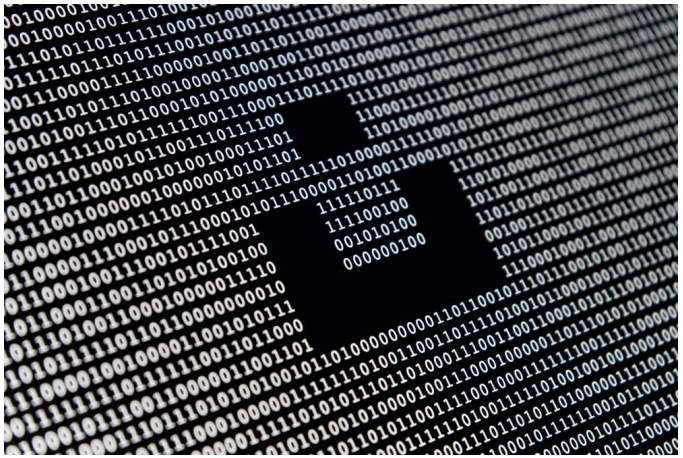
COS 217: Introduction to Programming Systems

Machine Language



PRINCETON UNIVERSITY

Instruction Set Architecture (ISA)

There are many kinds of computer chips out there:

ARM (AARCH64)

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* – binary encoding of instructions

@swimstaralex (previous slide)

Machine Language

Today we'll cover:

- A motivating example from Assignment 6: Buffer Overrun
- The AARCH64 machine language
- So that you can build your attack

Next time (our last lecture …) we'll cover:

- The assembly and linking processes

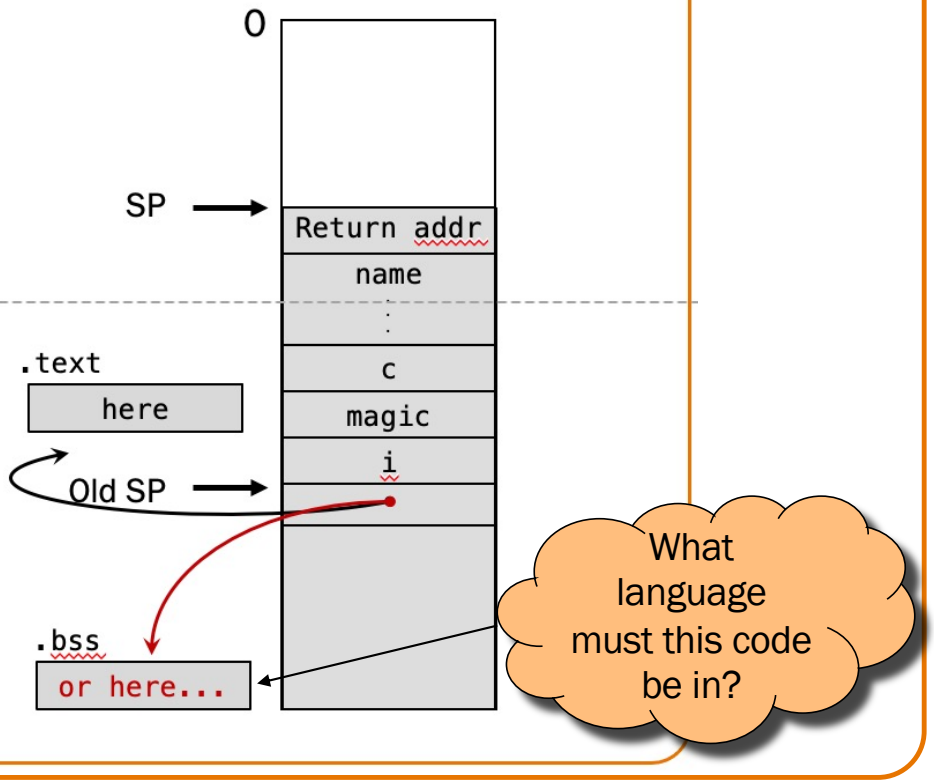
Flashback to last lecture …

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!

```
#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;
}
```



Assignment 6: The “Grader” Program

```
enum {BUFSIZE = 48};
char grade = 'D';
char name[BUFSIZE];
...
int main(void)
{
   mprotect(...);
   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?
Jaswinder Singh
D is your grade.
Thank you, Jaswinder Singh.
$ ./grader
What is your name?
Andrew Appel
B is your grade.
Thank you, Andrew Appel.
```

Assignment 6: Attack the "Grader" Program

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

Unchecked write to buffer

```
/* 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];
}
```

Opportunity to inject instructions into persistent memory, so they stay after readstring() exits. If we overwrite return address on stack with their start address , can return to execute them

Attacking the Grader Program

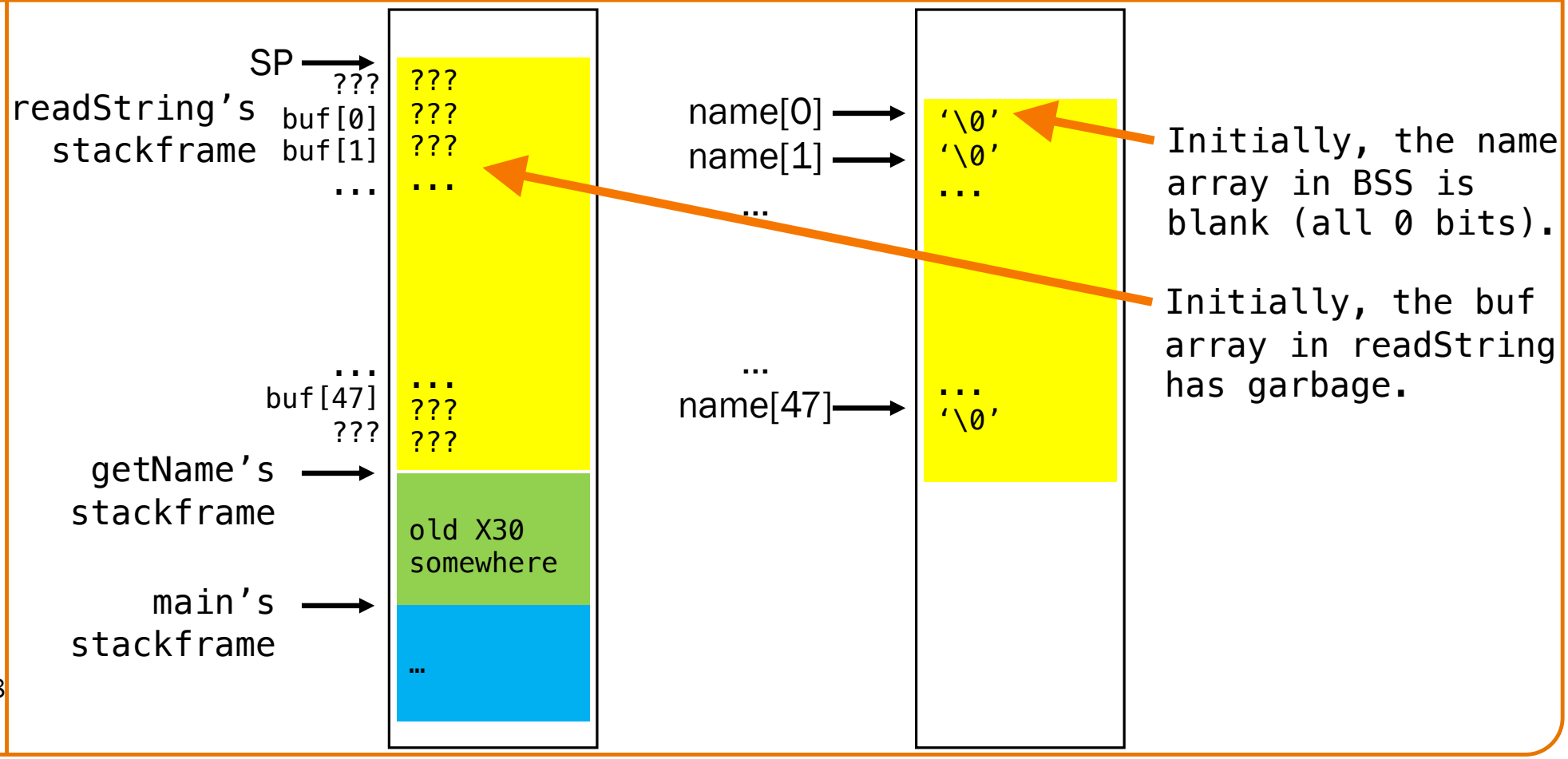
No attack case

- User enters name, followed by newline
- First loop stops reading at newline, adds null character to terminate output properly, and all is well

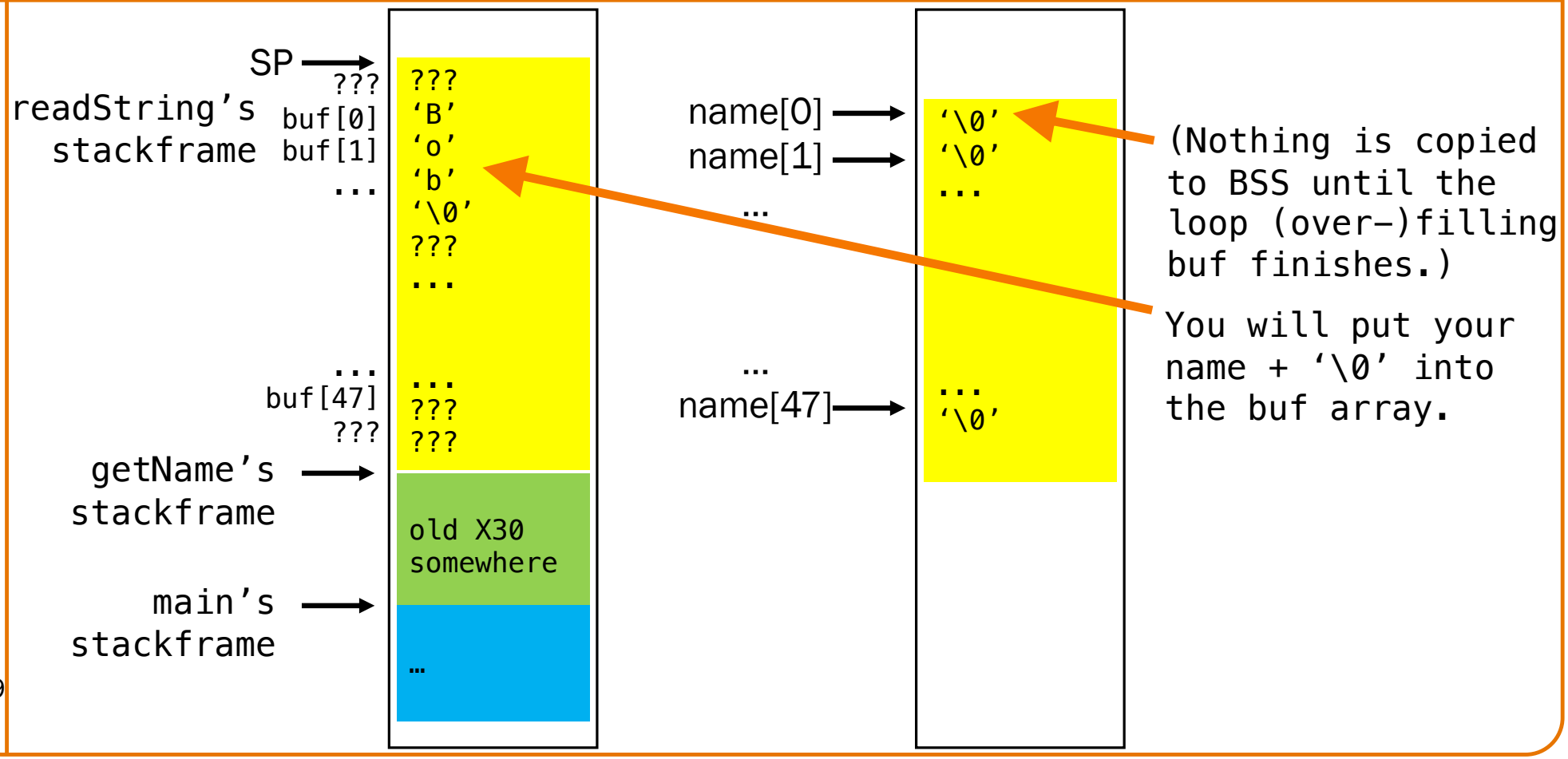
Attack case. User enters:

- Name, but no newline after it, so reading loop doesn't terminate there
- Instead, a null byte after it, so output is written as just name and not corrupted
- Then, filler until reaches x30 (return address)
- Then, address of first malicious instruction
  - Could be in text section, for 'B' attack
  - An array element of name[], for 'A' attack

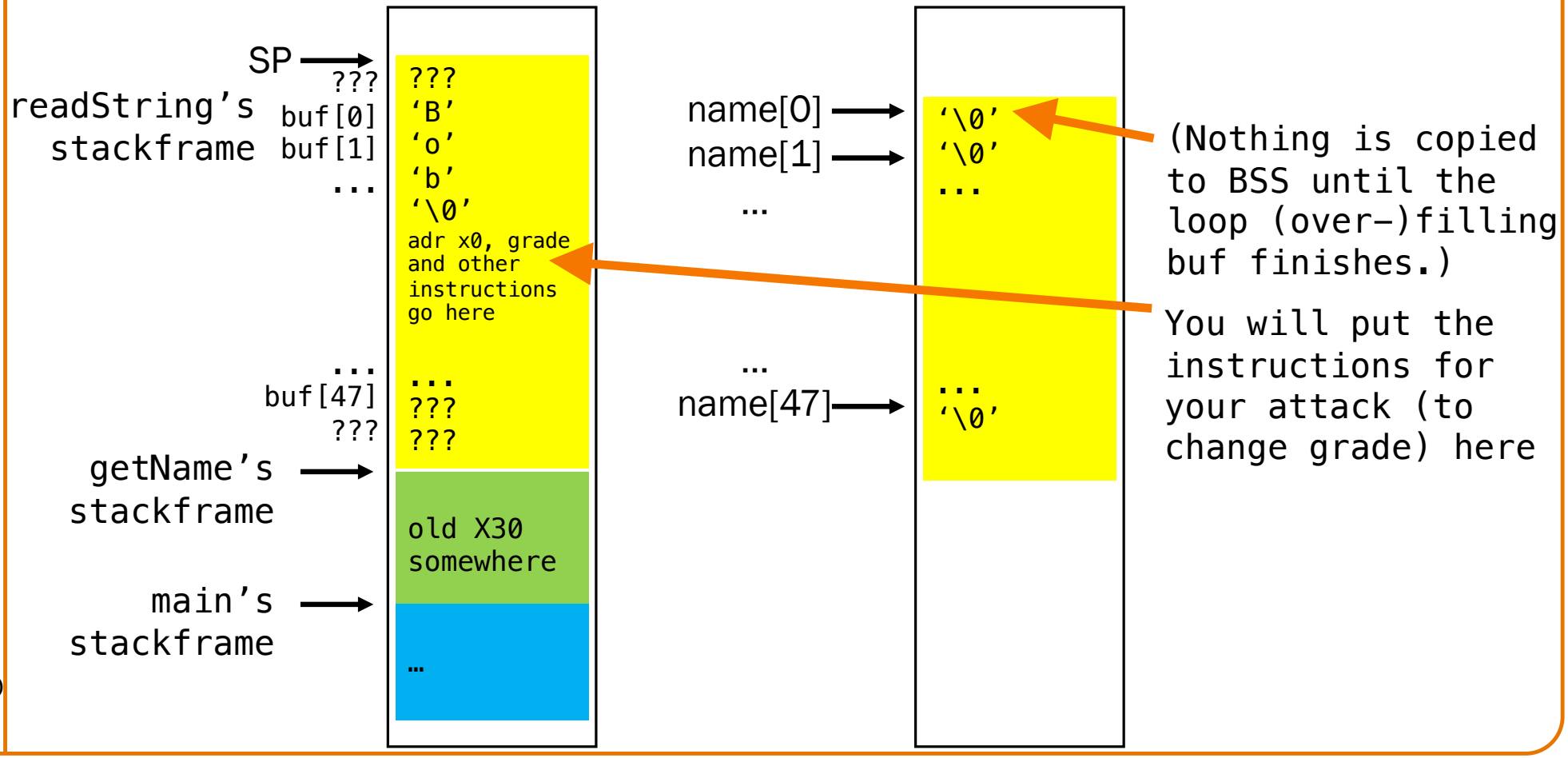
Stack and BSS Sections: Entering `readstring()`



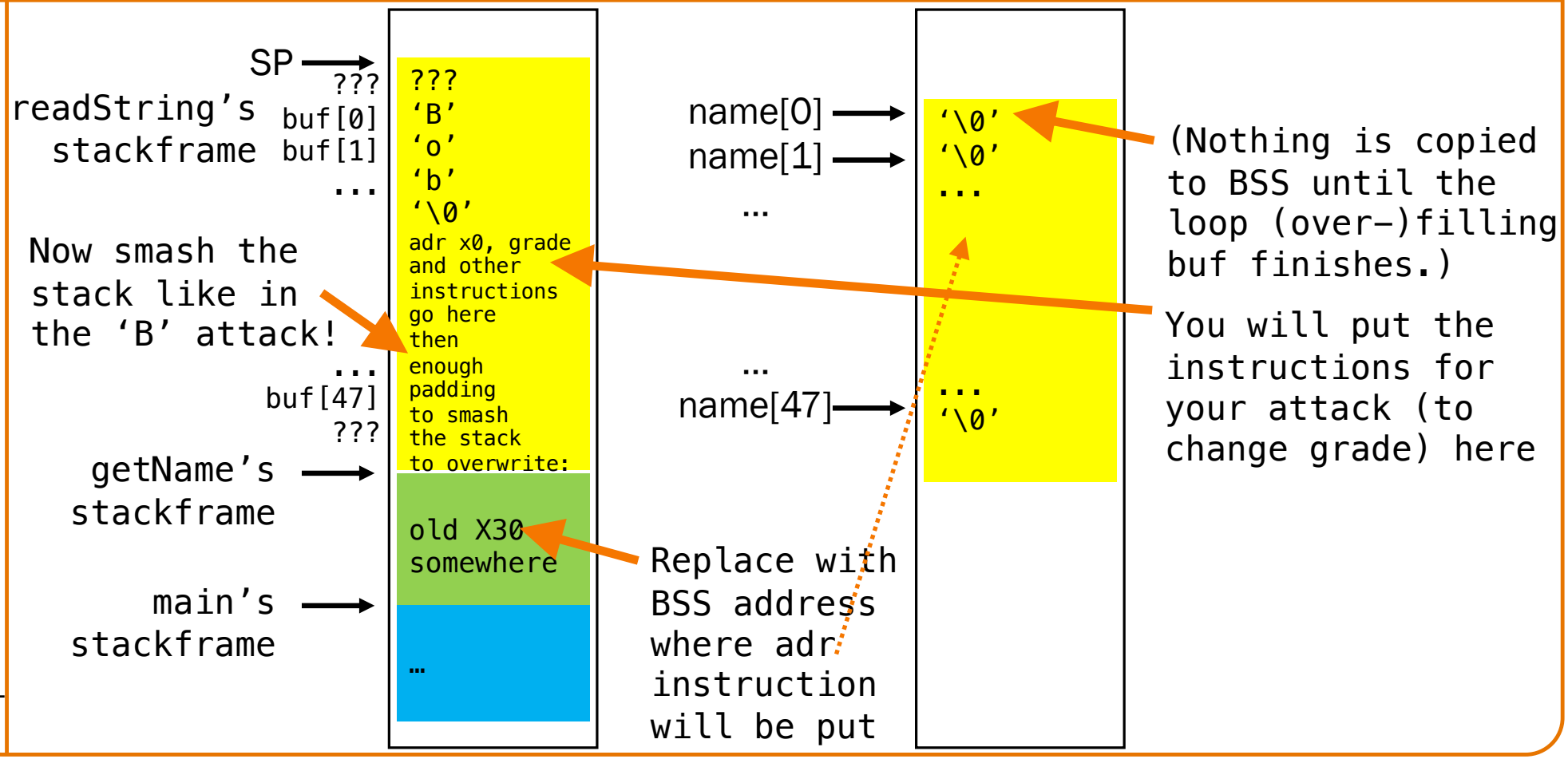
Stack and BSS Sections



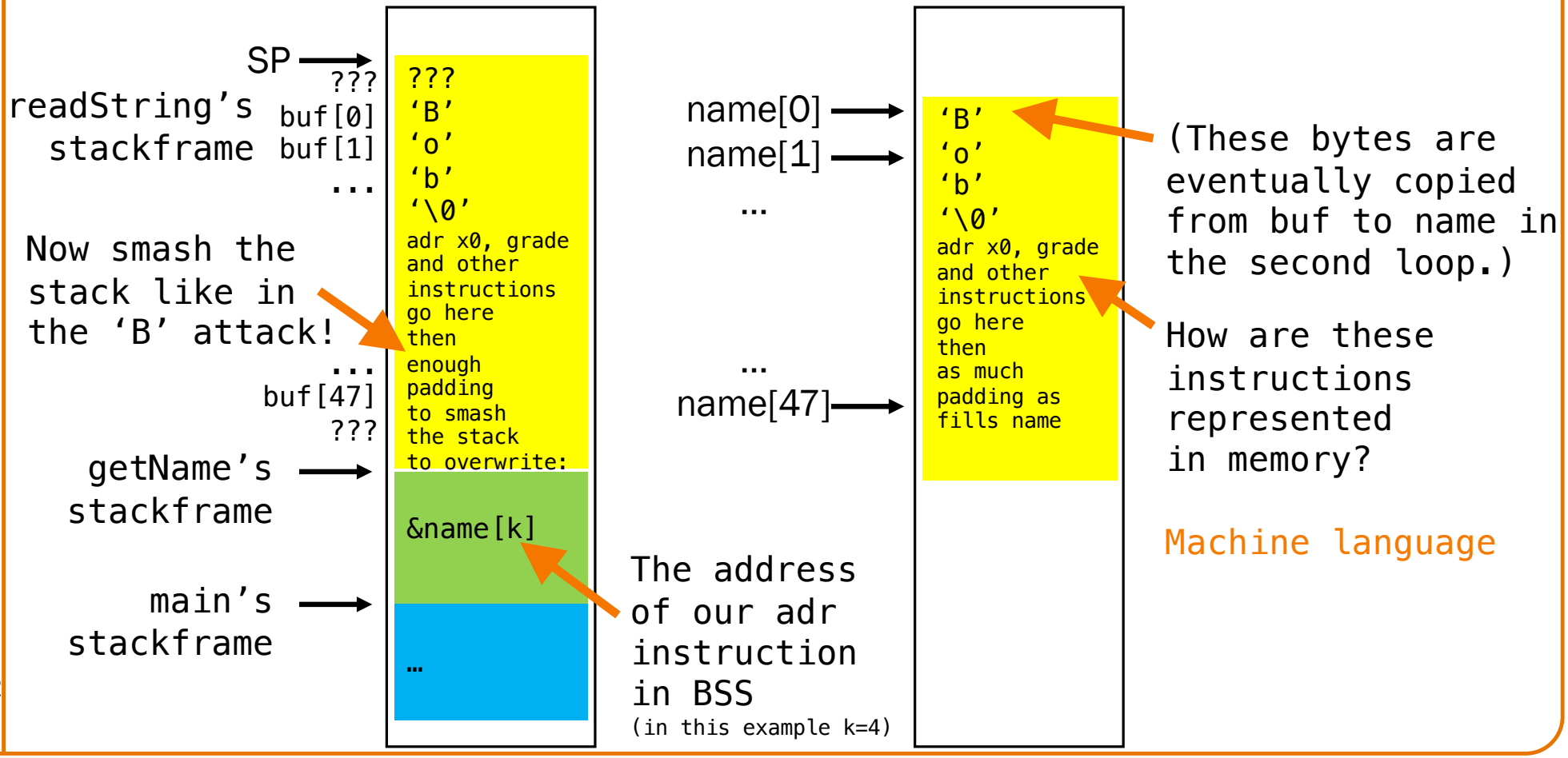
Stack and BSS Sections



Stack and BSS Sections



Stack and BSS Sections



And Then …

`readstring()` returns to `getname()`

`getname()` is done, so restores x30 from stack and returns

But transfers execution not to `main()`, but to address that was on stack for contents of x30, which is `&name[4]`

That's the attack code, in machine language, which then executes

Agenda

A6 “A” Attack

AARCH64 Machine Language

Assembly Language: `add x1, x2, x3`

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

Machine Language: TOY → AARCH64

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

```
INSTRUCTION FORMATS

              | . . . . | . . . . | . . . . | . . . . |
  Format RR:  | opcode  |    d    |    s    |    t    |  (0-6, A-B)
  Format A:   | opcode  |    d    |       addr        |  (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 6
  (Every instruction format you'll need is in the following slides… we think…)

AARCH64 Instruction Format: Operation Groups

Operation group

- Encoded in bits 25-28. So, these four bits say which operation group the instruction belongs to
- Doesn't use all four bits, Unused bits (x below) are repurposed, customized to operation group
- `x101`: Data processing – 3-register
- `100x`: Data processing – immediate + register(s)
- `101x`: Branch
- `x1x0`: Load/store

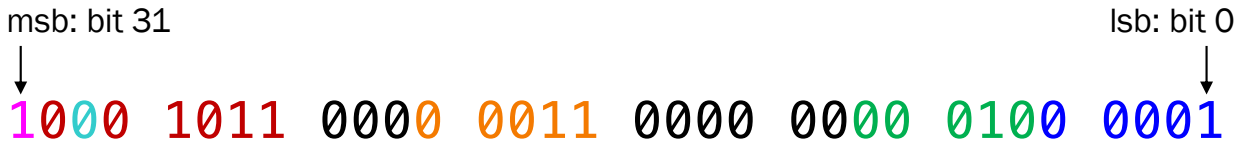
Let's look at these groups in turn, with examples that may be relevant to A6

Data Processing (3-Register) Operation Group

Op. Group: Data processing – 3-register

- Register 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



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 (Immed + Reg) Operation Group

Op. Group: Data processing – immediate + register(s)

- Two types: two-register and one-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
- 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: Subtract Immediate

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

Example: Move Immediate

**You may find this slide useful for A6

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

Branch Operation Group

Op. Group: Branch

- E.g., here, msb is not interpreted as register size, since no registers
- *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. Because this is invariable, we can omit those two bits from our representation. Doing so provides more range with the same number of bits
- Type of conditional branch encoded in bits 0-3

Displacement Discombobulation

msb: bit 31 ↓ lsb: bit 0 ↓

```
xxx1 01ii iiii iiii iiii iiii iiii iiii
```

What is the range of the relative address?

A. 0 – 64MB

B. -32MB – +32MB

C. 0 – +256MB

D. -128MB – +128MB

D: 26 bits + 2 "chopped off" bits
= 28 bits: 256MB.

2's complement splits half
negative / half non-negative

Example: Unconditional Branch

**You may find this slide useful for A6

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: Negative number. Interpreted in two's complement, which is 11_b.
  Shift left by 2: $1100_b = 12$. So, offset is -12.

Example: `bl` for Function Call

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: Negative number. Interpreted in two's complement, which is 11_b. Shift left by 2: $1100_b = 12$. So, offset is -12.
- Just additionally puts next instruction's address into x30

Example: Conditional Branch

Example: `ble someLabel`

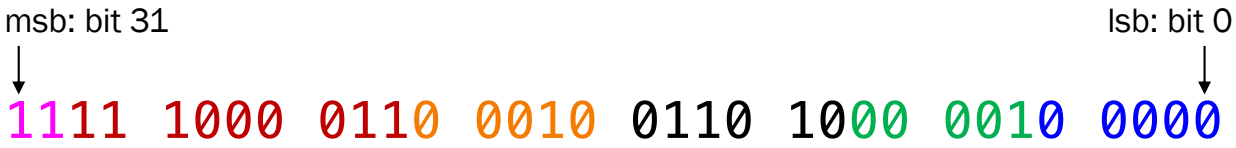
- This depends on where `someLabel` is relative to this instruction
  For this example, `someLabel` is 3 instructions (12 bytes) *later*
- opcode: conditional branch
- Conditional branch type in bits 0-3: LE (full table in precept handout)
- *Relative* address in bits 5-23: 11_b. Shift left by 2: $1100_b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

Load/Store Operation Group

Op. Group: Load / store (recall: memory addresses are in registers)

- Width of data to load/store in bits 30-31: 00 = 8-bit, 01 = 16-bit, 10 = 32-bit, 11 = 64-bit
- For [Xn,Xm] addressing mode: second source register (containing offset) in bits 16-20
- For [Xn,offset] addressing mode: offset in bits 10-21,
  shifted left by 3 bits for 64-bit load/store, 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, e.g. scaled mode

Example: Load with Register Offset



Example: `ldr x0, [x1, x2]`

- opcode: load with register offset
- 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

Example: Store with Immediate Offset

Example: `str x0, [sp,24]`

- opcode: store with immediate offset
- Instruction width in bits 30-31: 11 = 64-bit
- One less bit used for opcode, so an extra bit to use for offset
- Offset value in bits 12-20: 11_b, shifted left by 3 (3 trailing zeros assumed since addresses are 8 byte aligned) = 11000_b = 24
- "Source" (really destination, since it's a store) register in bits 5-9: all ones (31) = stack pointer
- "Destination" (really source, since it's a store) register in bits 0-4: 0
- Remember that store instructions use the opposite convention from others: "source" and "destination" are flipped

- Key point: assumed shifts depend on load/store data width

Ex: Store Byte with Register Offset **You may find this slide useful for A6

Example: `strb w0, [sp,24]`

- opcode: store byte with register offset
- Instruction width in bits 30-31: 00 = 8-bit
- One less bit used for opcode, so an extra bit to use for offset
- Offset value in bits 12-20: 11000_b (don't shift left since 8-bit so 1-byte aligned) = 24
- "Source" (really destination, since it's a store) register in bits 5-9: all ones (31) = stack pointer
- "Destination" (really source, since it's a store) register in bits 0-4: 0
- Remember that store instructions use the opposite convention from others: "source" and "destination" are flipped

- Key point: assumed shifts depend on load/store data width

A Variant: adr

ADR instruction (Distinct from others with Op Group bits 100x – which are Data Processing with Register + Immediate)

- Specifies *relative* position of label (data location)
- But label for adr could be a further-away address (in rodata, data, bss, or text), so want to maximize the size of the relative address
- Destination register in bits 0-4
- 19 High-order bits of offset in bits 5-23
- 2 Low-order bits of offset in bits 29-30

- So have to put together disjoint portions to get the relative address

Example of `adr`

**You may find this slide useful for A6

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