Lecture 14
(end of Chapter 3)
Simplified and condensed Slides

Previously

- Complex data:
  - Arrays
  - structures and their alignment
  - Unions

- Questions about any of the exercises suggested last time? On assembly translation of structures and alignment

Today (change of order)

1. Compiling Floating Point (Section 3.11)
2. Understanding Pointers (Section 3.10.1)
3. Memory Layout and unions (Section 3.9.2)
4. Buffer Overflow (Section 3.10.3)
   - Vulnerability
   - Protection

Topics

1. Compiling Floating Point (Section 3.11)
2. Understanding Pointers (Section 3.10.1)
3. Memory Layout and (Sec. 3.9.2)
4. Buffer Overflow (Section 3.10.3)
   - Vulnerability
   - Protection

2. Compiling Floating Point

- Floating Point Data handled at machine-level with a separate set of registers and instructions
- Since Pentium/MMX processors in 1997
  - Successive sets of media instructions to support graphics and image processing
  - Focus: multiple operations performed in parallel
    - SIMD (sim-dee) simple instruction, multiple data
  - Progression of Extensions to instruction set have different names:
    - MMX to SSE (*streaming SIMD extensions*)
    - AVX (*advanced vector extensions*)

Compiling Floating Point (2)

- Accordingly, names of registers are also different:
  - MM registers (64 bits) for MMX
  - XMM registers for SSE
  - YMM registers for AVX
- AVX2 with Core i7 Haswell processor in 2013
  - textbook covers only instructions that arise in compiling fp programs with gcc, i.e. scalar AVX instructions
- XMM registers have 16 bytes (128 bits) (low order 16 bytes of the YMM registers)
- YMM registers have 32 bytes (256 bits)

Programming with SSE3 (fig 3.45 p.295)

XMM Registers

- 16 total, each 16 bytes, low-order 16 bytes (128 bits)
- 16 single-byte integers

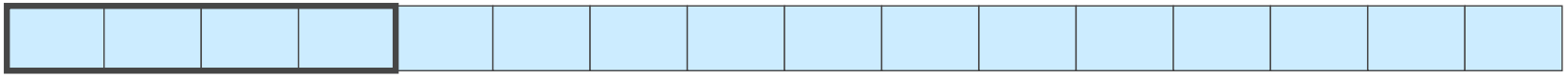
- 8 16-bit integers

- 4 32-bit integers

- 4 single-precision floats

- 2 double-precision floats

- 1 single-precision float

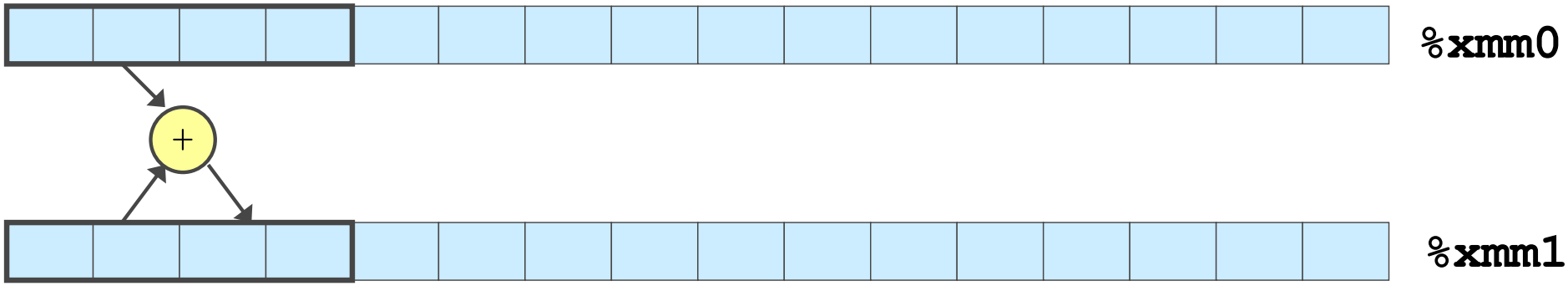


- 1 double-precision float

Scalar & SIMD Operations

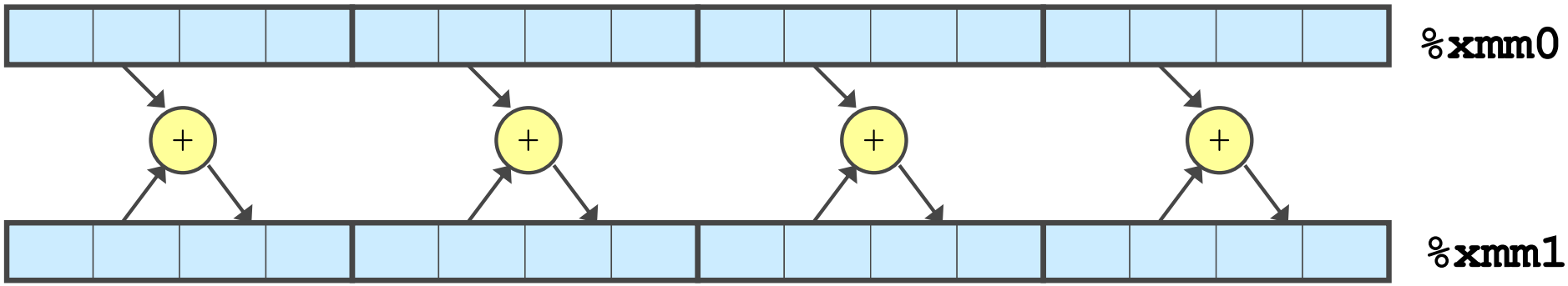
- Scalar Operations: Single Precision

addss %xmm0,%xmm1



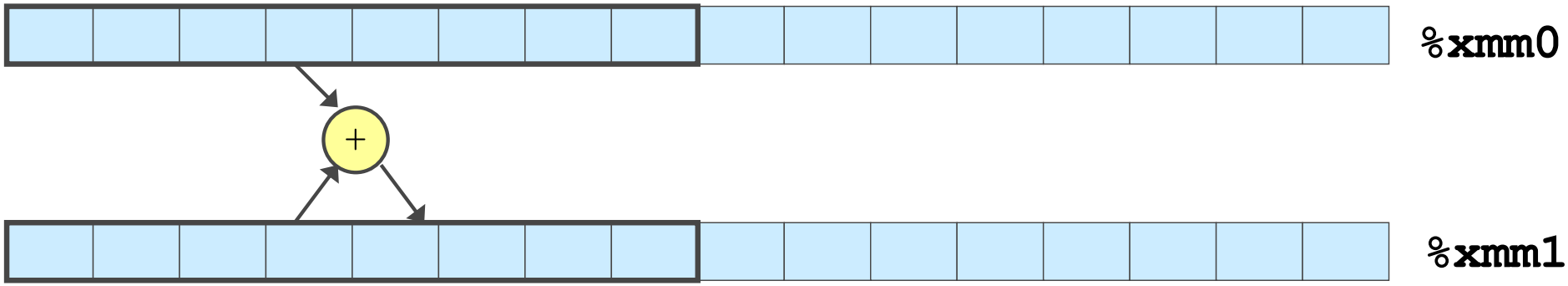
- SIMD Operations: Single Precision

addps %xmm0,%xmm1



- Scalar Operations: Double Precision

addsd %xmm0,%xmm1



FP Basics

- Arguments passed in `%xmm0`, `%xmm1`, …
- Result returned in `%xmm0`
- All XMM registers caller-saved

```
float fadd(float x, float y)
{
    return x + y;
}
```

```
double dadd(double x, double y)
{
    return x + y;
}
```

```
  # x in %xmm0, y in %xmm1
  addss   %xmm1, %xmm0
  ret
```

```
 # x in %xmm0, y in %xmm1
  addsd   %xmm1, %xmm0
  ret
```

FP Memory Referencing

- Integer (and pointer) arguments passed in regular registers
- FP values passed in XMM registers
- Different mov instructions to move between XMM registers, and between memory and XMM registers

```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

a is for aligned, p is for packed,
d is for double, s for single
`movsd` scalar double prec fp

```
  # p in %rdi, v in %xmm0
  movapd  %xmm0, %xmm1    # Copy v
  movsd   (%rdi), %xmm0   # x = *p
  addsd   %xmm0, %xmm1    # t = x + v
  movsd   %xmm1, (%rdi)   # *p = t
  ret
```

Other Aspects of FP Code

- *Lots* of instructions
  - Different operations, different formats, ...
- Floating-point comparisons
  - Instructions **`ucomiss`** and **`ucomisd`**
  - Set condition codes CF, ZF, and PF
- Using constant values
  - Set XMM0 register to 0 with instruction **`xorpd %xmm0, %xmm0`**
  - Others loaded from memory (cannot be directly used in the addressing mode as with integers and pointers)
- 32-bit memory req. 4-byte alignment (what we said about this being system dependent)
- 64-bit memory req. 8-byte alignment

Summary (of data objects)

- Arrays
  - Elements packed into contiguous region of memory
  - Use index arithmetic to locate individual elements
- Structures
  - Elements packed into single region of memory
  - Access using offsets determined by compiler
  - Possibly require internal and external padding to ensure alignment
- Combinations
  - Can nest structure and array code arbitrarily
- Floating Point
  - Data held and operated on in the XMM registers

Topics (sections in different order)

1. Wrap-up structs, UNIONs (Section 3.9.2)
2. Compiling Floating Point (Section 3.11)
3. Understanding Pointers (Section 3.10.1)
4. Memory Layout (Section 3.9.2)
5. Buffer Overflow (Section 3.10.3)
   - Vulnerability
   - Protection

Pointers

- Uniform way to generate references to elements within data structures
- Source of confusion (for novice programmers)
- Key principles and their mapping into machine code must be understood.

Key principles of Pointers (sec. 3.10.1)

1. Every pointer has an associated type.
2. Every pointer has a value.
3. Pointers are created with the ‘&’ operator.
4. Pointers are dereferenced with the ‘*’ op.
5. Strong relationship: pointers and arrays.
6. Casting from one type of pointer to another changes the type but not its value.
7. Pointers can also point to functions.

Suggested reading

Use the [KRC] book to refresh:

- sizeof unary operator: compile-time, used to compute the size of any object. It yields an integer equal to the size of the specified object or type in bytes. Produces an unsigned integer value whose type is size_t
- pointers and arrays (Chapter 5): specially sections 2,6,11 & 12

The image part with relationship ID rId13 was not found in the file.

How to read the next slides

- In the next seven (7) slides there are expressions with pointers to help you review how pointers work in C. (Explain a few; on your own the remaining)
- Each table on each slide shows declarations using arrays and/or pointers and then show the possible uses of the variables declared that way.
- In the cells, three aspects are presented:
  - does the expression compile
  - could it lead to a bad pointer reference
  - size of the value returned by the expression

Understanding Pointers & Arrays #1

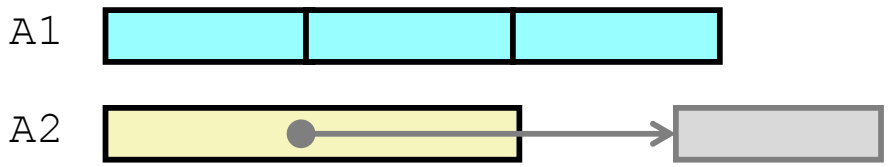
Decl	*An*			**An*		
	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`						
`int *A2`						

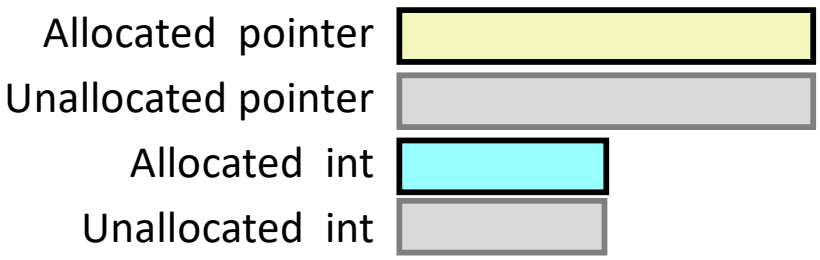
- Cmp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by `sizeof`

Understanding Pointers /Arrays #1

Decl	*An*			**An*		
	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4
`int *A2`	Y	N	8	Y	Y	4

* dereferences
& gets address





dereferencing a null pointer gives an execution error

- Cmp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by `sizeof`

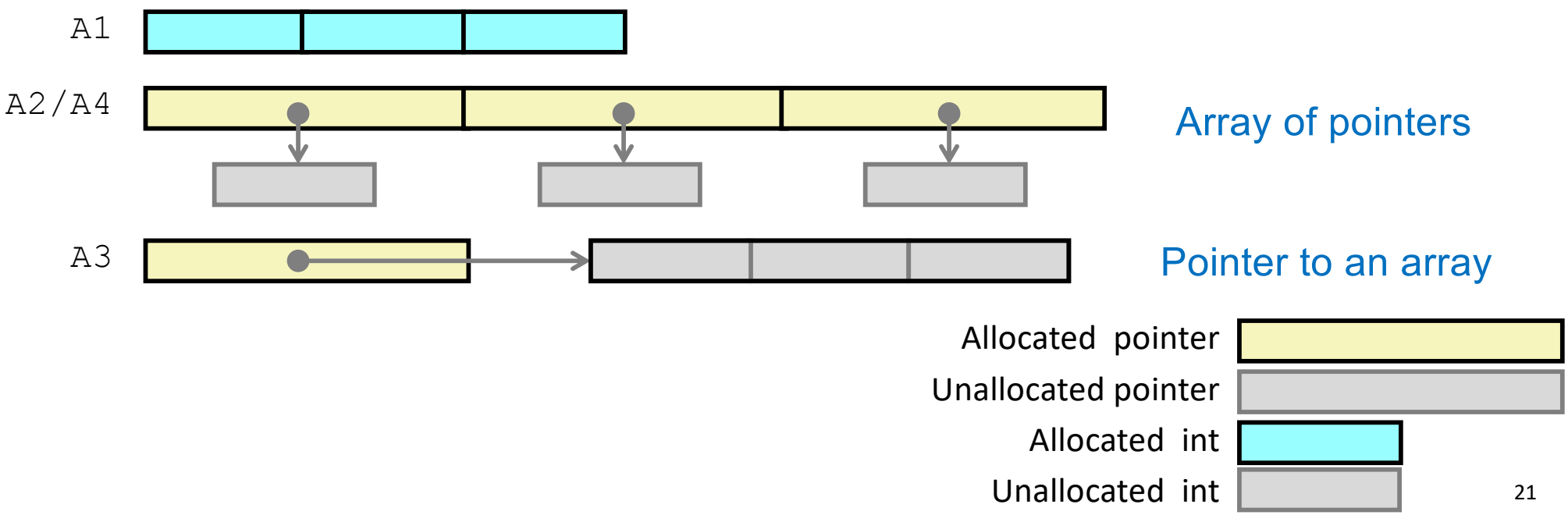
Understanding Pointers/Arrays #2

Decl	*An*			**An*			***An*		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`									
`int *A2[3]`									
`int (*A3)[3]`									
`int (*A4[3])`									

- Cmp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by `sizeof`

Understanding Pointers/Arrays #2

Decl	*An*			**An*			***An*		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4	N	–	–
`int *A2[3]`	Y	N	24	Y	N	8	Y	Y	4
`int (*A3)[3]`	Y	N	8	Y	Y	12	Y	Y	4
`int (*A4[3])`	Y	N	24	Y	N	8	Y	Y	4



Understanding Pointers/Arrays #3

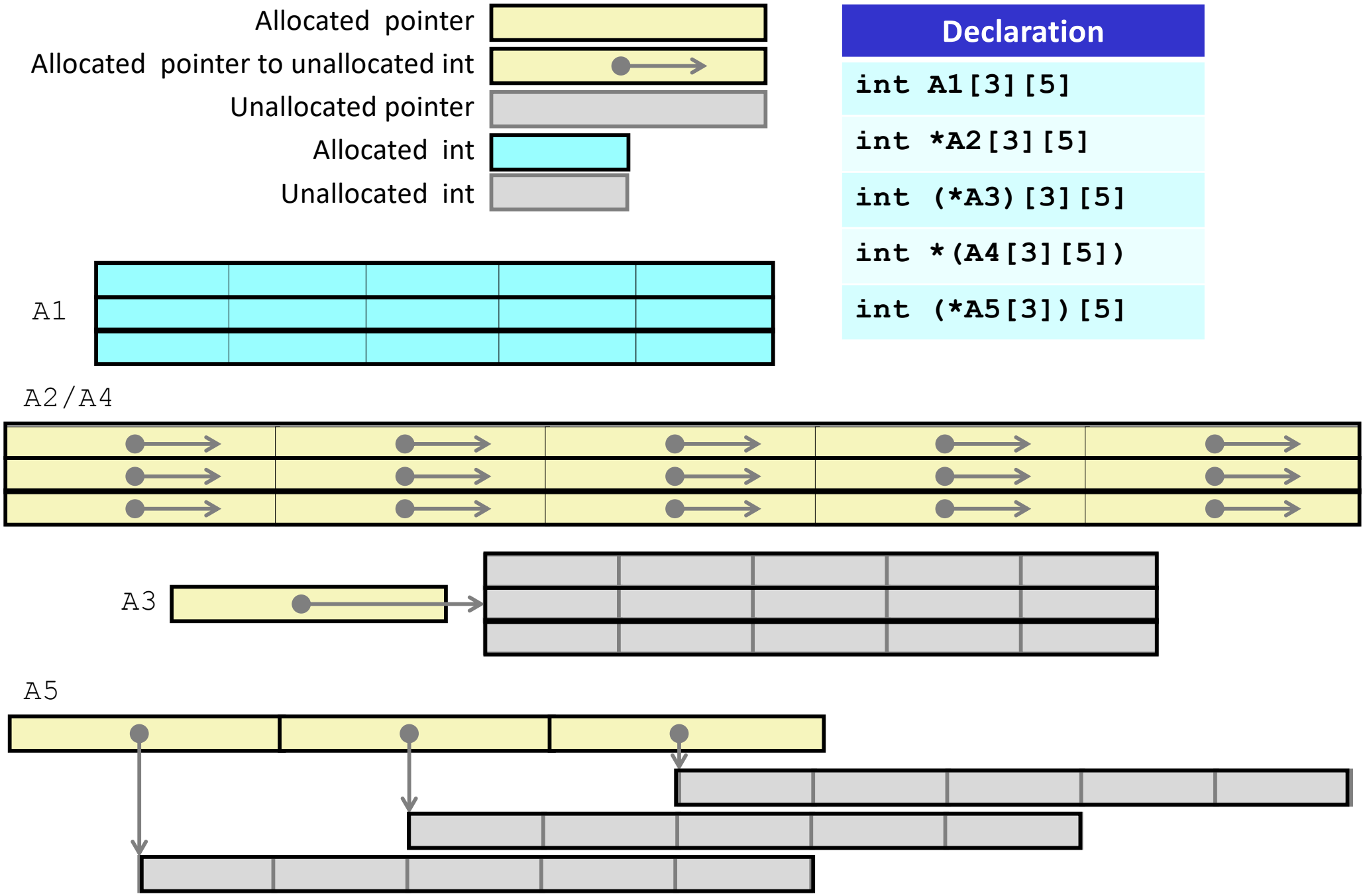
Decl	*An*			**An*			***An*		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`									
`int *A2[3][5]`									
`int (*A3)[3][5]`									
`int *(A4[3][5])`									
`int (*A5[3])[5]`									

- Cmp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by `sizeof`

Decl	****An*		
	Cmp	Bad	Size
`int A1[3][5]`			
`int *A2[3][5]`			
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

Operator precedence in C

Operator	Description	Associativity
( ) [ ] . ->>>>



Declaration
`int A1[3][5]`
`int *A2[3][5]`
`int (*A3)[3][5]`
`int *(A4[3][5])`
`int (*A5[3])[5]`

Understanding Pointers & Arrays #3

Decl	*An*			**An*			***An*		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`	Y	N	60	Y	N	20	Y	N	4
`int *A2[3][5]`	Y	N	120	Y	N	40	Y	N	8
`int (*A3)[3][5]`	Y	N	8	Y	Y	60	Y	Y	20
`int *(A4[3][5])`	Y	N	120	Y	N	40	Y	N	8
`int (*A5[3])[5]`	Y	N	24	Y	N	8	Y	Y	20

- Cmp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by `sizeof`

Decl	****An*		
	Cmp	Bad	Size
`int A1[3][5]`	N	–	–
`int *A2[3][5]`	Y	Y	4
`int (*A3)[3][5]`	Y	Y	4
`int *(A4[3][5])`	Y	Y	4
`int (*A5[3])[5]`	Y	Y	4

Topics

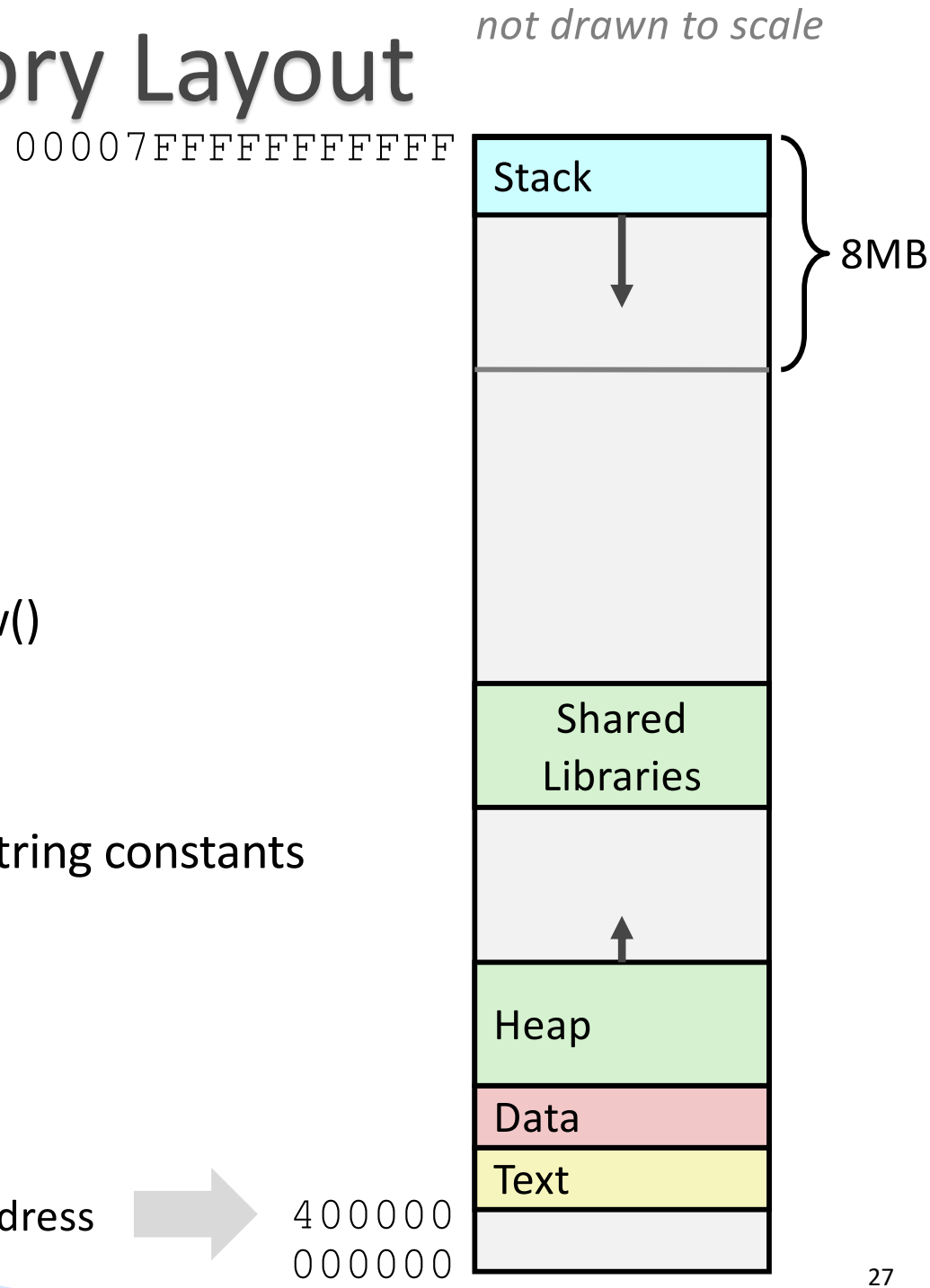
1. Wrap-up structs, UNIONs (Section 3.9.2)
2. Compiling Floating Point (Section 3.11)
3. Understanding Pointers (Section 3.10.1)
4. Memory Layout and unions (Section 3.9.2)
5. Buffer Overflow (Section 3.10.3)
   - Vulnerability
   - Protection

The image part with relationship ID rId13 was not found in the file.

x86-64 Linux Memory Layout

- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables
- Heap
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- Data
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only

The image part with relationship ID rId13 was not found in the file.



Memory Allocation Example

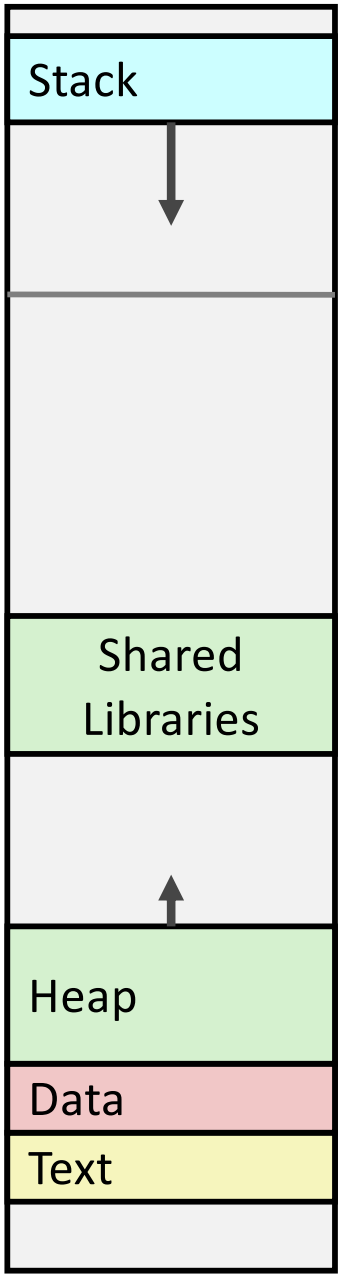
not drawn to scale

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```



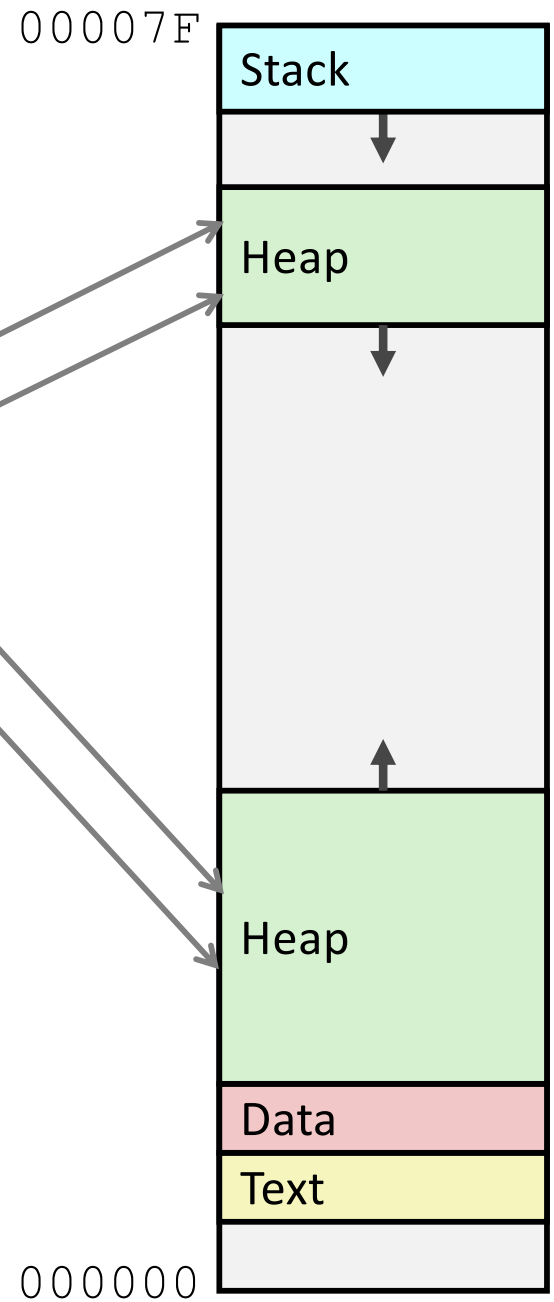
Where does everything go?

x86-64 Example Addresses

not drawn to scale

address range ~2^{47}

`local`	`0x00007ffe4d3be87c`
`p1`	`0x00007f7262a1e010`
`p3`	`0x00007f7162a1d010`
`p4`	`0x000000008359d120`
`p2`	`0x000000008359d010`
`big_array`	`0x0000000080601060`
`huge_array`	`0x0000000000601060`
`main()`	`0x000000000040060c`
`useless()`	`0x0000000000400590`



The image part with relationship ID rId13 was not found in the file.

Topics (notice change of order)

1. Wrap-up structs, UNIONs (Section 3.9.2)
2. Compiling Floating Point (Section 3.11)
3. Understanding Pointers (Section 3.10.1)
4. Memory Layout (Section 3.9.2)
5. Buffer Overflow (Section 3.10.3)
   - Vulnerability
   - Protection

Recall: Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)  → 3.14
fun(1)  → 3.14
fun(2)  → 3.1399998664856
fun(3)  → 2.00000061035156
fun(4)  → 3.14
fun(6)  → Segmentation fault
```
(another example, not only deref null pointer)

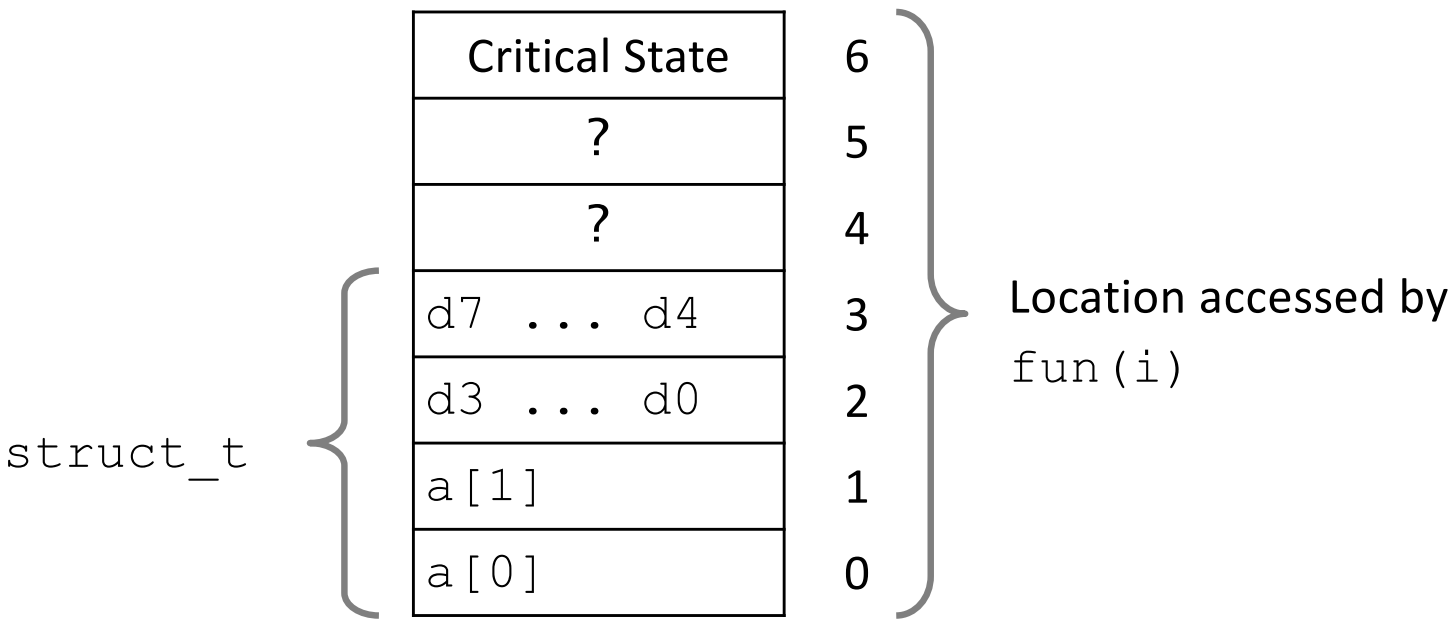
- Result is system specific

Memory Referencing Bug Example(2)

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0)  → 3.14
fun(1)  → 3.14
fun(2)  → 3.1399998664856
fun(3)  → 2.00000061035156
fun(4)  → 3.14
fun(6)  → Segmentation fault
```

Explanation:



Such problems are a BIG deal

- Generally called a “buffer overflow”
  - when exceeding the memory size allocated for an array
- Why a big deal?
  - It’s the #1 technical cause of security vulnerabilities (still)
    - #1 overall cause is social engineering / user ignorance
- Most common form
  - Unchecked lengths on string inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

The image part with relationship ID rId13 was not found in the file.

String Library Code

- Implementation of Unix function `gets()`

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - No way to specify limit on number of characters to read
- Similar problems with other library functions
  - **`strcpy`, `strcat`**: Copy strings of arbitrary length
  - **`scanf`, `fscanf`, `sscanf`,** when given **`%s`** conversion specification

Vulnerable Buffer Code (uses gets)

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

← btw, how big is big enough?

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901 23
0123456789012345678901 23
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

The image part with relationship ID rId13 was not found in the file.

Buffer Overflow Disassembly

$0x18 = 24_{10}$

echo:

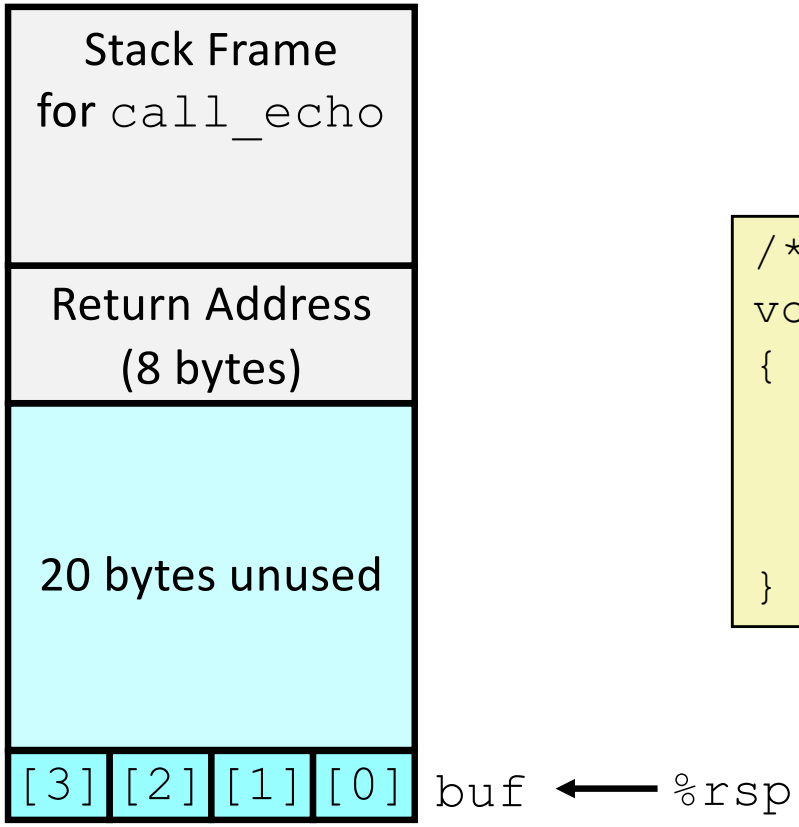
```
00000000004006cf <echo>:
 4006cf:  48 83 ec 18            sub    $0x18,%rsp
 4006d3:  48 89 e7               mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff         callq  400680 <gets>
 4006db:  48 89 e7               mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff         callq  400520 <puts@plt>
 4006e3:  48 83 c4 18            add    $0x18,%rsp
 4006e7:  c3                     retq
```

call_echo:

```
4006e8:   48 83 ec 08            sub    $0x8,%rsp
 4006ec:  b8 00 00 00 00         mov    $0x0,%eax
 4006f1:  e8 d9 ff ff ff         callq  4006cf <echo>
 4006f6:  48 83 c4 08            add    $0x8,%rsp
 4006fa:  c3                     retq
```

Buffer Overflow Stack

Before call to gets

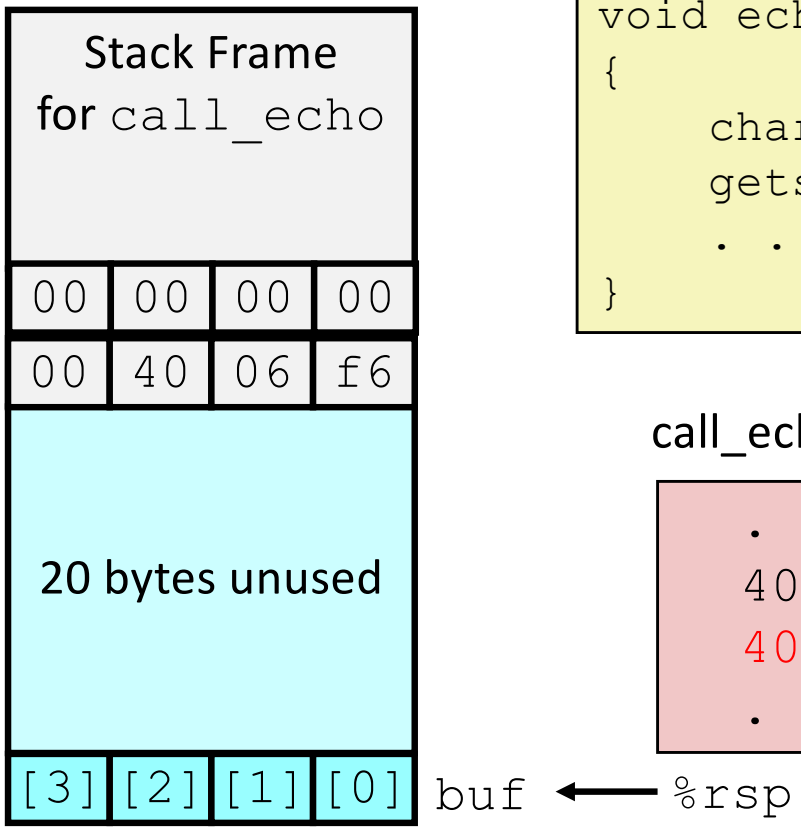



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

Buffer Overflow Stack Example

Before call to gets




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
. . .
4006f1:  callq  4006cf <echo>
4006f6:  add    $0x8,%rsp
. . .
```

Buffer Overflow Stack Example #1

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
. . .
4006f1:  callq  4006cf <echo>
4006f6:  add    $0x8,%rsp
. . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901 2
0123456789012345678901 2
```

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack Example #2

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
. . .
4006f1:  callq  4006cf <echo>
4006f6:  add    $0x8,%rsp
. . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

Overflowed buffer and corrupted return pointer

Buffer Overflow Stack Example #3

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

```
. . .
4006f1:  callq  4006cf <echo>
4006f6:  add    $0x8,%rsp
. . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901 23
0123456789012345678901 23
```

Overflowed buffer, corrupted return pointer, but program seems to work!

Buffer Overflow Stack Example #3 Explained

After call to gets

Stack Frame for `call_echo`			
00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

`buf` ← `%rsp`

register_tm_clones:

```
. . .
400600:  mov    %rsp,%rbp
400603:  mov    %rax,%rdx
400606:  shr    $0x3f,%rdx
40060a:  add    %rdx,%rax
40060d:  sar    %rax
400610:  jne    400614
400612:  pop    %rbp
400613:  retq
```

"Returns" to unrelated code

Lots of things happen, without modifying critical state

Eventually executes `retq` back to `main`