Machine Language

Machine Language

This lecture (Monday) is about

- Machine language (in general)
- A motivating example from computer security
  - (And, therefore, Assignment 5: Buffer Overrun)
- AARCH64 machine language (in particular)

These slides also contain

- The assembly and linking processes (Wednesday)

Instruction Set Architecture (ISA)

There are many kinds of computer chips out there:

ARM

Intel x86 series

IBM PowerPC

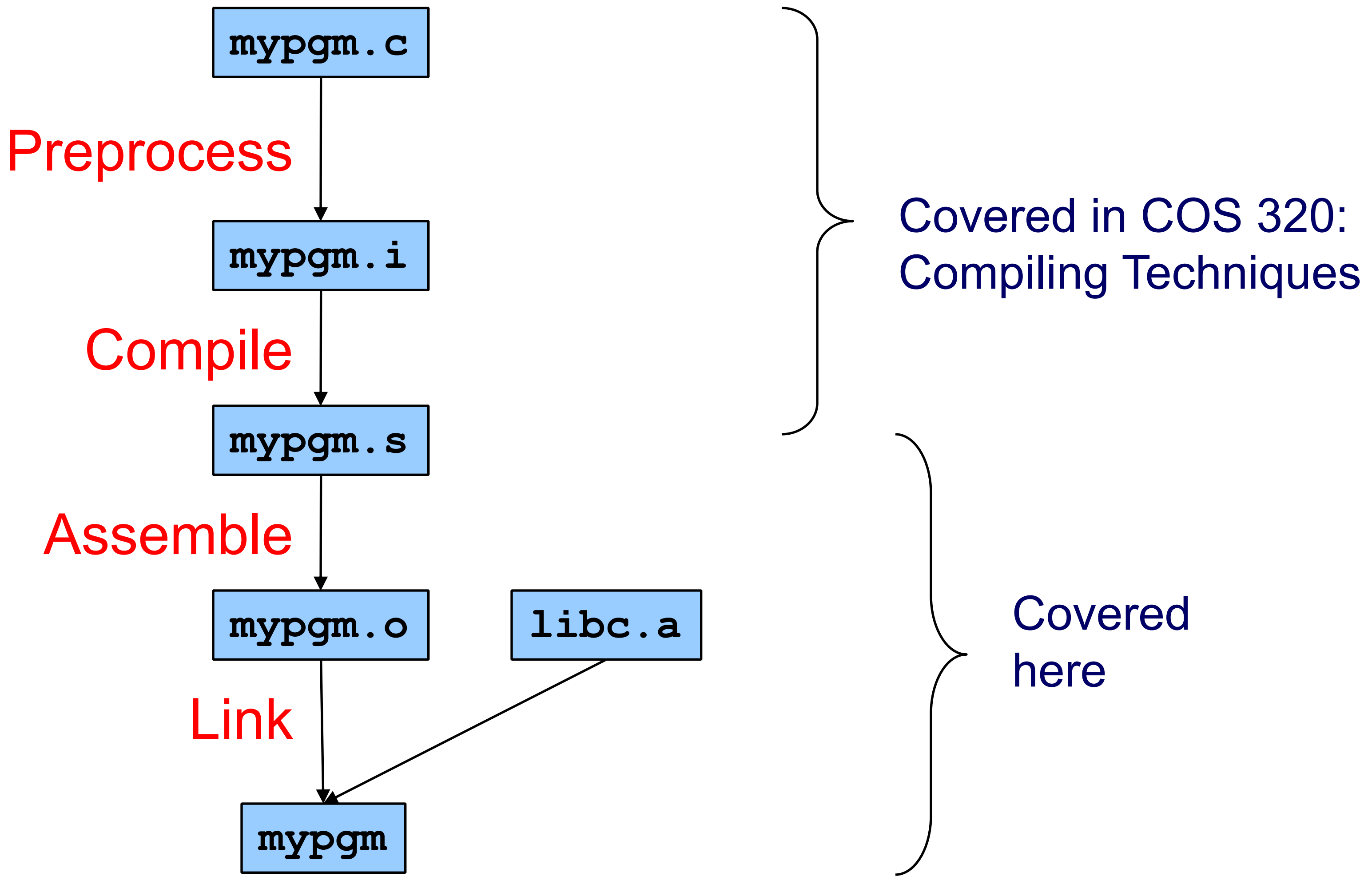
RISC-V

MIPS

(and, in the old days, dozens more)

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

The Build Process



A Program

```c
#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;
}
```

```
$ ./a.out
What is your name?
John Smith
Thank you, John Smith.
The answer to life, the universe, and everything is 42
```

Why People With Long Names Have Prob

```
#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;
}
```

```
$ ./a.out
What is your name?
Christopher Moretti
Thank you, Christopher Mor
tti.
The answer to life, the universe, and everything is 6911092
```

?

??? (!)

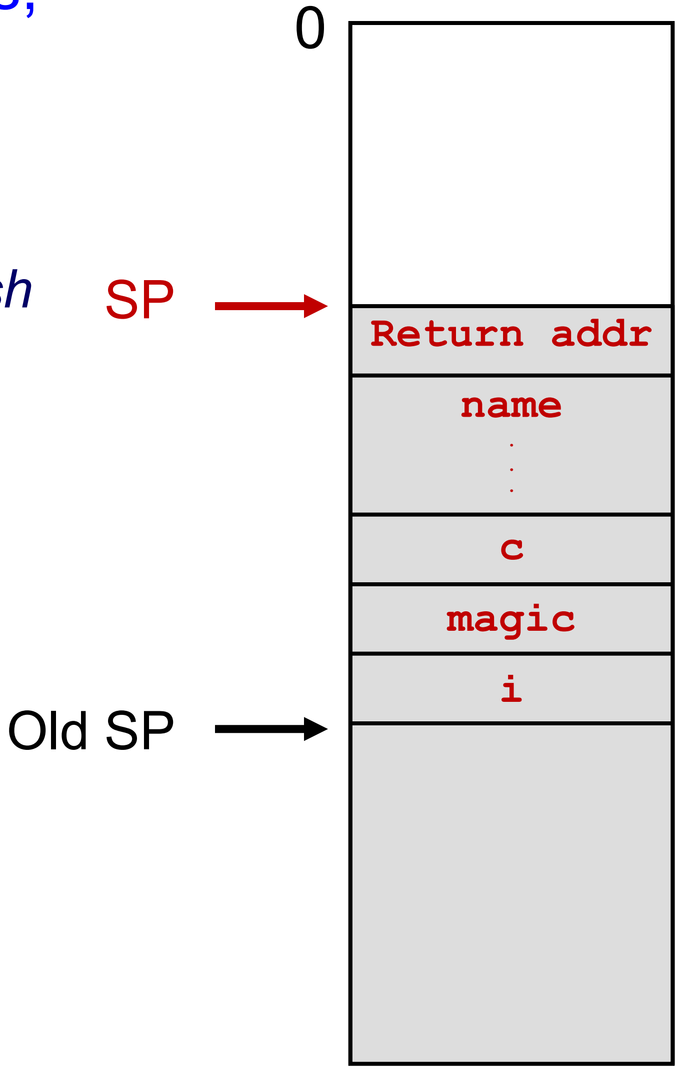
(depending on the area code, this might be an interesting phone number, but probably not one you should call for the answer to life, the universe, and everything)

Explanation: Stack Frame Layout

When there are too many characters, program carelessly writes beyond space "belonging" to name.

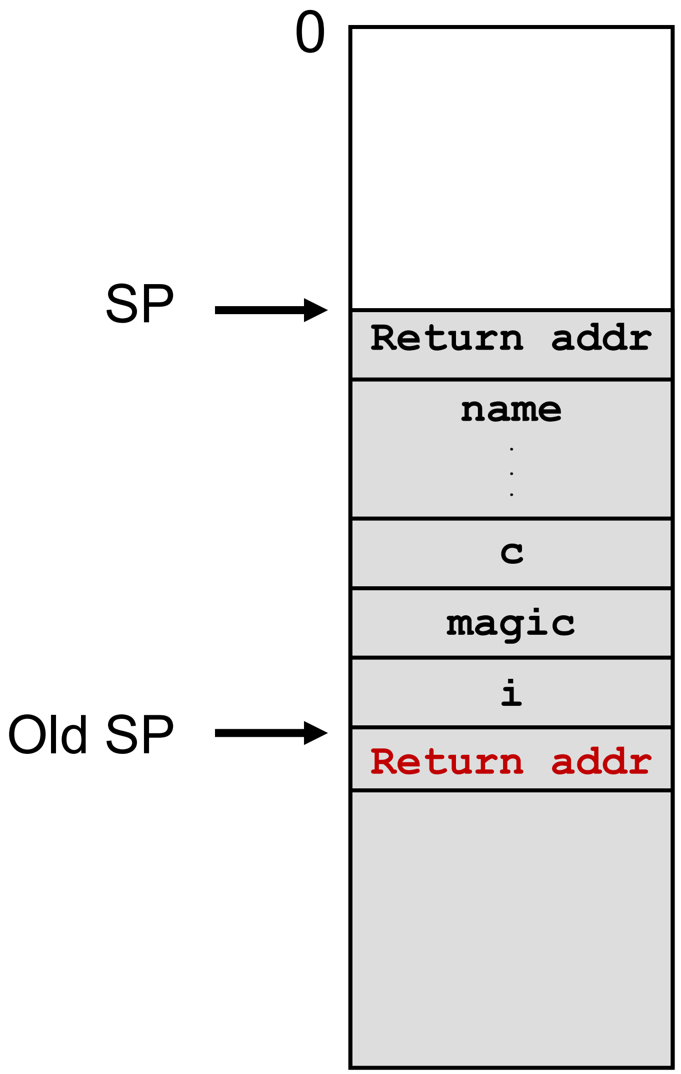
- Overwrites other variables
- This is a *buffer overrun*, or *stack smash*
- The program has a security bug!

```
#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 Worse…

Buffer overrun can overwrite return address of a previous stack frame!



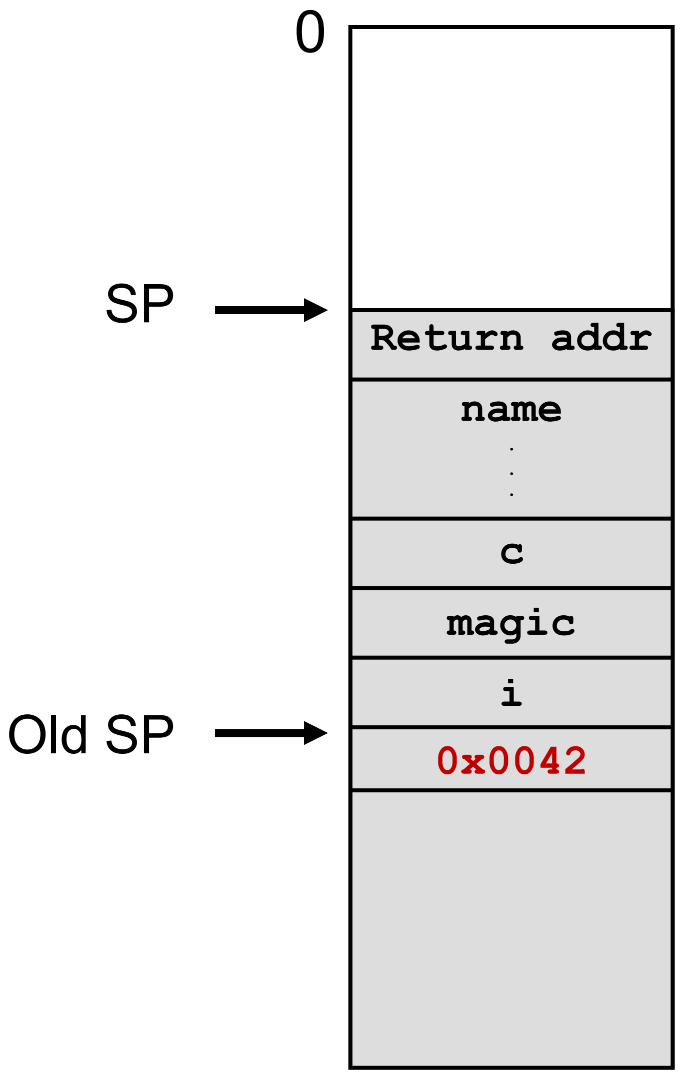
```
#include <stdio.h>
int main(void)
{
   char name[12], c;
   int i = 0, magic = 42;
   printf("What is your name?\n");
   while ((c = getchar()) != '\n')
      name[i++] = c;
   name[i] = '\0';
   printf("Thank you, %s.\n", name);
   printf("The answer to life, the universe, "
          "and everything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

It Gets Worse…

Buffer overrun can overwrite return address of a previous stack frame!

- Value can be an invalid address, leading to a segfault,…

```c
#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;
}
```

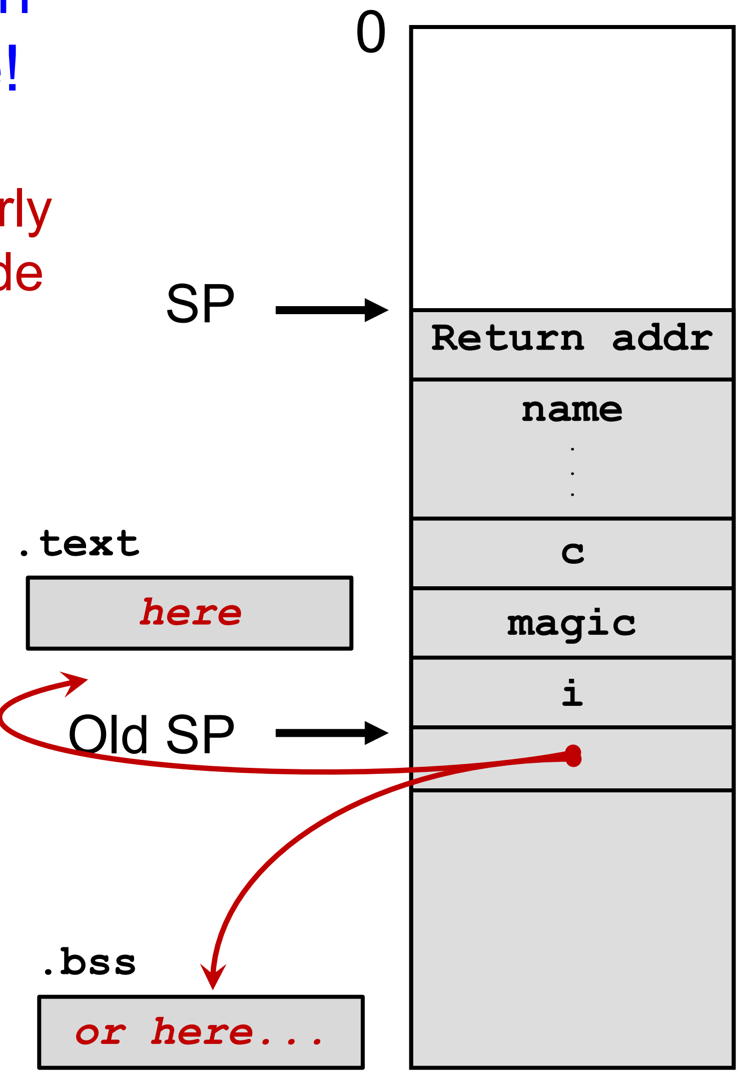


It Gets Much, Much Worse…

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 unintended or 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;
}
```



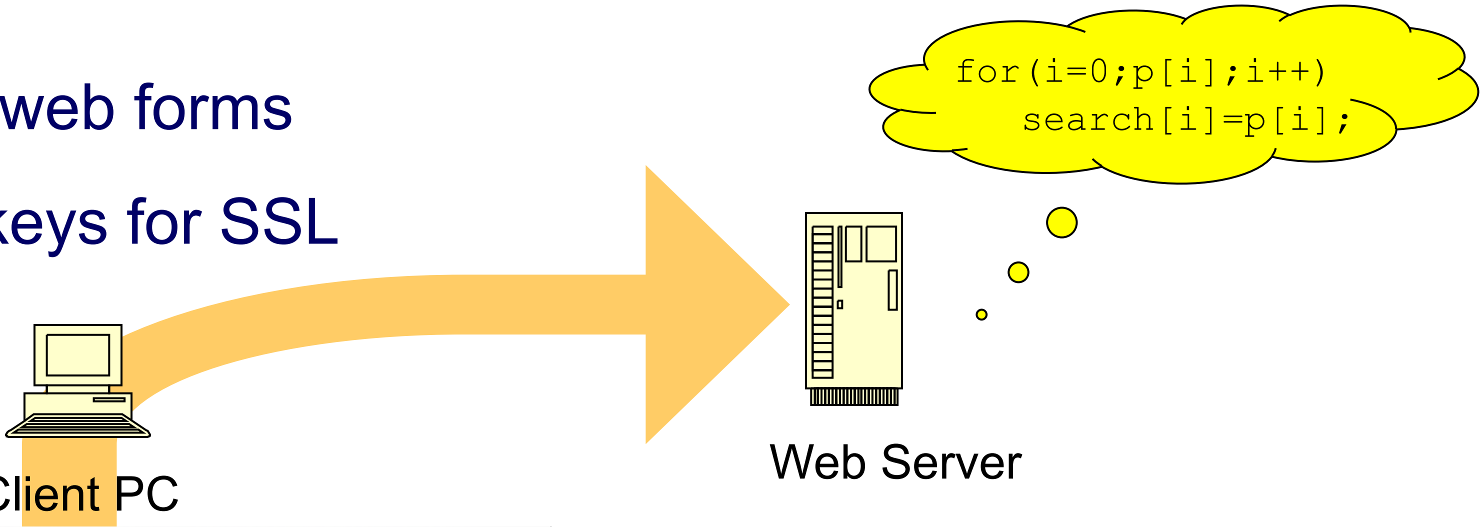
Attacking a Web Server

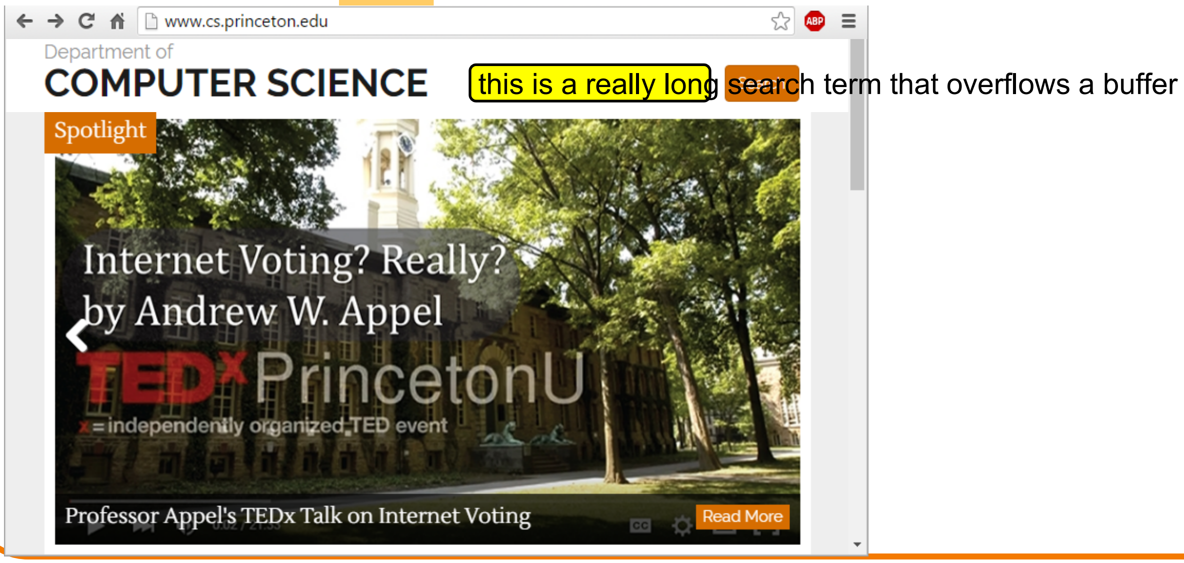
URLs

Input in web forms

Crypto keys for SSL

etc.

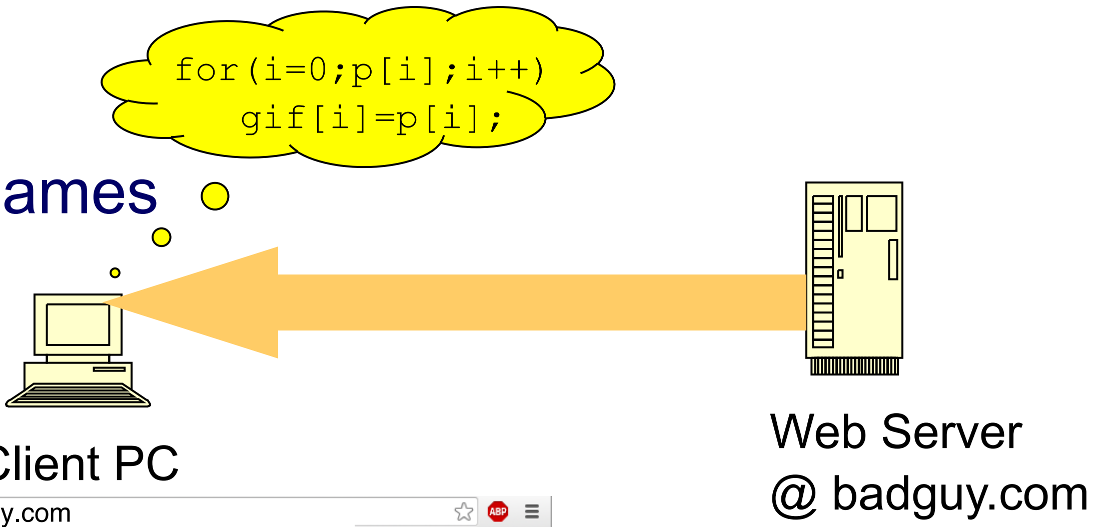




www.cs.princeton.edu
Department of
COMPUTER SCIENCE
this is a really long search term that overflows a buffer
Spotlight
Internet Voting? Really? by Andrew W. Appel
TEDx PrincetonU
x = independently organized TED event
Professor Appel's TEDx Talk on Internet Voting
Read More

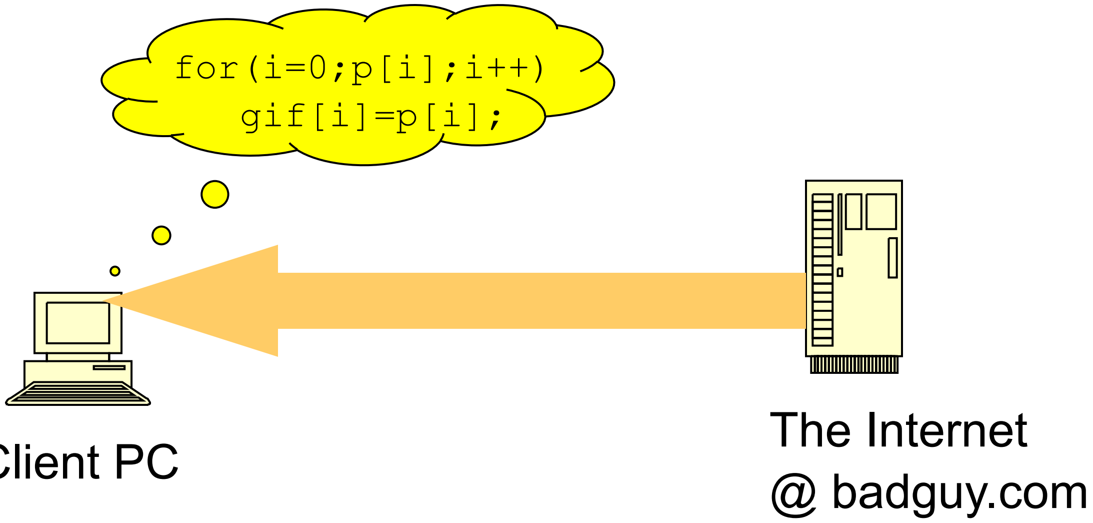
Attacking a Web Browser

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



Earn $$$ Thousands
working at home!

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, ...)

None of these would have prevented the "Heartbleed" attack

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

Asgt. 5: Attack the "Grader" Program

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

```
$ ./grader
What is your name?
Bob
D is your grade.
Thank you, Bob.
$ ./grader
What is your name?
Andrew Appel
B is your grade.
Thank you, Andrew Appel.
```

Asgt. 5: Attack the “Grader” Program

```
/* 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];
}
```

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

Unchecked write to buffer!

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;
}
```

```
$ ./grader
What is your name?
Bob\0(#@&$%*#&(*^!@%*!!(&#$%(@*
B is your grade.
Thank you, Bob.
```

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;
}
```

```
$ ./grader
What is your name?
Susan\0?!*!????*???!*!%!?!(!*%(*^^?
A is your grade.
Thank you, Susan.
```

Agenda

AARCH64 Machine Language

AARCH64 Machine Language after Assembly

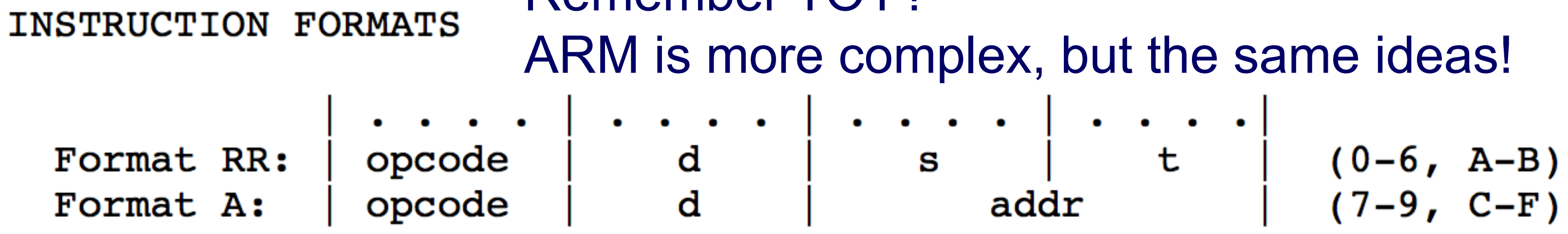
AARCH64 Machine Language after Linking

Buffer overrun vulnerabilities

Assembly Language: **`add x1, x2, x3`**

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

AARCH64 Machine Language



```
INSTRUCTION FORMATS

             | . . . . | . . . . | . . . . | . . . .|
  Format RR: | opcode  |    d    |    s    |    t    |  (0-6, A-B)
  Format A:  | opcode  |    d    |       addr        |  (7-9, C-F)
```

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

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

```
msb: bit 31                                                 lsb: bit 0
↓                                                                    ↓
xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
```

Operation group

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

AARCH64 Instruction Format

```
msb: bit 31                                          lsb: bit 0
↓                                                             ↓
wxsx 101x xxxr rrrr xxxx xxrr rrrr rrrr
```

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

```
msb: bit 31                                              lsb: bit 0
↓                                                                 ↓
1000 1011 0000 0011 0000 0000 0100 0001
```

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

```
msb: bit 31                                      lsb: bit 0
↓                                                         ↓
wxs1 00xx xxii iiii iiii iirr rrrr rrrr
wxx1 0010 1xxi iiii iiii iiii iiir rrrr
```

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

AARCH64 Instruction Format

```
msb: bit 31                                          lsb: bit 0
↓                                                            ↓
0111 0001 0000 0000 1010 1000 0100 0001
```

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_b = 42
- First source register in bits 5-9: 2
- Destination register in bits 0-4: 1
- Additional information about instruction: none

AARCH64 Instruction Format

msb: bit 31 ↓ lsb: bit 0 ↓

```
1101 0010 1000 0000 0000 0101 0100 0001
```

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

AARCH64 Instruction Format

msb: bit 31 lsb: bit 0

```
xxx1 01ii iiii iiii iiii iiii iiii iiii
xxx1 01xx iiii iiii iiii iiii iiix cccc
```

Op. Group: 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
- 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

AARCH64 Instruction Format

msb: bit 31 ↓ lsb: bit 0 ↓

```
0001 0111 1111 1111 1111 1111 1111 1101
```

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.

AARCH64 Instruction Format

msb: bit 31 ↓ … lsb: bit 0 ↓

```
1001 0111 1111 1111 1111 1111 1111 1101
```

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)
- *Relative* address in bits 0-25: two's complement of 11_b. Shift left by 2: 1100_b = 12. So, offset is -12.

AARCH64 Instruction Format

msb: bit 31 ↓ lsb: bit 0 ↓

```
0101 0100 0000 0000 0000 0000 0110 1101
```

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

```
msb: bit 31                                         lsb: bit 0
↓                                                            ↓
wwxx 1x0x xxxr rrrr xxxx xxrr rrrr rrrr
wwxx 1x0x xxii iiii iiii iirr rrrr rrrr
```

Op. Group: 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

AARCH64 Instruction Format

msb: bit 31 ↓ lsb: bit 0 ↓

```
1111 1000 0110 0010 0110 1000 0010 0000
```

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

msb: bit 31 ↓ lsb: bit 0 ↓

```
1111 1001 0000 0000 0000 1111 1110 0000
```

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 every other instruction: “source” and “destination” are flipped!

AARCH64 Instruction Format

msb: bit 31 ↓ lsb: bit 0 ↓

```
0011 1001 0000 0000 0110 0011 1110 0000
```

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_b (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 every other instruction: "source" and "destination" are flipped!

AARCH64 Instruction Format

msb: bit 31 ↓ lsb: bit 0 ↓

```
0ii1 0000 iiii iiii iiii iiii iiir rrrr
```

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

AARCH64 Instruction Format

msb: bit 31 ↓ lsb: bit 0 ↓

```
0101 0000 0000 0000 0000 0001 1001 0011
```

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_b = 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

An Example Program

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

```
#include <stdio.h>
int main(void)
{  printf("Type a char: ");
   if (getchar() == 'A')
      printf("Hi\n");
   return 0;
}
```

```
       .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
```

Let's consider the machine language equivalent…

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:
 0000 54797065 20612063 6861723a 20004869  Type a char: .Hi
 0010 0a00                                 ..
```

Offsets

Contents

- 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:

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
```

Run objdump to see instructions

Assembly language

Examining 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
```

Run objdump to see instructions

Machine language

Examining 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
```

Run objdump to see instructions

Offsets

Let's examine one line at a time…

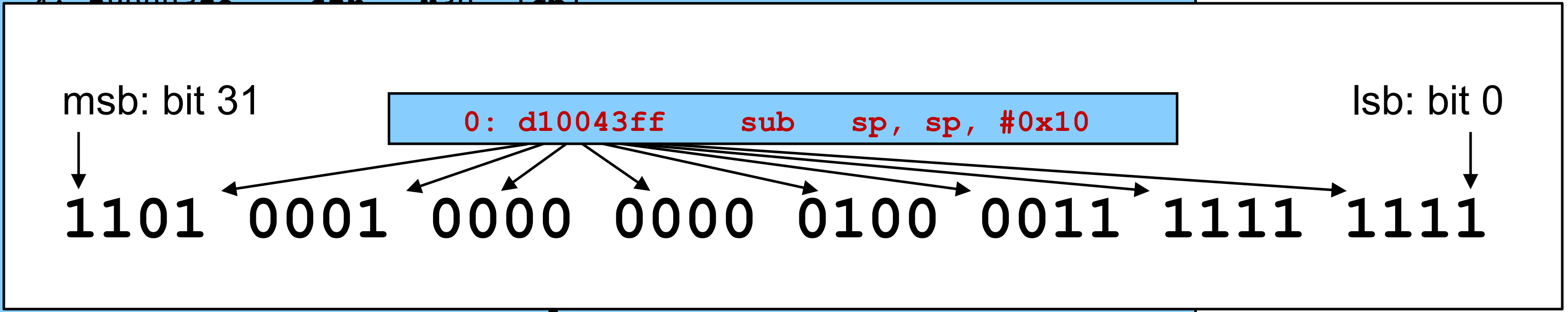
sub sp, sp, #0x10

```
$ 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]
```



```
  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`

msb: bit 31

```
0: d10043ff    sub    sp, sp, #0x10
```

lsb: bit 0

```
1101 0001 0000 0000 0100 0011 1111 1111
```

- 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_b = 0x10 = 16
- First source register in bits 5-9: 31 = sp
- Destination register in bits 0-4: 31 = sp
- Additional information about instruction: none

str x30, [sp]

```
$ 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
```

`str x30, [sp]`

msb: bit 31

```
4: f90003fe    str   x30, [sp]
```

lsb: bit 0

```
1111 1001 0000 0000 0000 0011 1111 1110
```

- 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>
                   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
```

`adr x0, 0 <main>`

msb: bit 31

```
8: 10000000     adr    x0, 0 <main>
```

lsb: bit 0

```
0001 0000 0000 0000 0000 0000 0000 0000
```

- 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 **`msg1`** lives!
  - Assembler knew that **`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

Examining Machine Lang: TEXT

Run objdump to see instructions

```
$ 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 records

R_AARCH64_ADR_PREL_LO21 .rodata

```
$ 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
```

$\underbrace{\texttt{R\_AARCH64\_}}_{\text{This part is always the same, it's the name of the machine architecture!}}$

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

bl   0 <printf>

```
$ 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 <printf>`

msb: bit 31 ↓

```
c:  94000000     bl      0 <printf>
```

lsb: bit 0 ↓

```
1001 0100 0000 0000 0000 0000 0000 0000
```

- 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

R_AARCH64_CALL26 printf

```
$ 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 2

```
c: R_AARCH64_CALL26     printf
```

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

bl 0 <getchar>

```
$ 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>`

msb: bit 31

```
10:  94000000     bl      0 <getchar>
```

lsb: bit 0

```
1001 0100 0000 0000 0000 0000 0000 0000
```

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

- Same situation as before – relocation record coming up!

Relocation Record 3

```
10: R_AARCH64_CALL26    getchar
```

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

cmp w0, #0x41

```
$ 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`

msb: bit 31

```
14: 7101041f     cmp    w0, #0x41
```

lsb: bit 0

```
0111 0001 0000 0001 0000 0100 0001 1111
```

- 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_b = 0x41 = ‘A’
- First source register in bits 5-9: 0
- Destination register in bits 0-4: 31 = wzr
- Note that register #31 (11111_b) is used to mean either sp or xzr/wzr, depending on the instruction

b.ne 24 <skip>

```
$ 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>`

msb: bit 31 ↓

```
18:  54000061     b.ne   24 <skip>
```

lsb: bit 0 ↓

```
0101 0100 0000 0000 0000 0000 0110 0001
```

- 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: 11_b. Shift left by 2: 1100_b = 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

R_AARCH64_ADR_PREL_LO21 .rodata+0xe

```
$ 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

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
```

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

0000000000000024 <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

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

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

detecta:     file format elf64-littleaarch64

Contents of section .rodata:
 400710 01000200 00000000 00000000 00000000  ................
 400720 54797065 20612063 6861723a 20004869  Type a char: .Hi
 400730 0a00                                 ..
```

Addresses, not offsets

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**

Examining Machine Lang: 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
```

Run objdump to see instructions

Addresses, not offsets

Examining Machine Lang: 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
```

Additional code

Examining Machine Lang: 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
```

No relocation records!

Let’s see what the linker did with them…

adr   x0, 400720 <msg1>

```
$ 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
```

adr x0, 400720 <msg1>

msb: bit 31

```
400658:    10000640     adr    x0, 400720 <msg1>
```

lsb: bit 0

```
0001 0000 0000 0000 0000 0110 0100 0000
```

- 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 ✓

bl   4004e0 <printf@plt>

```
$ 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
```

`bl      4004e0 <printf@plt>`

msb: bit 31

```
40065c:    97ffffa1    bl    4004e0 <printf@plt>
```

lsb: bit 0

```
1001 011 1111 1111 1111 1111 1010 0001
```

- opcode: branch and link
- *Relative* address in bits 0-25: 26-bit two's complement of 1011111_b . But remember to shift left by two bits (see earlier slides)! This gives $-1\ 0111\ 1100_b$ = –0x17c

- **`printf`** is at 0x4004e0; this instruction is at 0x40065c
- 0x4004e0 – 0x40065c = –0x17c ✓

Everything Else is Similar…

```
$ 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
```

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