

# COMP201

## Computer Systems & Programming

Lecture #27 – Security Vulnerabilities

---



KOÇ  
UNIVERSITY

Aykut Erdem // Koç University // Fall 2020

# Recap

- Arrays
- Structures
  - Allocation
  - Access
  - Alignment

# Plan for Today

- Structures and Alignment
- Floating Point
- Memory Layout
- Buffer Overflow

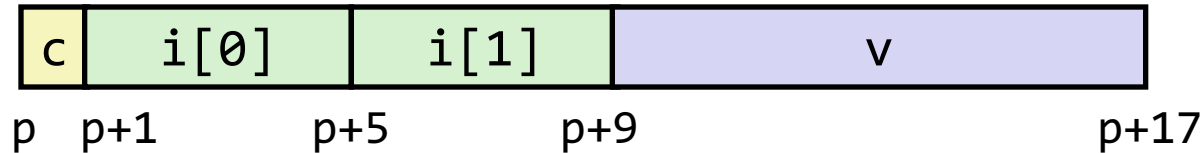
**Disclaimer:** Slides for this lecture were borrowed from  
—Randal E. Bryant and David R. O'Hallaron's CMU 15-213 class

# Lecture Plan

- Structures and Alignment
- Floating Point
- Memory Layout
- Buffer Overflow

# Structures & Alignment

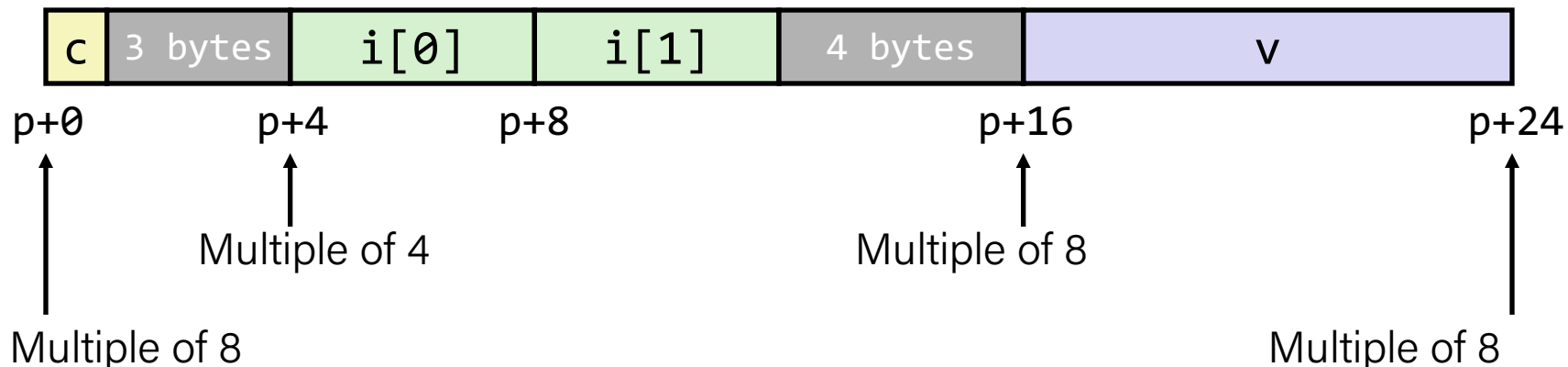
## Unaligned Data



```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```

## Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K



# Alignment Principles

## **Aligned Data**

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on x86-64

## **Motivation for Aligning Data**

- Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory trickier when datum spans 2 pages

## **Compiler**

- Inserts gaps in structure to ensure correct alignment of fields

# Specific Cases of Alignment (x86-64)

- 1 byte: `char`, ...
  - no restrictions on address
- 2 bytes: `short`, ...
  - lowest 1 bit of address must be  $0_2$
- 4 bytes: `int`, `float`, ...
  - lowest 2 bits of address must be  $00_2$
- 8 bytes: `double`, `long`, `char *`, ...
  - lowest 3 bits of address must be  $000_2$
- 16 bytes: `long double` (GCC on Linux)
  - lowest 4 bits of address must be  $0000_2$

# Satisfying Alignment with Structures

## Within structure:

- Must satisfy each element's alignment requirement

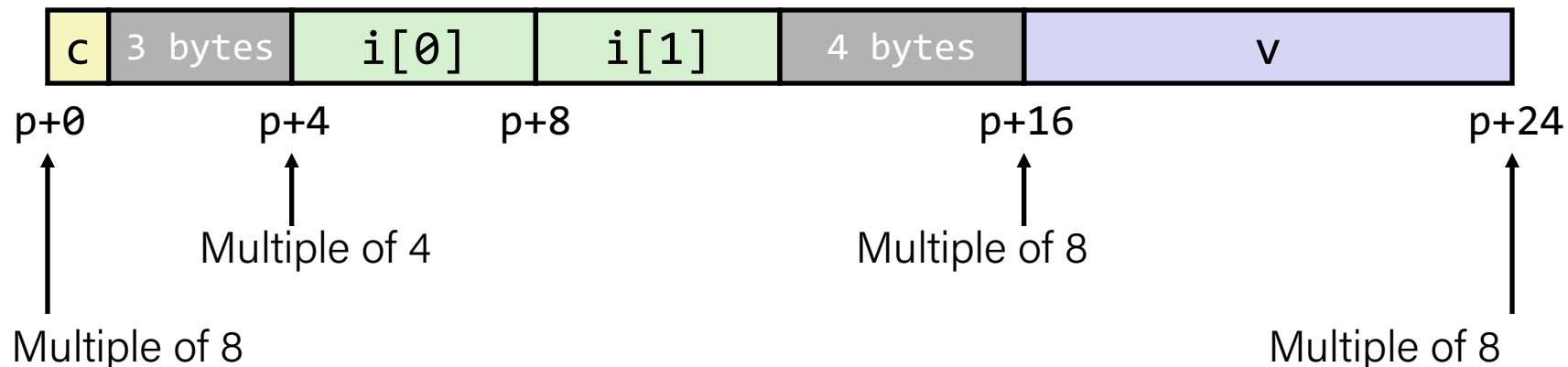
## Overall structure placement

- Each structure has alignment requirement K
  - K = Largest alignment of any element
- Initial address & structure length must be multiples of K

```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```

## Example:

- K = 8, due to **double** element

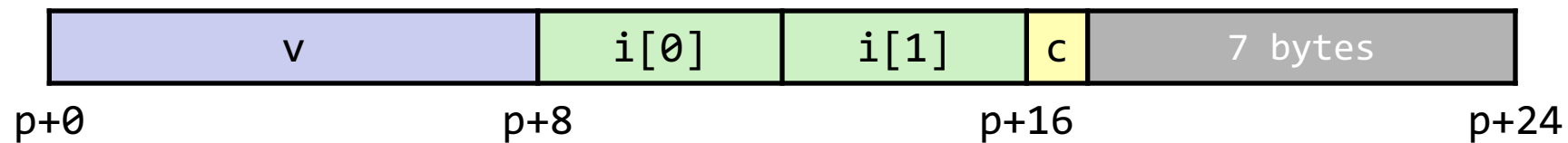




# Meeting Overall Alignment Requirement

- For largest alignment requirement  $K$
- Overall structure must be multiple of  $K$

```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```

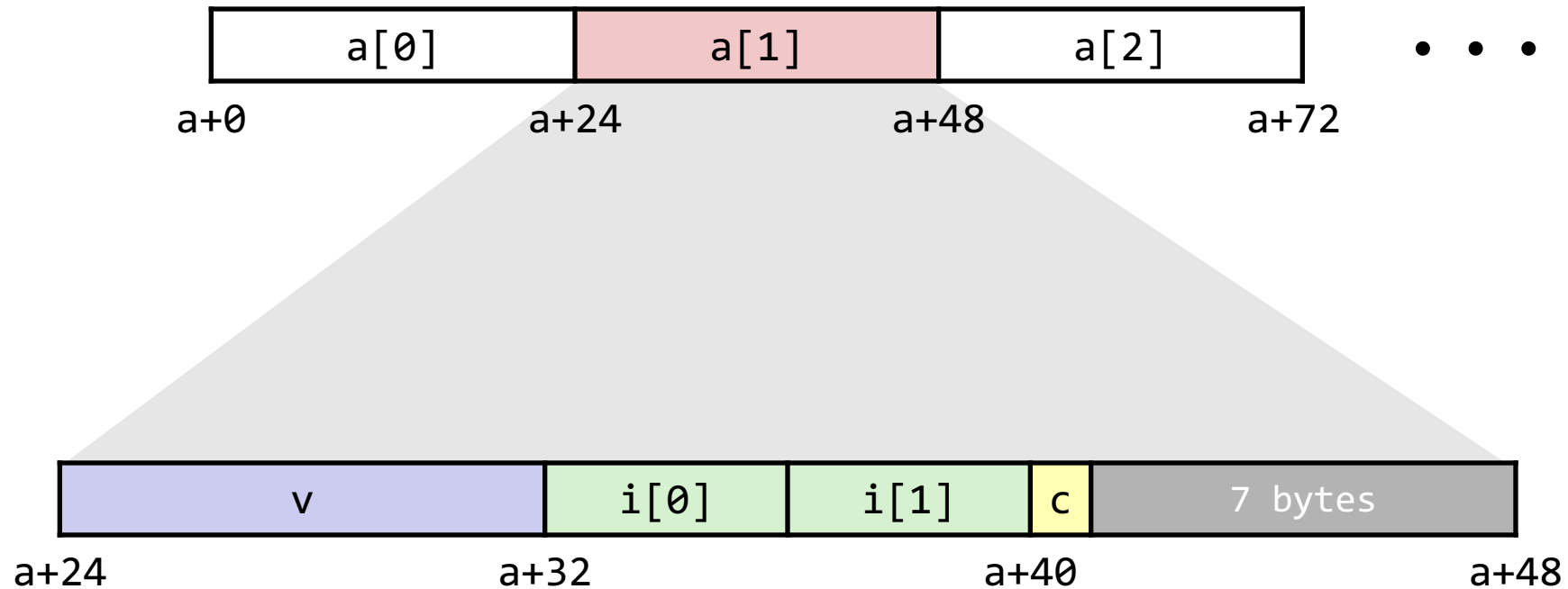


Multiple of  $K=8$

# Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

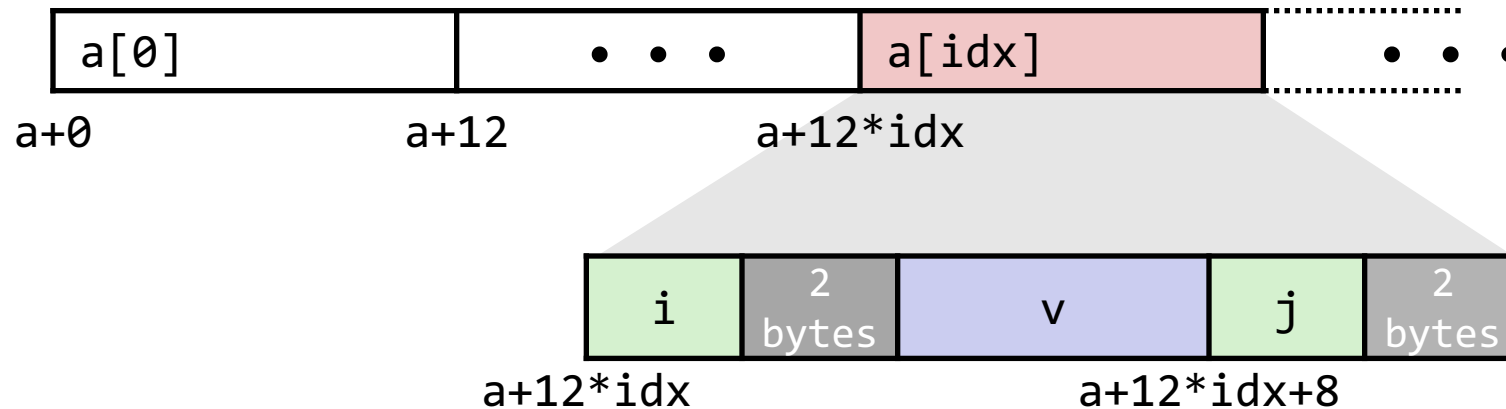
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} a[10];
```



# Accessing Array Elements

- Compute array offset  $12 \cdot \text{idx}$ 
  - `sizeof(S3)`, including alignment spacers
- Element `j` is at offset 8 within structure
- Assembler gives offset `a+8` (resolved during linking)

```
struct S3 {  
    short i;  
    float v;  
    short j;  
} a[10];
```



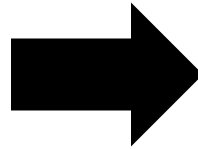
```
short get_j(int idx) {  
    return a[idx].j;  
}
```

```
# %rdi = idx  
leaq (%rdi,%rdi,2),%rax # 3*idx  
movzwl a+8(,%rax,4),%eax
```

# Saving Space

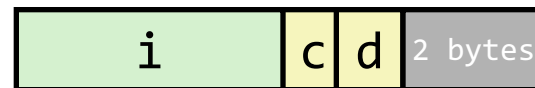
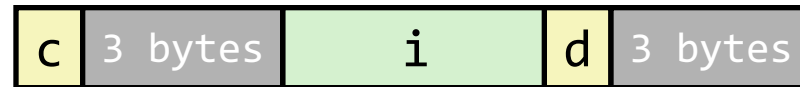
- Put large data types first

```
struct S4 {  
    char c;  
    int i;  
    char d;  
} *p;
```



```
struct S5 {  
    int i;  
    char c;  
    char d;  
} *p;
```

- Effect (K=4)



# Practice: Alignment



Event code:  
73165

Determine the offset of each field, the total size of the structure, and its alignment requirement for x86-64.

```
struct mystruct {  
    int *a;  
    float b;  
    char c;  
    short d;  
    long e;  
    double f;  
    int g;  
    char *h;  
};
```

Field	*a	b	c	d	e	f	g	*h	Total	Alignment
Size	8	4	1	2	8	8	4	8	56	8
Offset	0	8	12	16	24	32	40	48		

**Rearranged structure with minimum wasted space:**

Field	*a	h	f	e	b	g	d	c	Total	Alignment
Size	8	8	8	8	4	4	2	1	48	8
Offset	0	8	16	24	32	36	40	42		

5 bytes padded to satisfy alignment requirement

# Lecture Plan

- Structures and Alignment
- Floating Point
- Memory Layout
- Buffer Overflow

# Background

- History
  - x87 FP
    - Legacy, very ugly
  - Streaming SIMD Extensions (SSE) FP
    - SIMD: single instruction, multiple data
    - Special case use of vector instructions
  - AVX FP
    - Newest version
    - Similar to SSE
    - Documented in book

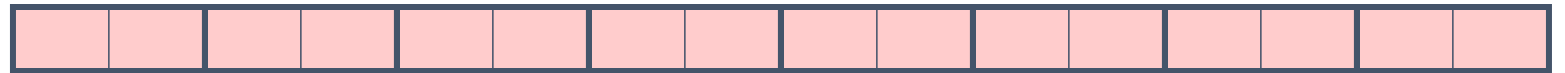
# Programming with SSE3

## XMM Registers

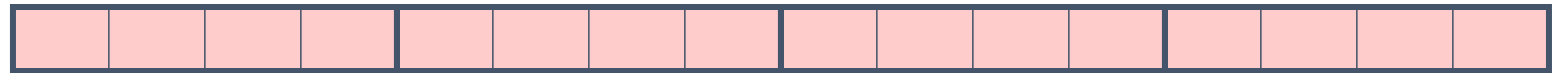
- 16 total, each 16 bytes
- 16 single-byte integers



- 8 16-bit integers



- 4 32-bit integers



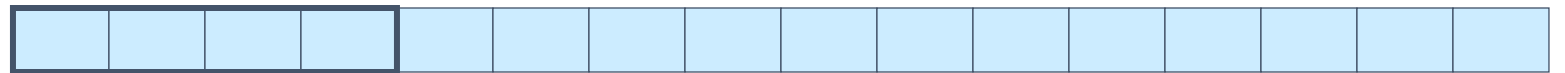
- 4 single-precision floats



- 2 double-precision floats



- 1 single-precision float



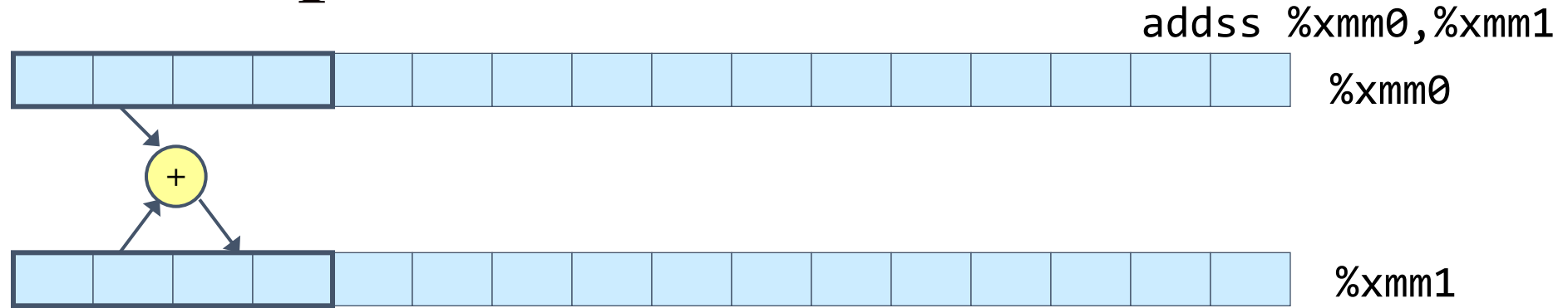
- 1 double-precision float



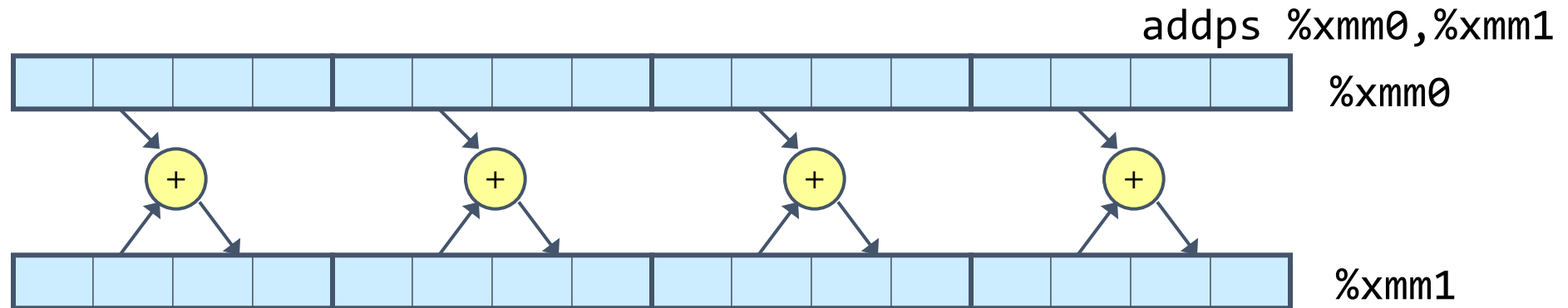


# Scalar & SIMD Operations

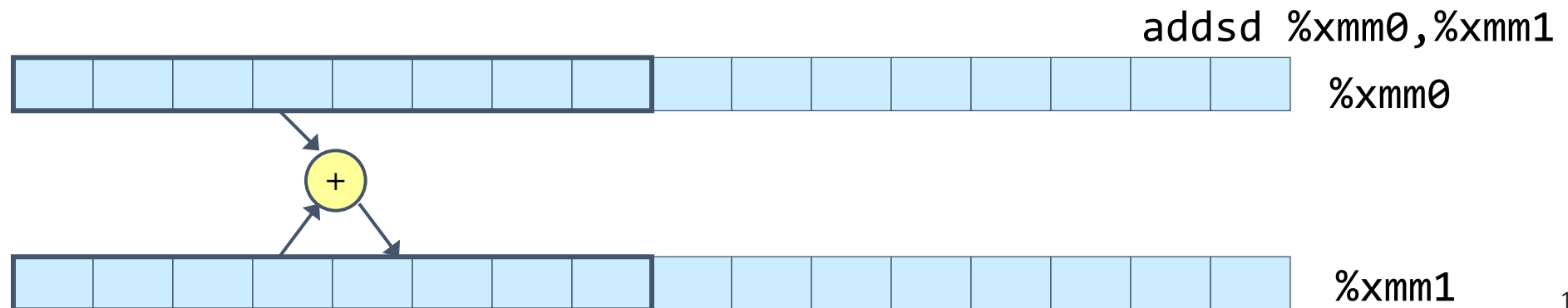
- Scalar Operations:  
Single Precision



- SIMD Operations:  
Single Precision



- Scalar Operations:  
Double Precision



# 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;
}
```

```
# 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

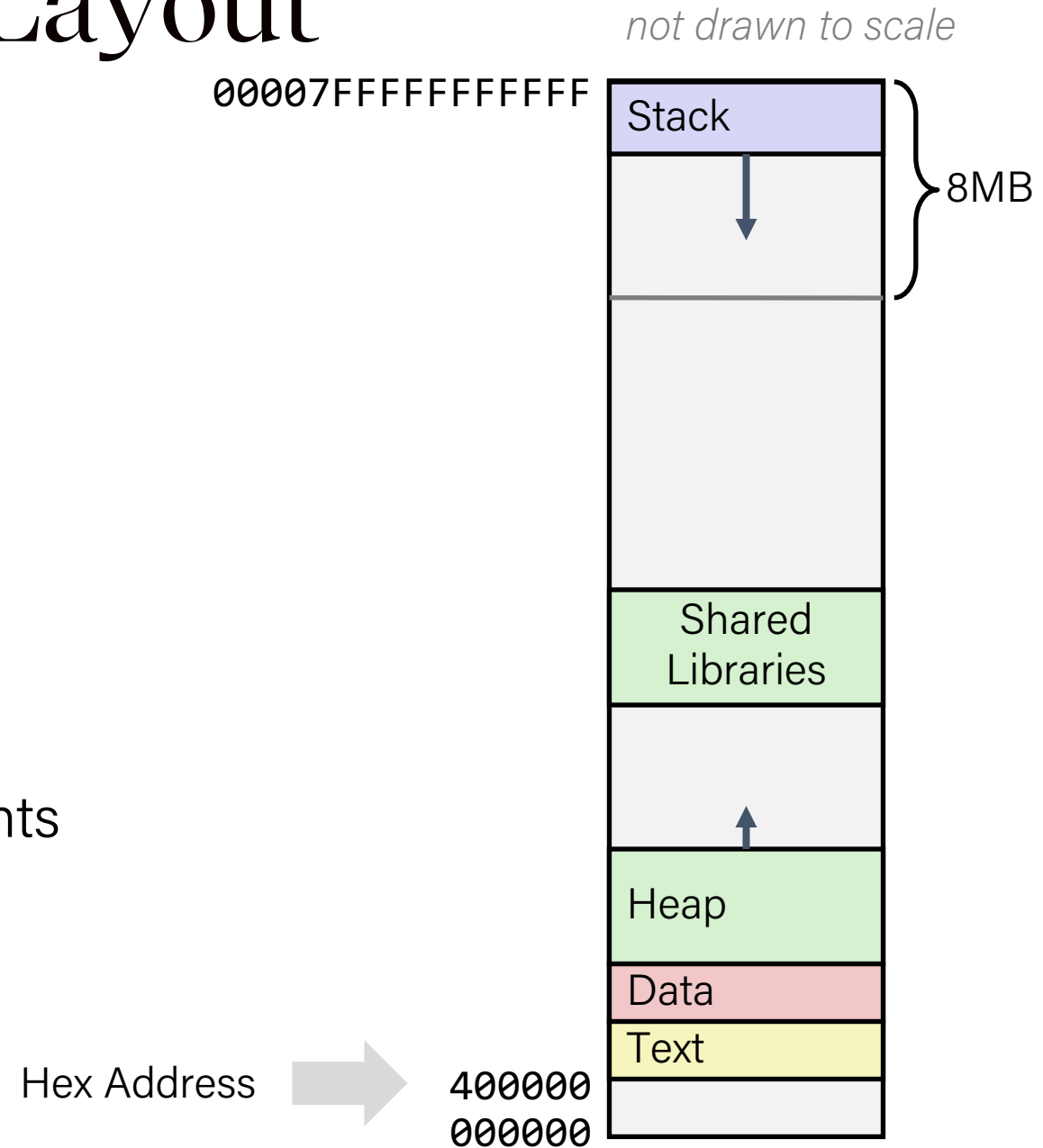
- *Lots* of instructions
  - Different operations, different formats, ...
- Floating-point comparisons
  - Instructions `ucomiss` and `ucomisd`
  - Set condition codes CF, ZF, and PF
- Using constant values
  - Set XMM0 register to 0 with instruction `xorpd %xmm0, %xmm0`
  - Others loaded from memory

# Lecture Plan

- Structures and Alignment
- Floating Point
- Memory Layout
- Buffer Overflow

# 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



# Memory Allocation Example

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

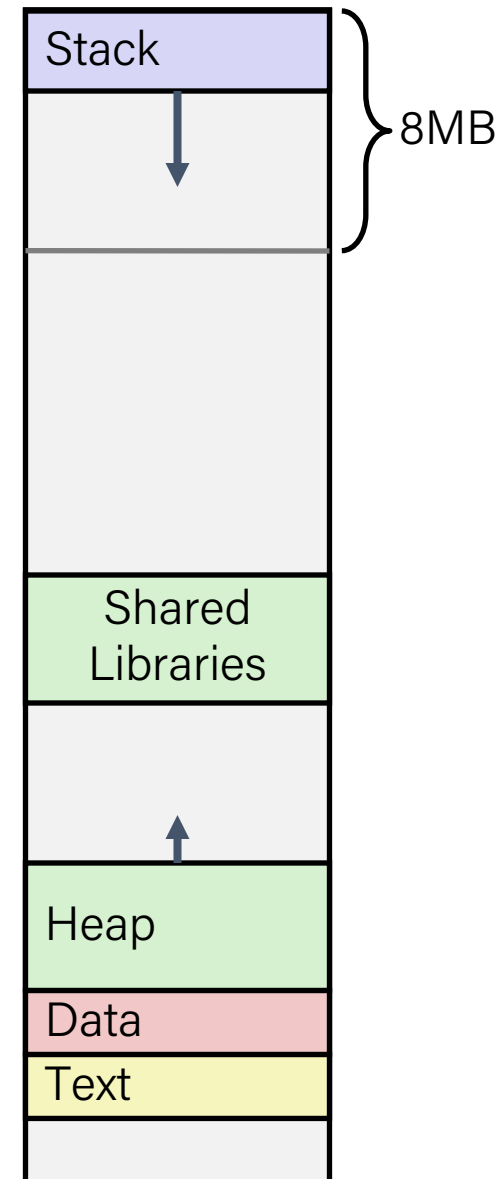
int global = 0;

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?

*not drawn to scale*



# x86-64 Example Addresses

*address range  $\sim 2^{47}$*

local  
p1  
p3  
p4  
p2  
big\_array  
huge\_array  
main()  
useless()

0x00007ffe4d3be87c

0x00007f7262a1e010

0x00007f7162a1d010

0x0000000008359d120

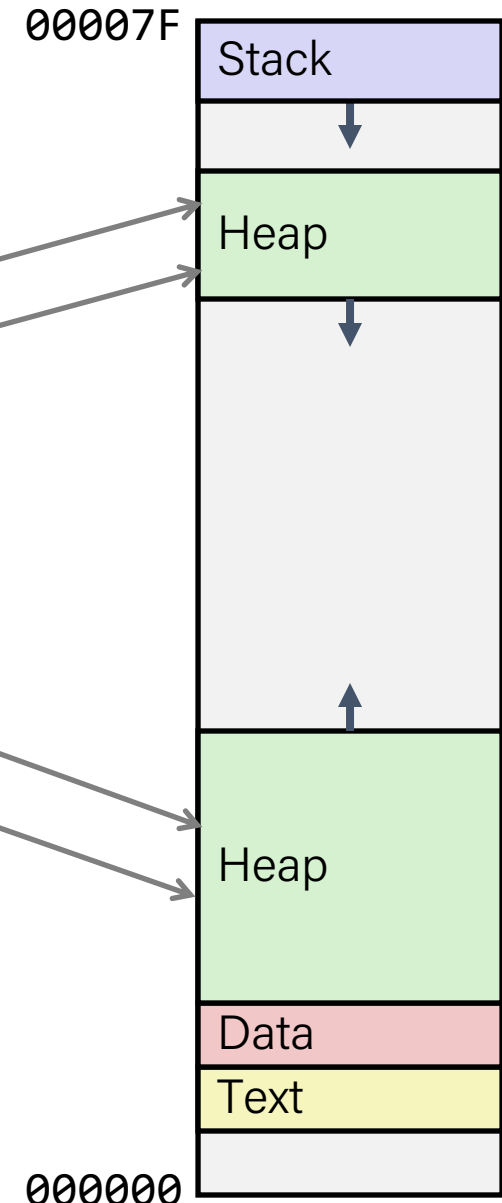
0x0000000008359d010

0x00000000080601060

0x00000000000601060

0x0000000000040060c

0x00000000000400590





# Lecture Plan

- Structures and Alignment
- Floating Point
- Memory Layout
- Buffer Overflow
  - 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

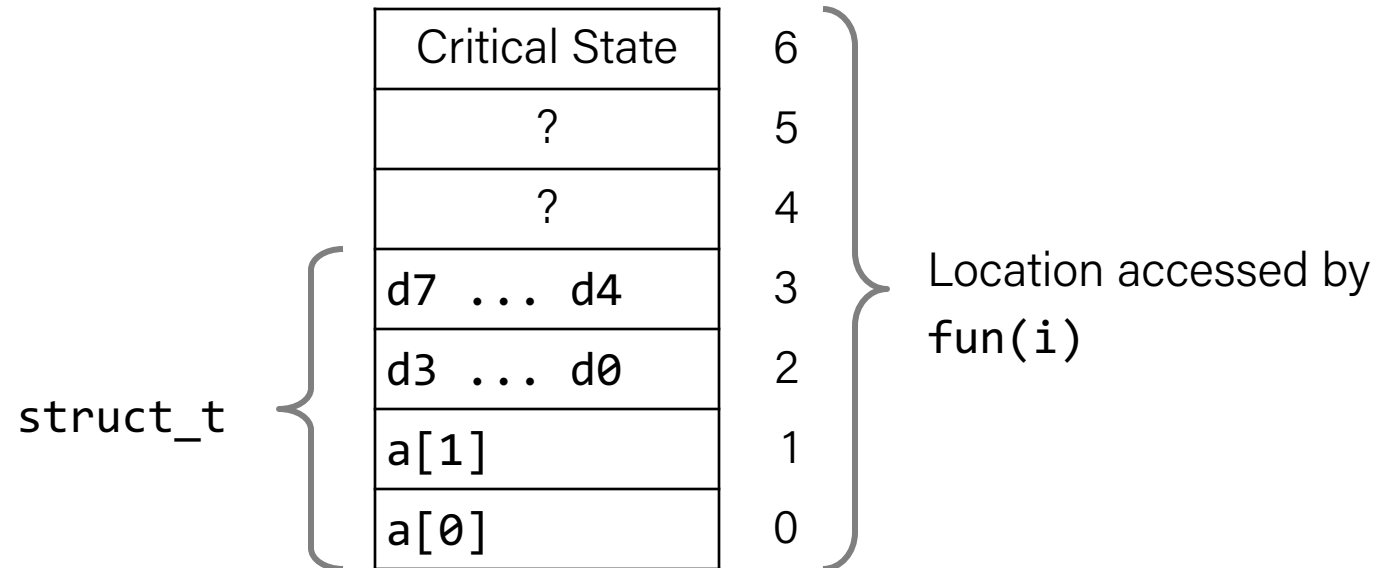
**Result is system specific**

# Memory Referencing Bug Example

```
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
    - #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

# 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

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

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

↑  
btw, how big  
is big enough?

```
unix>./bufdemo-nsp
Type a string:
012345678901234567890123
012345678901234567890123
```

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

# Buffer Overflow Disassembly

## echo:

00000000004006cf <echo>:

4006cf: 48 83 ec 18

4006d3: 48 89 e7

4006d6: e8 a5 ff ff ff

4006db: 48 89 e7

4006de: e8 3d fe ff ff

4006e3: 48 83 c4 18

4006e7: c3

sub \$0x18,%rsp

mov %rsp,%rdi

callq 400680 <gets>

mov %rsp,%rdi

callq 400520 <puts@plt>

add \$0x18,%rsp

retq

---

## call\_echo:

4006e8: 48 83 ec 08

4006ec: b8 00 00 00 00

4006f1: e8 d9 ff ff ff

4006f6: 48 83 c4 08

4006fa: c3

sub \$0x8,%rsp

mov \$0x0,%eax

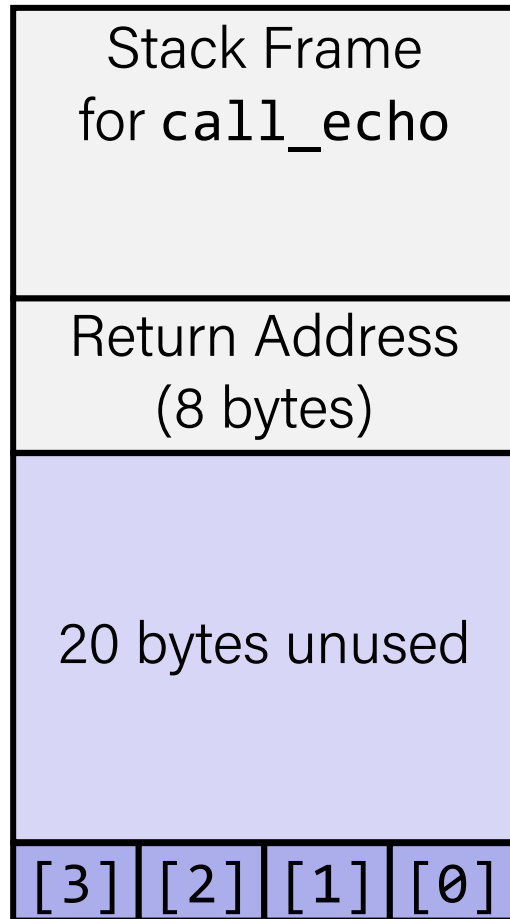
callq 4006cf <echo>

add \$0x8,%rsp

retq

# Buffer Overflow Stack

Before call to gets



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

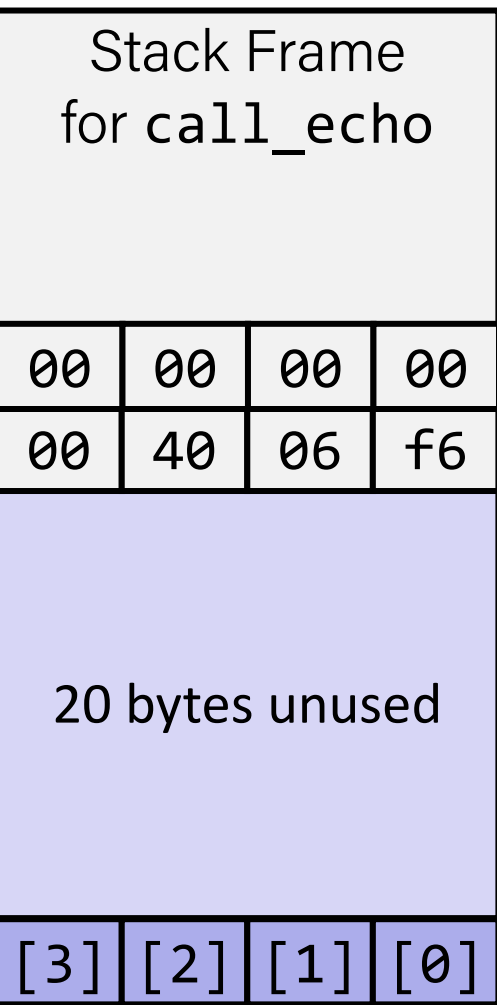
---

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



# Buffer Overflow Stack Example

Before call to gets



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  
    . . .
```

# 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:
01234567890123456789012
01234567890123456789012
```

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	29	28
27	26	25	24
23	22	21	20

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-nspunix>./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:
012345678901234567890123
012345678901234567890123
```

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

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

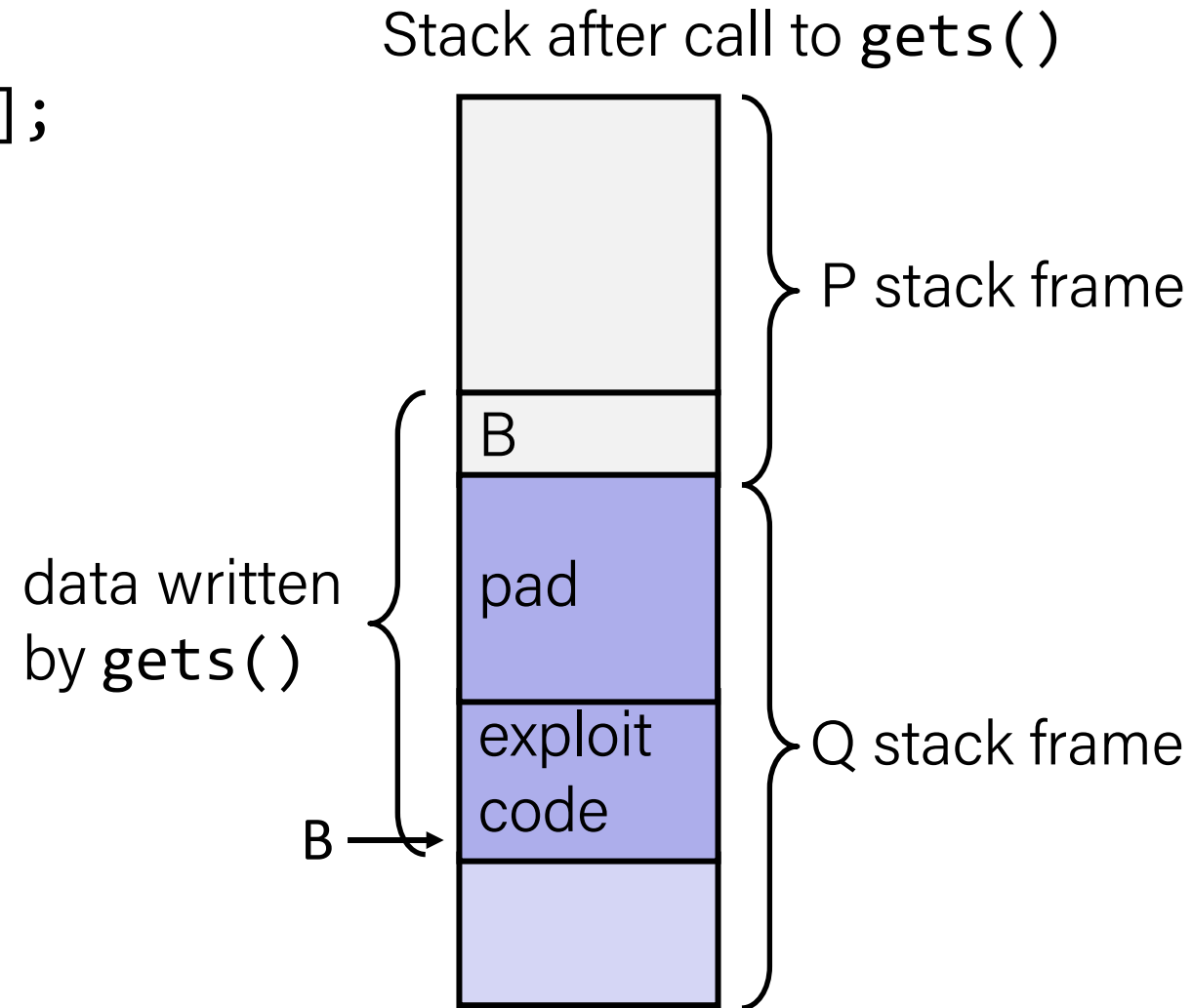
Lots of things happen, without modifying critical state  
Eventually executes retq back to main

buf ← %rsp

# Code Injection Attacks

```
void P(){  
    Q();  
    ... ← return address A  
}  
  
int Q() {  
    char buf[64];  
    gets(buf);  
    ...  
    return ...;  
}
```

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes `ret`, will jump to exploit code



# Exploits Based on Buffer Overflows

- **Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines**
- Distressingly common in real programs
  - Programmers keep making the same mistakes 😞
  - Recent measures make these attacks much more difficult
- Examples across the decades
  - Original "Internet worm" (1988)
  - "IM wars" (1999)
  - Twilight hack on Wii (2000s)
  - ... and many, many more
- You will learn some of the tricks in Assignment 5
  - Hopefully to convince you to never leave such holes in your programs!!

# Example: the original Internet worm (1988)

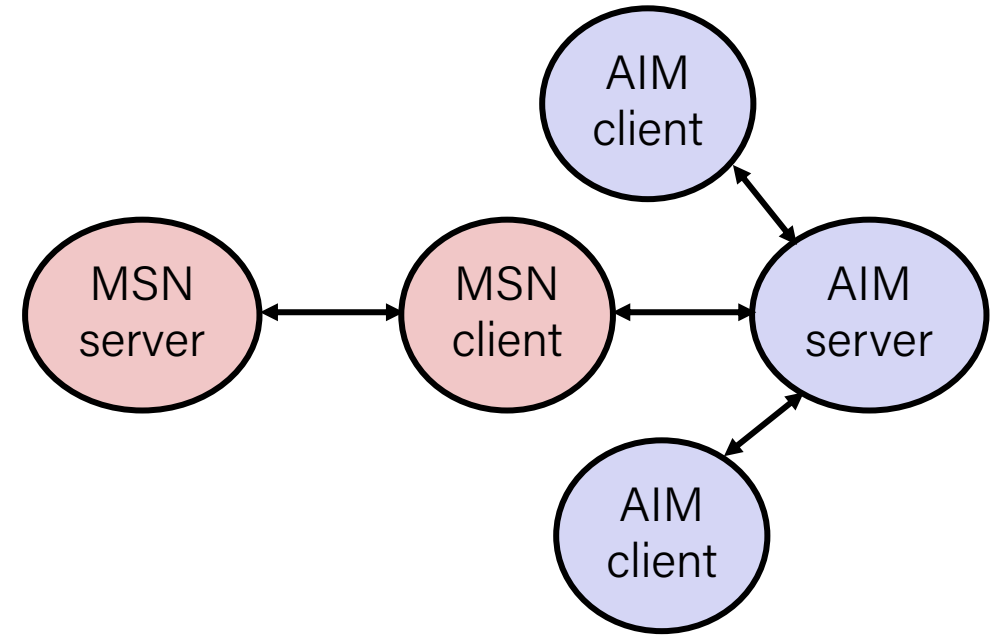
- Exploited a few vulnerabilities to spread
  - Early versions of the finger server (fingerd) used `gets()` to read the argument sent by the client:
    - `finger droh@linuxpool.ku.edu.tr`
  - Worm attacked fingerd server by sending phony argument:
    - `finger "exploit-code padding new-return-address"`
    - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.
- Once on a machine, scanned for other machines to attack
  - invaded ~6000 computers in hours (10% of the Internet 😊)
    - see June 1989 article in Comm. of the ACM
  - the young author of the worm was prosecuted...
  - and CERT was formed... homed at CMU



# Example 2: IM War

## July 1999:

- Microsoft launches MSN Messenger (instant messaging system).
- Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



# IM War (cont.)

## **August 1999:**

- Mysteriously, Messenger clients can no longer access AIM servers
- Microsoft and AOL begin the IM war:
  - AOL changes server to disallow Messenger clients
  - Microsoft makes changes to clients to defeat AOL changes
  - At least 13 such skirmishes
- What was really happening?
  - AOL had discovered a buffer overflow bug in their own AIM clients
  - They exploited it to detect and block Microsoft: the exploit code returned a 4-byte signature (the bytes at some location in the AIM client) to server
  - When Microsoft changed code to match signature, AOL changed signature location

Date: Wed, 11 Aug 1999 11:30:57 -0700 (PDT)  
From: Phil Bucking <philbucking@yahoo.com>  
Subject: AOL exploiting buffer overrun bug in their own software!  
To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you might find interesting because you are an Internet security expert with experience in this area. I have also tried to contact AOL but received no response.

I am a developer who has been working on a revolutionary new instant messaging client that should be released later this year.

...

It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now \*exploiting their own buffer overrun bug\* to help in its efforts to block MS Instant Messenger.

....

Since you have significant credibility with the press I hope that you can use this information to help inform people that behind AOL's friendly exterior they are nefariously compromising peoples' security.

Sincerely,  
Phil Bucking  
Founder, Bucking Consulting  
philbucking@yahoo.com

***It was later determined  
that this email originated  
from within Microsoft!***

# Aside: Worms and Viruses

- **Worm:** A program that
  - Can run by itself
  - Can propagate a fully working version of itself to other computers
- **Virus:** Code that
  - Adds itself to other programs
  - Does not run independently
- Both are (usually) designed to spread among computers and to wreak havoc

# OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”

# 1. Avoid Overflow Vulnerabilities in Code (!)

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

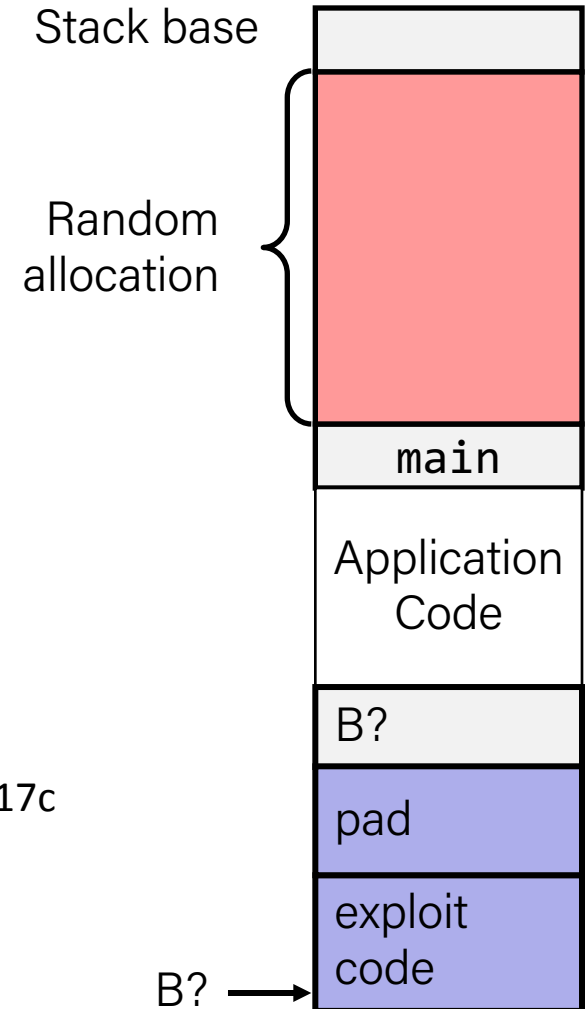
For example, use library routines that limit string lengths

- `fgets` instead of `gets`
- `strncpy` instead of `strcpy`
- Don't use `scanf` with `%s` conversion specification
  - Use `fgets` to read the string
  - Or use `%ns` where `n` is a suitable integer

## 2. System-Level Protections can help

### Randomized stack offsets

- At start of program, allocate random amount of space on stack
- Shifts stack addresses for entire program
- Makes it difficult for hacker to predict beginning of inserted code
- E.g.: 5 executions of memory allocation code



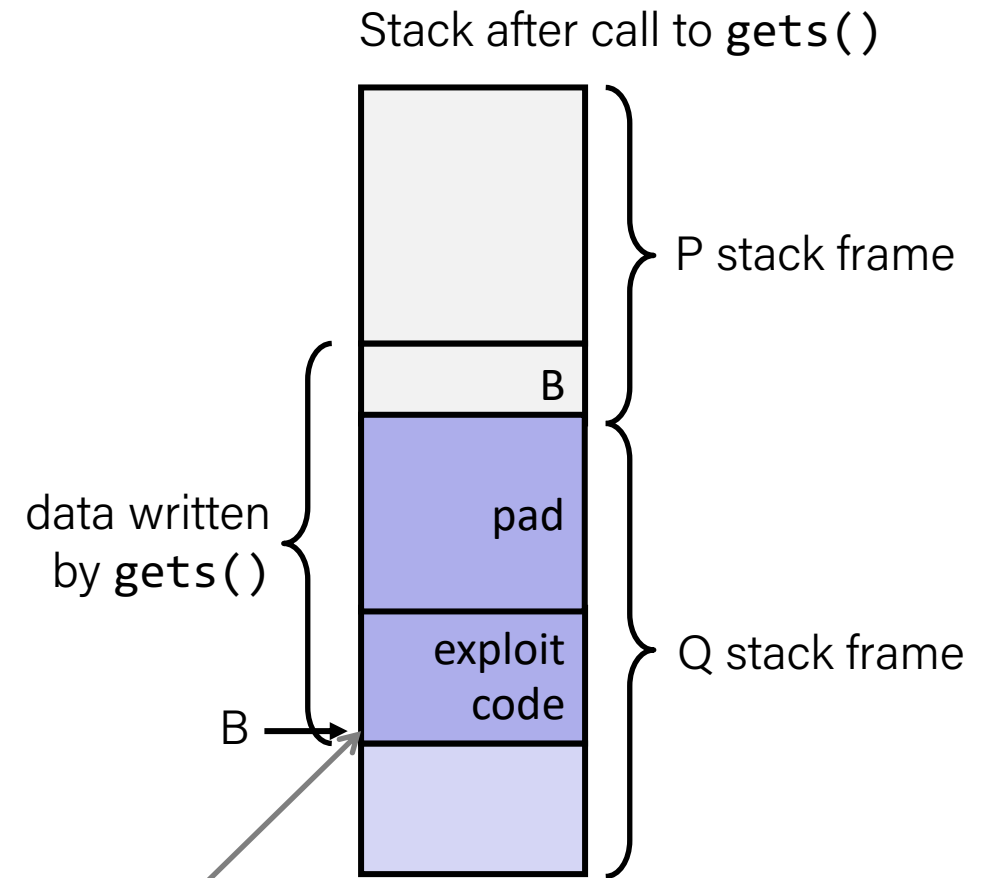
local            0x7ffe4d3be87c    0x7fff75a4f9fc    0x7ffeadb7c80c    0x7ffeaea2fdac    0x7ffcd452017c

- Stack repositioned each time program executes

## 2. System-Level Protections can help

### Nonexecutable code segments

- In traditional x86, can mark region of memory as either “read-only” or “writable”
  - Can execute anything readable
- X86-64 added explicit “execute” permission
- Stack marked as non-executable



Any attempt to execute this code will fail



# 3. Stack Canaries can help

## Idea:

- Place special value ("canary") on stack just beyond buffer
- Check for corruption before exiting function

## GCC Implementation

- -fstack-protector
- Now the default (disabled earlier)

```
unix>./bufdemo-sp  
Type a string:  
0123456  
0123456
```

```
unix>./bufdemo-sp  
Type a string:  
01234567  
*** stack smashing detected ***
```

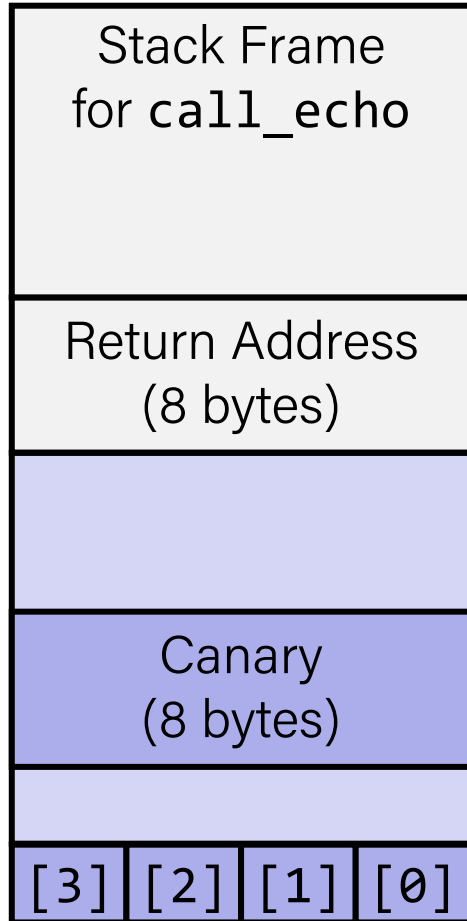
# Protected Buffer Disassembly

echo:

```
40072f: sub    $0x18,%rsp
400733: mov    %fs:0x28,%rax
40073c: mov    %rax,0x8(%rsp)
400741: xor    %eax,%eax
400743: mov    %rsp,%rdi
400746: callq  4006e0 <gets>
40074b: mov    %rsp,%rdi
40074e: callq  400570 <puts@plt>
400753: mov    0x8(%rsp),%rax
400758: xor    %fs:0x28,%rax
400761: je     400768 <echo+0x39>
400763: callq  400580 <__stack_chk_fail@plt>
400768: add    $0x18,%rsp
40076c: retq
```

# Setting Up Canary

**Before call to gets**



```
/* Echo Line */
```

```
void echo()
```

```
{
```

```
    char buf[4]; /* Way too small! */
```

```
    gets(buf);
```

```
    puts(buf);
```

```
}
```

---

```
echo:
```

```
    . . .
```

```
    movq    %fs:40, %rax    # Get canary
```

```
    movq    %rax, 8(%rsp)  # Place on stack
```

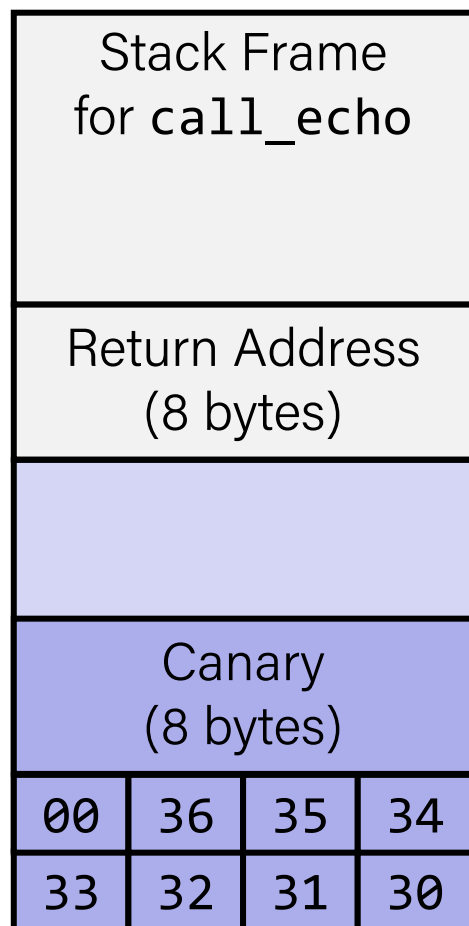
```
    xorl    %eax, %eax     # Erase canary
```

```
    . . .
```

buf ← %rsp

# Checking Canary

After call to gets



```
/* Echo Line */
```

```
void echo()
```

```
{
```

```
    char buf[4]; /* Way too small! */
```

```
    gets(buf);
```

```
    puts(buf);
```

```
}
```

---

**Input:** 0123456

---

```
echo:
```

```
    . . .
```

```
    movq    8(%rsp), %rax    # Retrieve from stack
```

```
    xorq    %fs:40, %rax    # Compare to canary
```

```
    je      .L6             # If same, OK
```

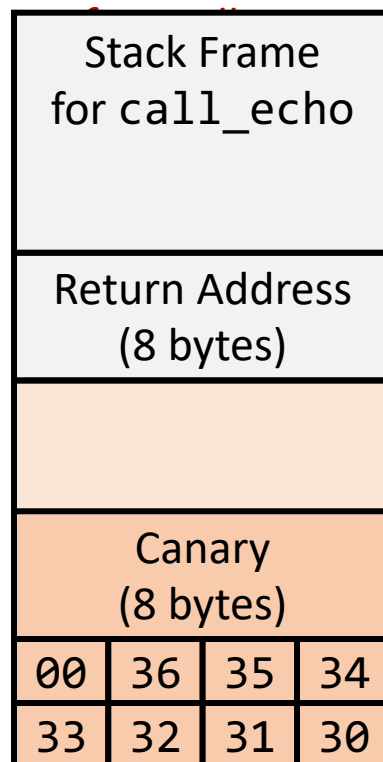
```
    call    __stack_chk_fail # FAIL
```

```
.L6:    . . .
```

buf ← %rsp

# Checking Canary

*After call to gets*



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

Input: 0123456

buf ← %rsp

echo:

```
    . . .
    movq    8(%rsp), %rax    # Retrieve from stack
    xorq    %fs:40, %rax    # Compare to canary
    je      .L6             # If same, OK
    call    __stack_chk_fail # FAIL
.L6: . . .
```

# Return-Oriented Programming Attacks

- Challenge (for hackers)
  - Stack randomization makes it hard to predict buffer location
  - Marking stack nonexecutable makes it hard to insert binary code
- Alternative Strategy
  - Use existing code
    - E.g., library code from `stdlib`
  - String together fragments to achieve overall desired outcome
  - *Does not overcome stack canaries*
- Construct program from gadgets
  - Sequence of instructions ending in `ret`
    - Encoded by single byte `0xc3`
  - Code positions fixed from run to run
  - Code is executable

# Gadget Example #1

```
long ab_plus_c  
  (long a, long b, long c) {  
    return a*b + c;  
  }
```

---

00000000004004d0 <ab\_plus\_c>:

4004d0: 48 0f af fe imul %rsi,%rdi

4004d4: 48 8d 04 17 lea (%rdi,%rdx,1),%rax

4004d8: c3 retq



$\text{rax} \leftarrow \text{rdi} + \text{rdx}$

Gadget address = 0x4004d4

- Use tail end of existing functions

# Gadget Example #2

```
void setval(unsigned *p) {  
    *p = 3347663060u;  
}
```

Encodes `movq %rax, %rdi`

```
<setval>:  
4004d9: c7 07 d4 48 89 c7 movl $0xc78948d4, (%rdi)  
4004df: c3 retq
```

`rdi ← rax`

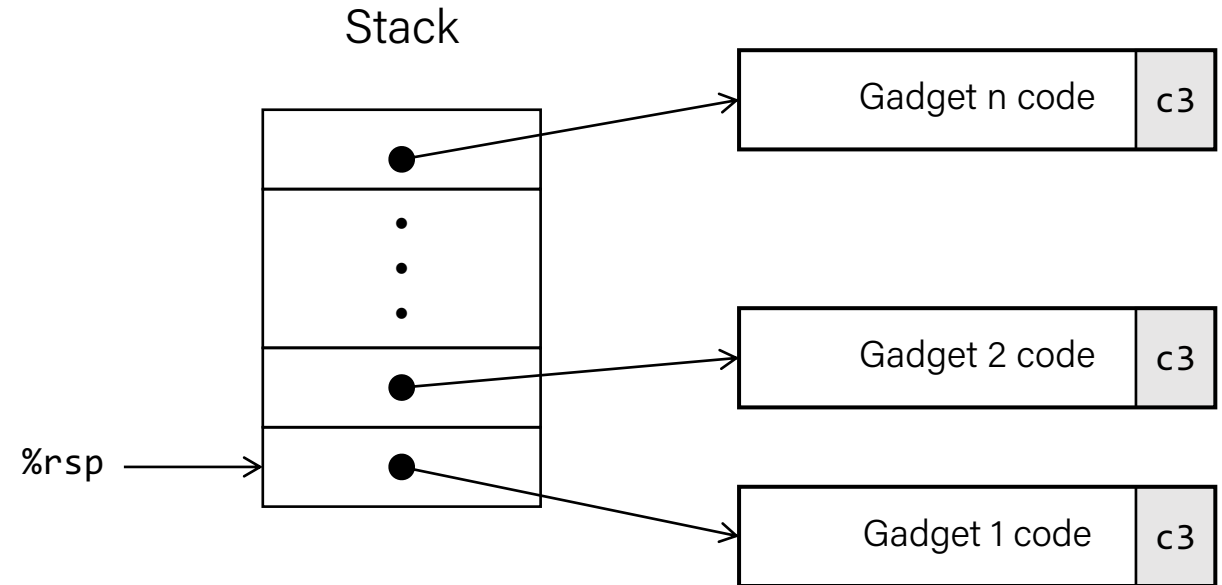
Gadget address = `0x4004dc`

- Repurpose byte codes



# ROP Execution

- Trigger with `ret` instruction
  - Will start executing Gadget 1
- Final `ret` in each gadget will start next one



# Recap

- Structures and Alignment
- Floating Point
- Memory Layout
- Buffer Overflow

**Next time:** memory hierarchy