information about instruction: none
AARCH64 Instruction Format

Example: `mov x1, 42`

- **opcode**: move immediate
- **Instruction width in bit 31**: 1 = 64-bit
- **Immediate value in bits 5-20**: `101010_b` = 42
- **Destination register in bits 0-4**: 1

- 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
- Because all instructions are 32 bits long and are 4-byte aligned, relative addresses end in 00. So, the values in the instruction must be shifted left by 2 bits. This provides more range with fewer bits!
- Type of conditional branch encoded in bits 0-3
Example: `b someLabel`

- This depends on where `someLabel` is relative to this instruction!
  For this example, `someLabel` is 3 instructions (12 bytes) earlier.
- opcode: unconditional branch
- *Relative* address in bits 0-25: two’s complement of $11_b$.
  Shift left by 2: $1100_b = 12$. So, offset is $-12$. 
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)
AARCH64 Instruction Format

Example: `ble someLabel`

- This depends on where `someLabel` is relative to this instruction!
  For this example, `someLabel` is 3 instructions (12 bytes) later
- opcode: conditional branch
- *Relative* address in bits 5-23: `11_b`. Shift left by 2: `1100_b = 12`
- Conditional branch type in bits 0-4: LE
AARCH64 Instruction Format


- 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
AARCH64 Instruction Format

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: $11_b$, shifted left by 3 = $11000_b = 24$
- “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 others: “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**: \(11000_2\) (don’t shift left!) = 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 others: “source” and “destination” are flipped!
AARCH64 Instruction Format

ADR instruction

(Distinct from others w/ Op Group bits 100x)

- 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
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 $110010_2 = 50$ bytes after this instruction
- Destination register in bits 0-4: 19
Agenda

Buffer overrun vulnerabilities

AARCH64 Machine Language

**AARCH64 Machine Language after Assembly**

AARCH64 Machine Language after Linking
Machine Language
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...

```
.section .rodata
msg1: .string "Type a char: 
msg2: .string "Hi\n"
.section .text
.global main
main:
    sub sp, sp, 16
    str x30, [sp]
    adr x0, msg1
    bl printf
    bl getchar
    cmp w0, 'A'
    bne skip
    adr x0, msg2
    bl printf
skip:
    mov w0, 0
    ldr x30, [sp]
    add sp, sp, 16
    ret
```
Examining Machine Lang: RODATA

Assemble program; run objdump

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

detecta.o:     file format elf64-littleaarch64

Contents of section .rodata:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents</th>
<th>Type a char:</th>
<th>Hi\n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>54797065 20612063 6861723a 20004869</td>
<td>.</td>
<td>Hi\n</td>
</tr>
<tr>
<td>0010</td>
<td>0a00</td>
<td></td>
<td>\n</td>
</tr>
</tbody>
</table>

- Assembler does not know addresses
- Assembler knows only offsets
- "Type a char: " starts at offset 0x0
- "Hi\n" starts at offset 0xe
Examining Machine Lang: TEXT

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

detecta.o: file format elf64-littleaarch64

Disassembly of section .text:

```

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>sub sp, sp, #0x10</td>
<td>sub sp, sp, #0x10</td>
</tr>
<tr>
<td>4: f90003fe</td>
<td>str x30, [sp]</td>
<td>str x30, [sp]</td>
</tr>
<tr>
<td>8: 10000000</td>
<td>adr x0, 0 &lt;main&gt;</td>
<td>adr x0, 0 &lt;main&gt;</td>
</tr>
<tr>
<td>c: 94000000</td>
<td>bl 0 &lt;printf&gt;</td>
<td>bl 0 &lt;printf&gt;</td>
</tr>
<tr>
<td>10: 94000000</td>
<td>bl 0 &lt;getchar&gt;</td>
<td>bl 0 &lt;getchar&gt;</td>
</tr>
<tr>
<td>14: 7101041f</td>
<td>cmp w0, #0x41</td>
<td>cmp w0, #0x41</td>
</tr>
<tr>
<td>18: 54000061</td>
<td>b.ne 24 &lt;skip&gt;</td>
<td>b.ne 24 &lt;skip&gt;</td>
</tr>
<tr>
<td>1c: 10000000</td>
<td>adr x0, 0 &lt;main&gt;</td>
<td>adr x0, 0 &lt;main&gt;</td>
</tr>
<tr>
<td>20: 94000000</td>
<td>bl 0 &lt;printf&gt;</td>
<td>bl 0 &lt;printf&gt;</td>
</tr>
</tbody>
</table>

```

0000000000000024 <skip>:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>24: 52800000</td>
<td>mov w0, #0x0</td>
<td>mov w0, #0x0</td>
</tr>
<tr>
<td>28: f94003fe</td>
<td>ldr x30, [sp]</td>
<td>ldr x30, [sp]</td>
</tr>
<tr>
<td>2c: 910043ff</td>
<td>add sp, sp, #0x10</td>
<td>add sp, sp, #0x10</td>
</tr>
<tr>
<td>30: d65f03c0</td>
<td>ret</td>
<td>ret</td>
</tr>
</tbody>
</table>
```

Run `objdump` to see instructions

Assembly language
Examining Machine Lang: TEXT

Run `objdump --disassemble --reloc detecta.o` to see instructions
Let’s examine one line at a time...

Run `objdump` to see instructions

Offsets
$ 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
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: 10000b = 0x10 = 16
• First source register in bits 5-9: 31 = sp
• Destination register in bits 0-4: 31 = sp
• Additional information about instruction: none
detecteda.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: 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: 31 = sp
• “Destination” (really source) register in bits 0-4: 30
• Additional information about instruction: none
adr  x0, 0  <main>

$ 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>
      R_AARCH64_ADR_PREL_LO21   .rodata
 c: 94000000    bl    0 <printf>
     R_AARCH64_CALL26  printf
 10: 94000000    bl    0 <getchar>
     R_AARCH64_CALL26  getchar
 14: 7101041f    cmp   w0, #0x41
 18: 54000061    b.ne  24 <skip>
 1c: 10000000   adr   x0, 0 <main>
     R_AARCH64_ADR_PREL_LO21   .rodata+0xe
 20: 94000000    bl    0 <printf>
     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
\textbf{adr} \ x0, 0 \ <main> \\

- **opcode**: generate address
- **19 High-order bits** of relative address in bits 5-23: 0
- **2 Low-order bits** of relative address in bits 29-30: 0
- **Relative data location** is 0 bytes after this instruction
- **Destination register** in bits 0-4:0

- Huh? That’s not where \texttt{msg1} lives!
  - Assembler knew that \texttt{msg1} is a label within the RODATA section
  - But assembler didn’t know address of RODATA section!
  - So, assembler couldn’t generate this instruction completely, left a placeholder, and will request help from the linker
Examine Machine Lang: TEXT

$ 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
```
$ 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 1

8: R_AARCH64_ADR_PREL_LO21 .rodata

This part is always the same, it’s the name of the machine architecture!

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

* 19 High-order bits of relative address in bits 5-23: 0
2 Low-order bits of relative address in bits 29-30: 0
$ 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: branch and link
• Relative address in bits 0-25: 0

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
$ 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
Dear Linker,

Please patch the TEXT section at offset 0xc. Patch in a 26-bit signed offset relative to the PC, appropriate for the function call (bl) instruction format. When you determine the address of printf, 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
```
bl 0 <getchar>

- opcode: branch and link
- Relative address in bits 0-25: 0

- Same situation as before – relocation record coming up!
Dear Linker,

Please patch the TEXT section at offset 0x10. Patch 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
$ 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

cmp   w0, #0x41
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_{16} = 0x41 = ‘A’`
- **First source register in bits 5-9:** 0
- **Destination register in bits 0-4:** 31 = wzr
  - Note that register #31 (11111_{2}) is used to mean either sp or xzr/wzr, depending on the instruction.
$ 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
b.ne 24 <skip>

- This instruction is at offset 0x18, and skip is at offset 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!
  - Assembler had to calculate [addr of skip] – [addr of this instr]
  - Assembler did know offsets of skip and this instruction
  - 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
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
Another printf, with relocation record...

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

decta.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
```
Last Example: Your Turn!

What does this relocation record mean?

20: 94000000    bl    0 <printf>

20: R_AARCH64_CALL26    printf

See context on previous slides with parallel records: bl printf (#50) bl getchar (#53)

Dear Linker,

Please patch the TEXT section at offset 0x20. Patch in a 26-bit signed offset relative to the PC, appropriate for the function call (bl) instruction format. When you determine the address of printf, use that to compute the offset you need to do the patch.

Sincerely,
Assembler
Everything Else is Similar...

$ 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

000000000000000024 <skip>:
  24: 52800000    mov   w0, #0x0
  28: f94003fe  ldr   x30, [sp]
  2c: 910043ff  add   sp, sp, #0x10
  30: d65f03c0    ret

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

- Buffer overrun vulnerabilities
- AARCH64 Machine Language
- AARCH64 Machine Language after Assembly
- AARCH64 Machine Language after Linking
Assembler writes its data structures to .o file

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

Relocation

- Linker patches (“relocates”) code
- Linker traverses relocation records, patching code as specified
Examining Machine Language: RODATA

Link program; run objdump on final executable

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

detecta:   file format elf64-littleaarch64

Contents of section .rodata:

<table>
<thead>
<tr>
<th>Address (Hex)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>400710</td>
<td>01000200 00000000 00000000 00000000</td>
</tr>
<tr>
<td>400720</td>
<td>54797065 20612063 6861723a 20004869</td>
</tr>
<tr>
<td>400730</td>
<td>0a00</td>
</tr>
</tbody>
</table>

RODATA is at 0x400710
Starts with some header info
Real start of RODATA is at 0x400720
"Type a char: " starts at 0x400720
"Hi\n" starts at 0x40072e

Addresses, not offsets
$ objdump --disassemble --reloc detecta

detecta:   file format elf64-littleaarch64

...  
0000000000400650 <main>:
  0000000000400650: d10043ff    sub   sp, sp, #0x10
  0000000000400654: f90003fe    str   x30, [sp]
  0000000000400658: 10000640    adr   x0, 400720 <msg1>
  000000000040065c: 97ffffa1    bl    4004e0 <printf@plt>
  0000000000400660: 97ffff9c    bl    4004d0 <getchar@plt>
  0000000000400664: 7101041f    cmp   w0, #0x41
  0000000000400668: 54000061    b.ne  400674 <skip>
  000000000040066c: 50000600    adr   x0, 40072e <msg2>
  0000000000400670: 97ffff9c    bl    4004e0 <printf@plt>

0000000000400674 <skip>:
  0000000000400674: 52800000    mov   w0, #0x0
  0000000000400678: f94003fe    ldr   x30, [sp]
  000000000040067c: 910043ff    add   sp, sp, #0x10
  0000000000400680: d65f03c0    ret

Examining Machine Language: TEXT
Addresses, not offsets

Run objdump to see instructions
$ objdump --disassemble --reloc detecta

detecta:  file format elf64-littleaarch64

...  
0000000000400650 <main>:
  400650:   d10043ff    sub   sp, sp, #0x10
  400654:   f90003fe    str   x30, [sp]
  400658:   10000640    adr   x0, 400720 <msg1>
  40065c:   97ffffa1    bl    4004e0 <printf@plt>
  400660:   97ffff9c    bl    4004d0 <getchar@plt>
  400664:   7101041f    cmp   w0, #0x41
  400668:   54000061    b.ne  400674 <skip>
  40066c:   50000600    adr   x0, 40072e <msg2>
  400670:   97ffff9c    bl    4004e0 <printf@plt>

0000000000400674 <skip>:
  400674:   52800000    mov   w0, #0x0
  400678:   f94003fe    ldr   x30, [sp]
  40067c:   910043ff    add   sp, sp, #0x10
  400680:   d65f03c0    ret
Examining Machine Language: TEXT

$ objdump --disassemble --reloc detecta

detecta:   file format elf64-littleaarch64

...  
0000000000400650 <main>:
  400650:   d10043ff    sub   sp, sp, #0x10
  400654:   f90003fe    str   x30, [sp]
  400658:   10000640  adr   x0, 400720 <msg1>
  40065c:   97ffffa1    bl    4004e0 <printf@plt>
  400660:   97ffff9c    bl    4004d0 <getchar@plt>
  400664:   7101041f  cmp   w0, #0x41
  400668:   54000061  b.ne  400674 <skip>
  40066c:   50000600  adr   x0, 40072e <msg2>
  400670:   97ffff9c  bl  4004e0 <printf@plt>

0000000000400674 <skip>:
  400674:   52800000  mov   w0, #0x0
  400678:   f94003fe  ldr   x30, [sp]
  40067c:   910043ff  add   sp, sp, #0x10
  400680:   d65f03c0  ret

Didn't I teach you anything, Linker?

No relocation records!

Let's see what the linker did with them...
$ objdump --disassemble --reloc detecta

detecta: file format elf64-littleaarch64

...  

0000000000000000400650 <main>:  
  400650: d10043ff sub sp, sp, #0x10  
  400654: f90003fe str x30, [sp]  
  400658: 10000640  adr x0, 400720 <msg1>  
  40065c: 97ffffa1 bl 4004e0 <printf@plt>  
  400660: 97ffff9c bl 4004d0 <getchar@plt>  
  400664: 7101041f cmp w0, #0x41  
  400668: 54000061 b.ne 400674 <skip>  
  40066c: 97fffa1 bl 4004e0 <printf@plt>  
  400670: 52800000 mov w0, #0x0  
  400674: f94003fe ldr x30, [sp]  
  400678: 910043ff add sp, sp, #0x10  
  40067c: d65f03c0 ret

adr x0, 400720 <msg1>
**adr x0, 400720 <msg1>**

- **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** 11001000b = 0xc8 bytes after this instruction
- **Destination register in bits 0-4:0**

- **msg1** is at 0x400720; this instruction is at 0x400658
- **0x400720 – 0x400658 = 0xc8**
$ objdump --disassemble --reloc detecta

detecta:    file format elf64-littleaarch64

...  

0000000000400650 <main>:  
  400650:   d10043ff    sub   sp, sp, #0x10  
  400654:   f90003fe    str   x30, [sp]  
  400658:   10000640    adr   x0, 400720 <msg1>  
  40065c:    97ffffa1    bl    4004e0 <printf@plt>  
  400660:    97ffff9c    bl    4004d0 <getchar@plt>  
  400664:    7101041f    cmp   w0, #0x41  
  400668:    54000061    b.ne  400674 <skip>  
  40066c:    50000600    adr   x0, 40072e <msg2>  
  400670:    97ffff9c    bl    4004e0 <printf@plt>  

0000000000400674 <skip>:  
  400674:    52800000    mov   w0, #0x0  
  400678:    f94003fe    ldr   x30, [sp]  
  40067c:    910043ff    add   sp, sp, #0x10  
  400680:    d65f03c0    ret
• **opcode**: branch and link
• *Relative* address in bits 0-25: 26-bit two’s complement of $1011111_2$. But remember to shift left by two bits (see earlier slides)!
  This gives $-10111100_2 = -0x17c$

• `printf` is at 0x4004e0; this instruction is at 0x40065c
• $0x4004e0 - 0x40065c = -0x17c \checkmark$
Everything Else is Similar...

```bash
$ objdump --disassemble --reloc detecta

detecta: file format elf64-littleaarch64

...  
0000000000400650 <main>:
  400650:   d10043ff    sub   sp, sp, #0x10
  400654:   f90003fe    str   x30, [sp]
  400658:   10000640   adr   x0, 400720 <msg1>
  40065c:   97ffffa1    bl   4004e0 <printf@plt>
  400660:   97ffff9c    bl   4004d0 <getchar@plt>
  400664:   7101041f    cmp   w0, #0x41
  400668:   54000061     b.ne 400674 <skip>
  40066c:   52800000    mov   w0, #0x0
  400670:   f94003fe    ldr   x30, [sp]
  400674:   910043ff    add   sp, sp, #0x10
  400678:   d65f03c0    ret
```

Everything Else is Similar...
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
  - 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