

# Machine-Level Programming V: Advanced Topics

COMP400727: Introduction to Computer Systems

Danfeng Shan  
Xi'an Jiaotong University

# Today

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection
  - Bypassing Protection

# x86-64 Linux Memory Layout

*not drawn to scale*

$(2^{47} - 4096 =)$  0000 7FFF FFFF F000

## ■ Stack

- Runtime stack (8MB limit)
- e.g., local variables

## ■ Heap

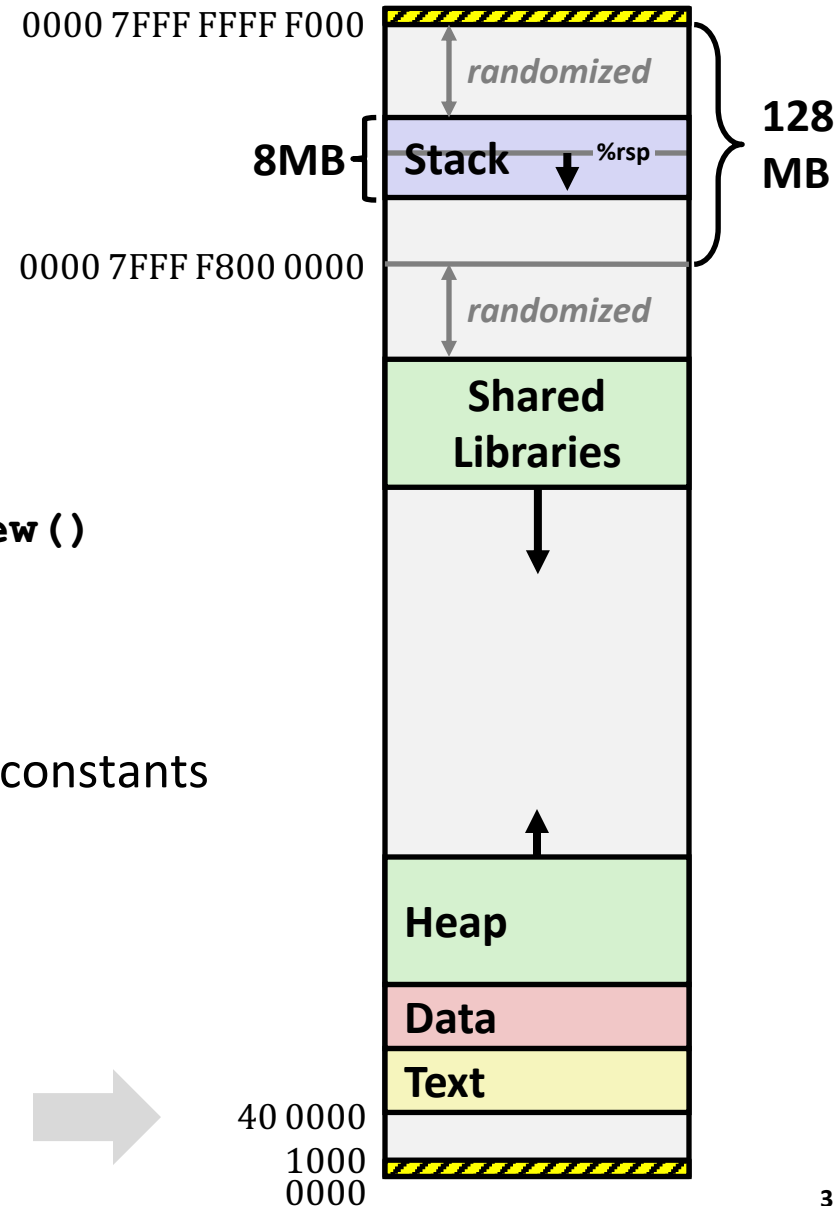
- Dynamically allocated as needed
- When call `malloc()`, `calloc()`, `new()`

## ■ Data

- Statically allocated data
- e.g., global vars, `static` vars, string constants

## ■ Text / Shared Libraries

- Executable machine instructions
- Read-only



*not drawn to scale*

# Memory Allocation Example

0000 7FFF FFFF F000

```

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

int global = 0;

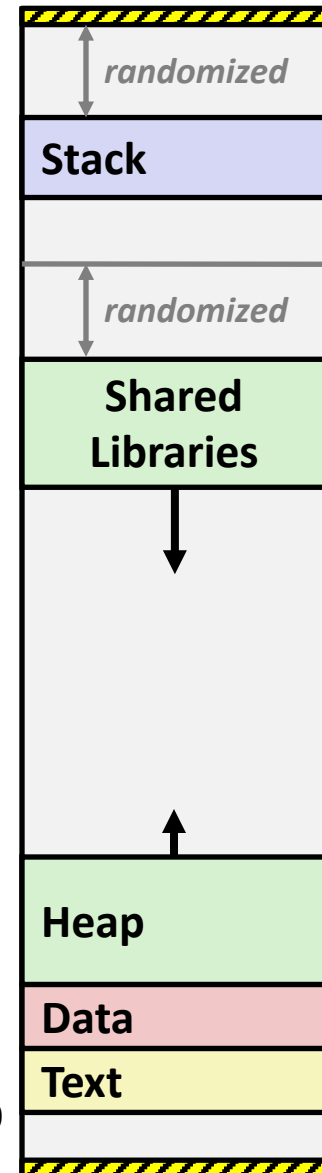
int useless() { return 0; }

int main ()
{
    void *phuge1, *psmall2, *phuge3, *psmall4;
    int local = 0;
    phuge1 = malloc(1L << 28);  /* 256 MB */
    psmall2 = malloc(1L << 8);  /* 256 B */
    phuge3 = malloc(1L << 32);  /* 4 GB */
    psmall4 = malloc(1L << 8);  /* 256 B */
    /* Some print statements ... */
}

```

40 0000

*Where does everything go?*



*not drawn to scale*

# x86-64 Example Addresses

0000 7FFF FFFF F000

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

```

local
phuge1
phuge3
psmall4
psmall2
big_array
huge_array
main()
useless()

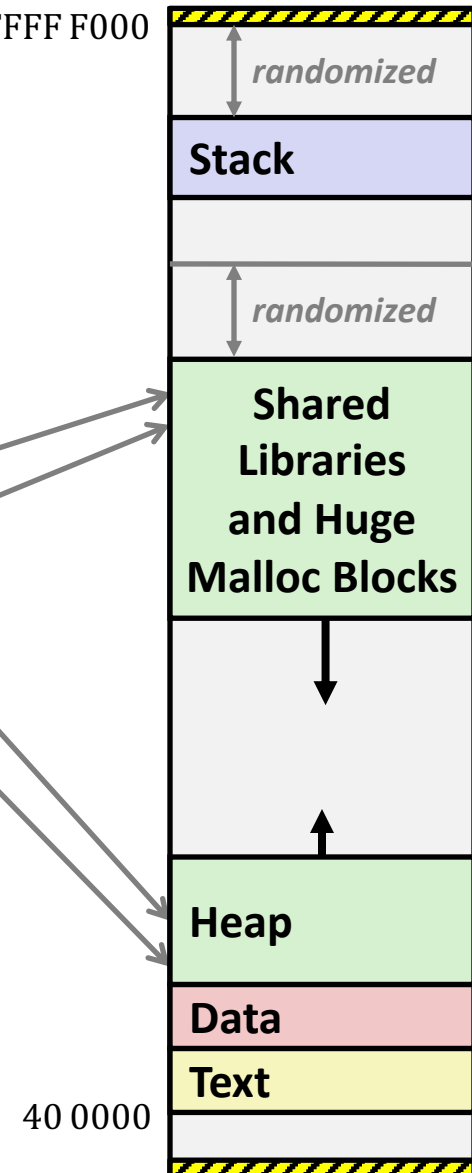
```

```

0x00007ffe4d3be87c
0x00007f7262a1e010
0x00007f7162a1d010
0x000000008359d120
0x000000008359d010
0x0000000080601060
0x0000000000601060
0x000000000040060c
0x0000000000400590

```

(Exact values can vary)



# Today

- Memory Layout
- **Buffer Overflow**
  - Vulnerability
  - Protection
  - Bypassing 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;  
}
```

<b>fun(0)</b>	<b>-&gt;</b>	<b>3.1400000000</b>
<b>fun(1)</b>	<b>-&gt;</b>	<b>3.1400000000</b>
<b>fun(2)</b>	<b>-&gt;</b>	<b>3.1399998665</b>
<b>fun(3)</b>	<b>-&gt;</b>	<b>2.0000006104</b>
<b>fun(6)</b>	<b>-&gt;</b>	<b>Stack smashing detected</b>
<b>fun(8)</b>	<b>-&gt;</b>	<b>Segmentation fault</b>

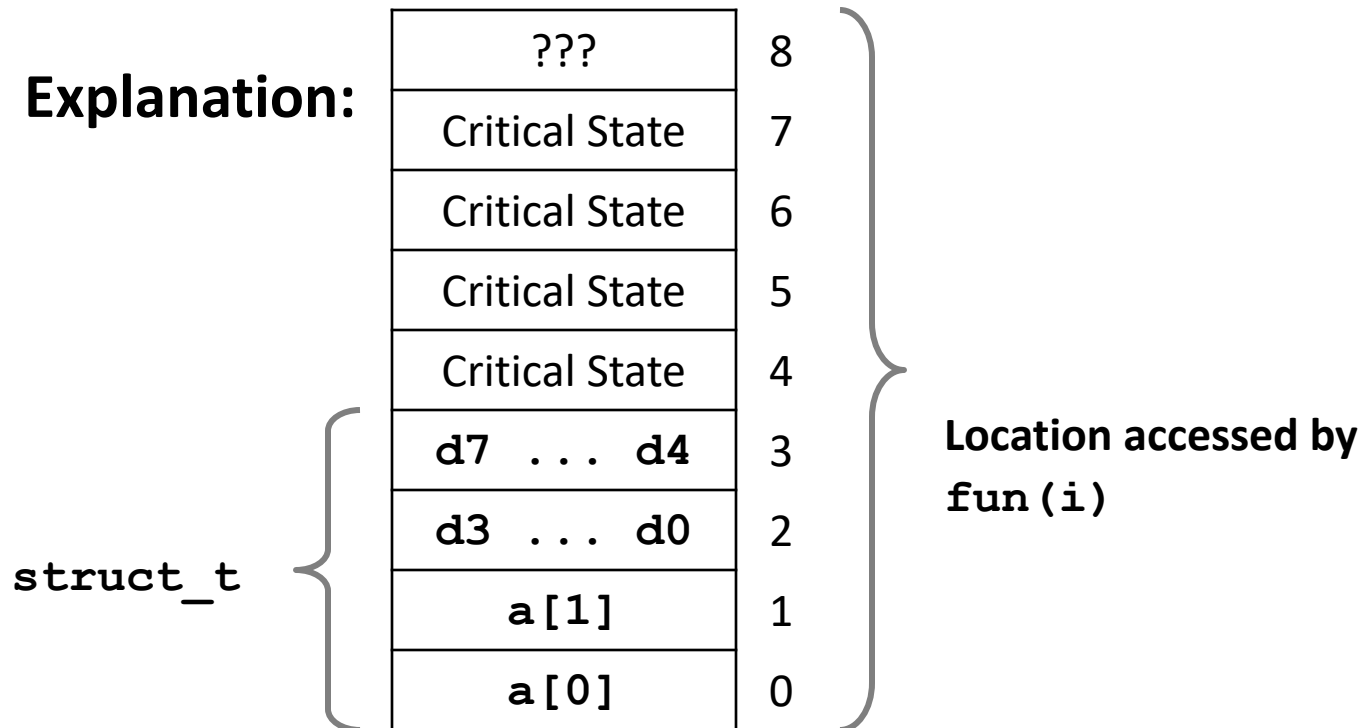
- Result is system specific

# Memory Referencing Bug Example

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

fun(0)	->	3.1400000000
fun(1)	->	3.1400000000
fun(2)	->	3.1399998665
fun(3)	->	2.0000006104
fun(4)	->	Segmentation fault
fun(8)	->	3.1400000000

**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
- **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);  
}
```

← BTW, how big  
is big enough?

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

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

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

# Buffer Overflow Disassembly

echo:

000000000040069c <echo>:

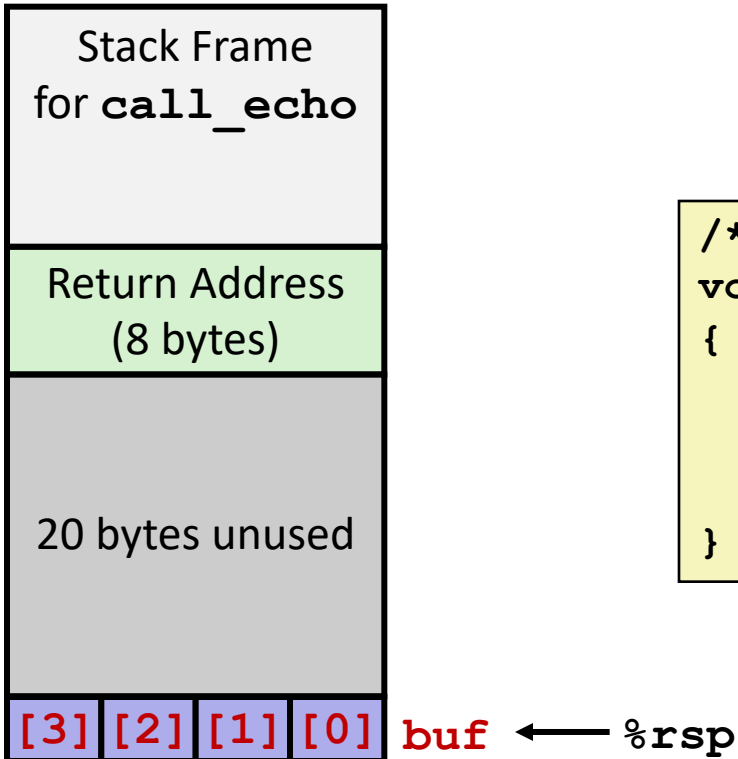
40069c:	48 83 ec 18	sub	<b>\$0x18</b> , %rsp
4006a0:	48 89 e7	mov	<b>%rsp</b> , %rdi
4006a3:	e8 a5 ff ff ff	callq	40064d <gets>
4006a8:	48 89 e7	mov	%rsp, %rdi
4006ab:	e8 50 fe ff ff	callq	400500 <puts@plt>
4006b0:	48 83 c4 18	add	\$0x18, %rsp
4006b4:	c3	retq	

call\_echo:

4006b5:	48 83 ec 08	sub	\$0x8, %rsp
4006b9:	b8 00 00 00 00	mov	\$0x0, %eax
4006be:	e8 d9 ff ff ff	callq	40069c <echo>
<b>4006c3:</b>	48 83 c4 08	add	\$0x8, %rsp
4006c7:	c3	retq	

# Buffer Overflow Stack Example

*Before call to gets*

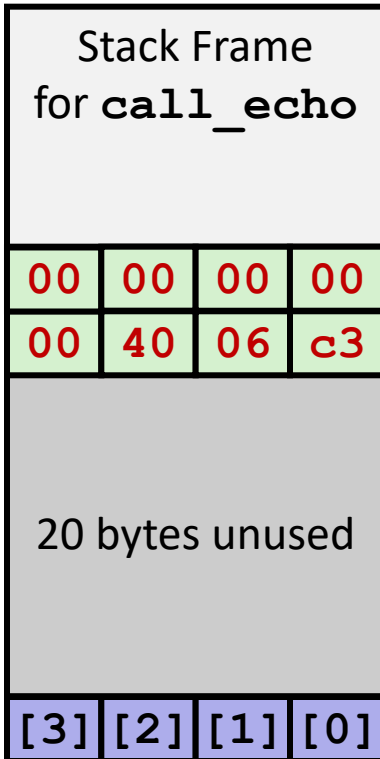


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    subq $0x18, %rsp
    movq %rsp, %rdi
    call gets
    . . .
```

# Buffer Overflow Stack Example

*Before call to gets*



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

```
echo:
    subq    $0x18, %rsp
    movq    %rsp, %rdi
    call    gets
    . . .
```

`call_echo:`

```
. . .
4006be: callq    4006cf <echo>
4006c3: add     $0x8,%rsp
. . .
```

# Buffer Overflow Stack Example #1

*After call to gets*

Stack Frame for <code>call_echo</code>			
00	00	00	00
00	40	06	c3
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    $0x18, %rsp
    movq    %rsp, %rdi
    call    gets
    . . .
```

`call_echo:`

```
. . .
4006be:  callq    4006cf <echo>
4006c3:  add      $0x8, %rsp
. . .
```

```
unix> ./bufdemo-nsp
Type a string: 01234567890123456789012
01234567890123456789012
```

```
"01234567890123456789012\0"
```

**Overflowed buffer, but did not corrupt state**

# Buffer Overflow Stack Example #2

*After call to gets*

Stack Frame for <code>call_echo</code>			
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    $0x18, %rsp
    movq    %rsp, %rdi
    call    gets
    . . .
```

`call_echo:`

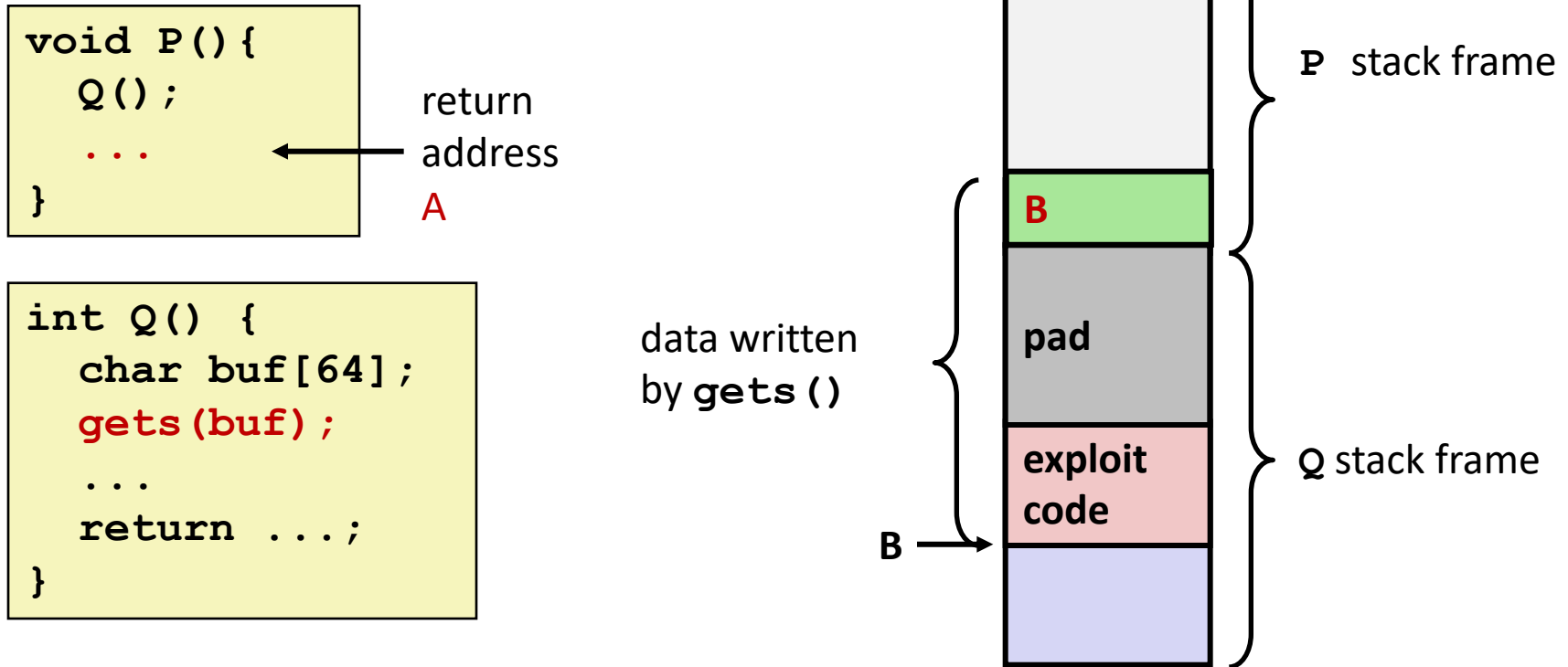
```
. . .
4006be:    callq    4006cf <echo>
4006c3:    add     $0x8,%rsp
. . .
```

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

Program “returned” to 0x0400600, and then crashed.

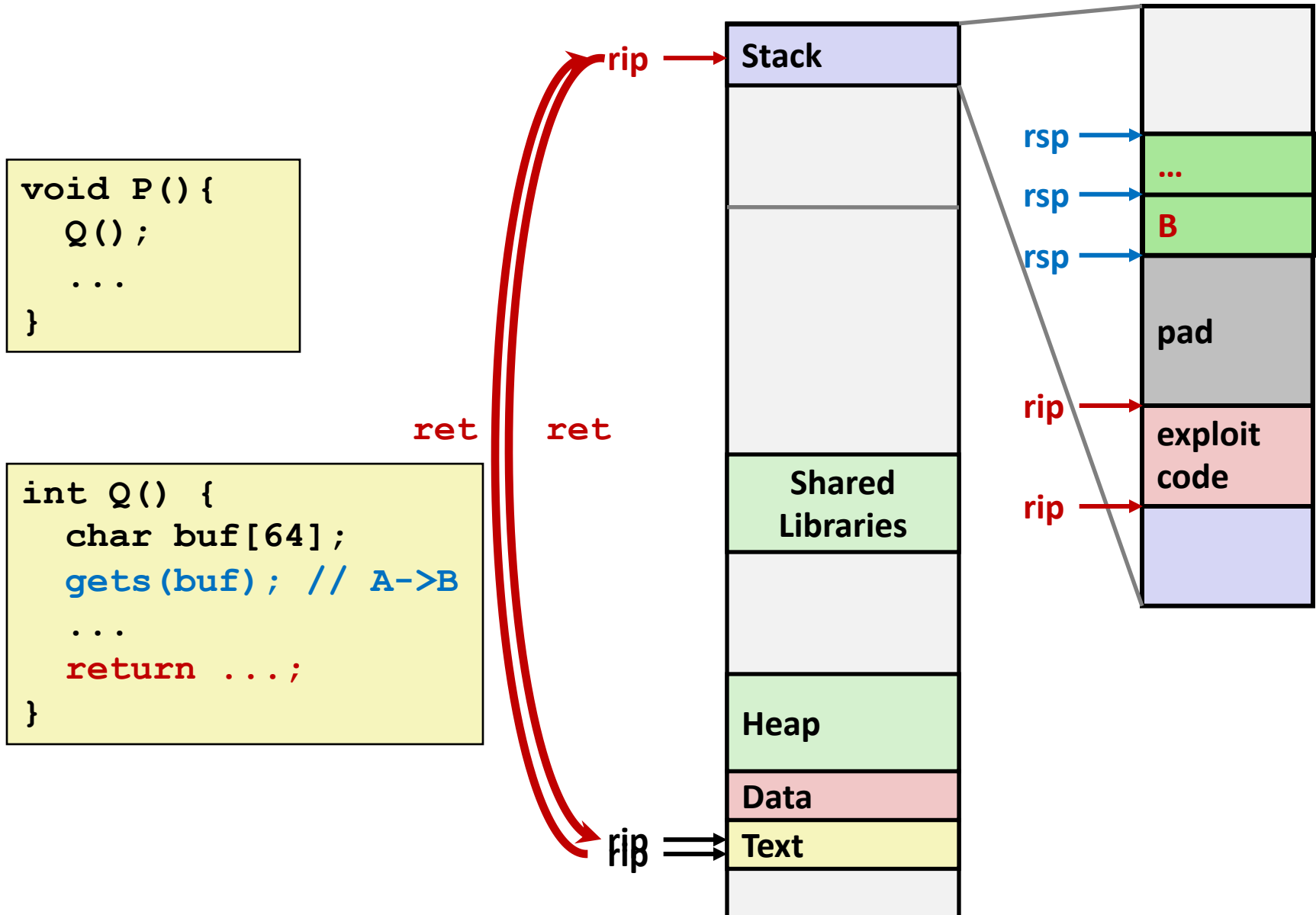


# Code Injection Attacks



- 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

# How Does The Attack Code Execute?



# 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 attacklab**~~
  - ~~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 dfshan@cs.xjtu.edu.cn`
- 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.

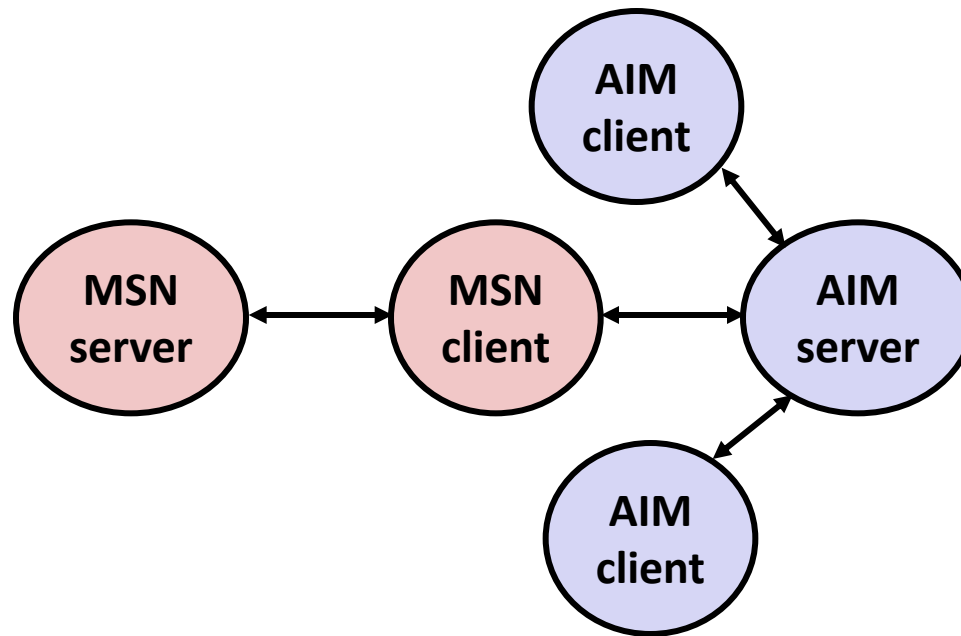
## ■ Once on a machine, scanned for other machines to attack

- 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... still 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: Trojans, Worms, Viruses, ...

- **Three different kinds of *malware* (malicious software)**
  - Categorized by how they spread
  - Lines have gotten fuzzier over time
- **A *trojan* tricks people into running it**
  - Named after the legend of the Trojan Horse
- **A *worm* spreads automatically, without human action**
  - Requires a way to copy and execute itself over the network
- **A *virus* takes control of programs that are already installed**
  - Like a biological virus



# Today

- Memory Layout
- **Buffer Overflow**
  - Vulnerability
  - Protection
  - Bypassing Protection

# What to Do About Buffer Overflow Attacks

- **Avoid overflow vulnerabilities**
- **Employ system-level protections**
- **Have compiler use “stack canaries”**
- **Lets talk about each...**

# 1. Avoid Overflow Vulnerabilities in Code (!)

```
/* Echo Line */  
void echo()  
{  
    char buf[4];  
    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

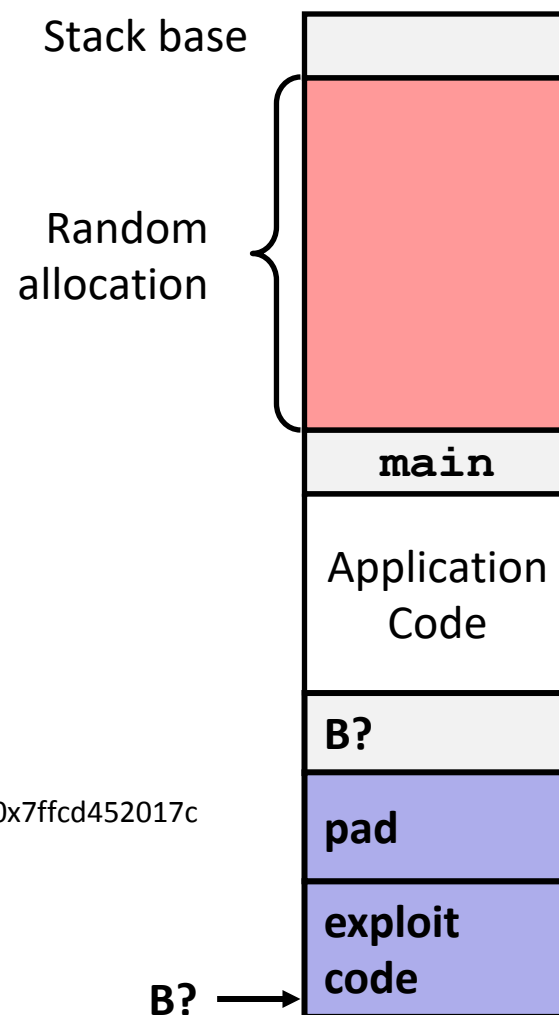
## 2. System-Level Protections Can Help

### ■ 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

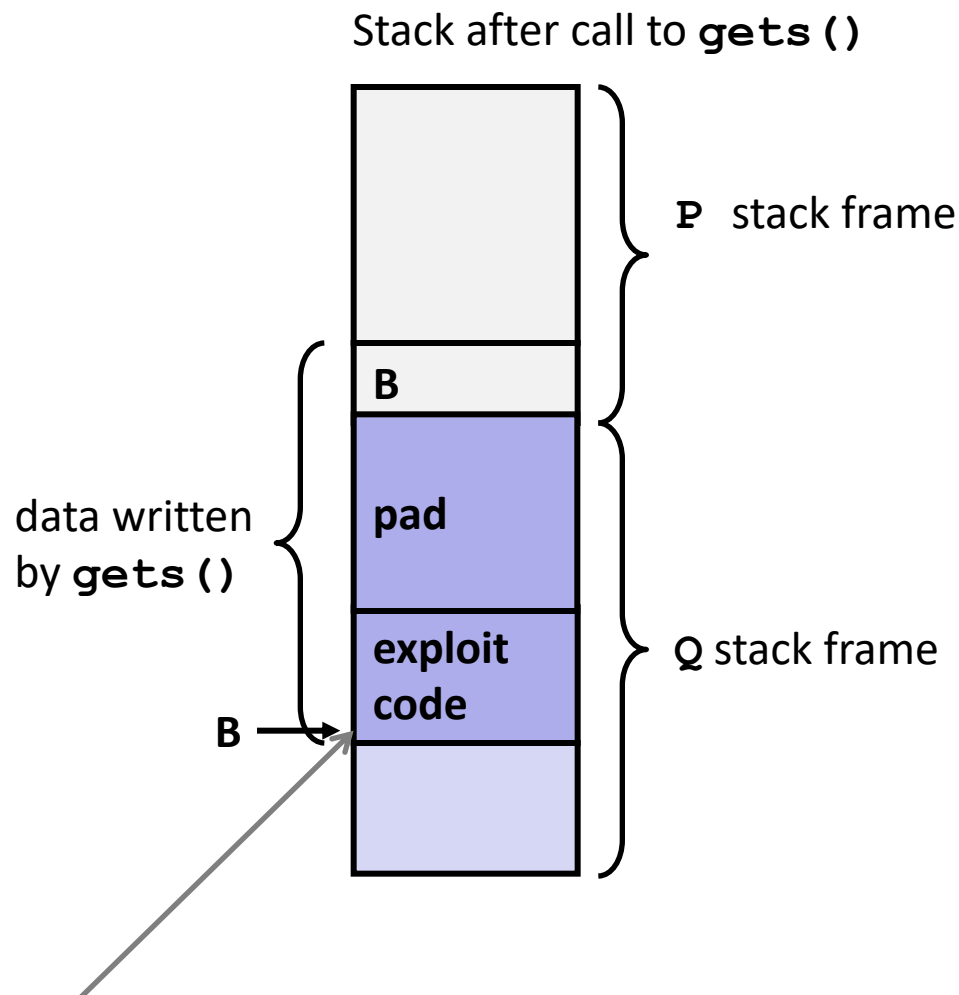
- Stack repositioned each time program executes



## 2. System-Level Protections Can Help

### ■ Non-executable memory

- Older x86 CPUs would execute machine code from any readable address
- x86-64 added a way to mark regions of memory as *not executable*
- Immediate crash on jumping into any such region
- Current Linux and Windows mark the stack this way



**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:012345678  
*** stack smashing detected ***
```

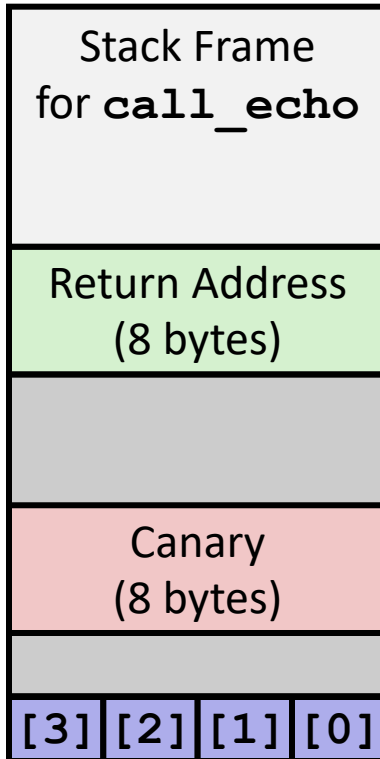
# Protected Buffer Disassembly

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*



`buf ← %rsp`

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

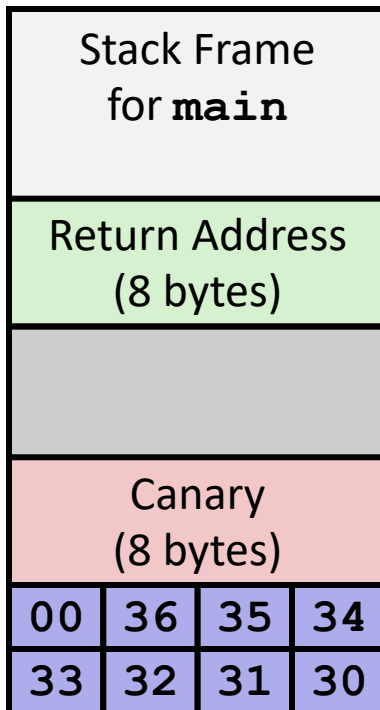
`echo:`

```
. . .
mov     %fs:0x28, %rax    # Get canary
mov     %rax, 0x8(%rsp)  # Place on stack
xor     %eax, %eax       # Erase register
. . .
```



# Checking Canary

*After call to gets*



`buf ← %rsp`

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

Input: *0123456*

*Some systems:*

*LSB of canary is 0x00*

*Allows input 01234567*

`echo:`

`. . .`

```
mov     0x8(%rsp),%rax    # Retrieve from stack
xor     %fs:0x28,%rax     # Compare to canary
je      .L6              # If same, OK
call    __stack_chk_fail # FAIL
```

# Return-Oriented Programming Attacks

## ■ Challenge (for hackers)

- Stack randomization makes it hard to predict buffer location
- Marking stack non-executable makes it hard to insert binary code

## ■ Alternative Strategy

- Use existing code
  - Part of the program or the C library
- 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
```

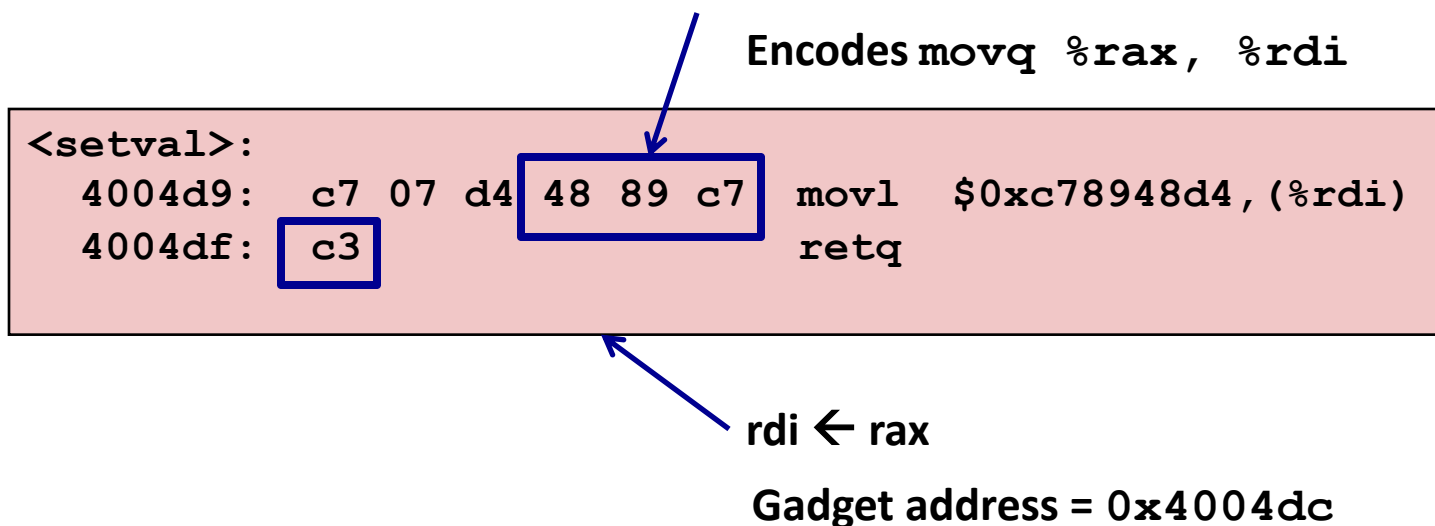
$\text{rax} \leftarrow \text{rdi} + \text{rdx}$

Gadget address = 0x4004d4

- Use tail end of existing functions

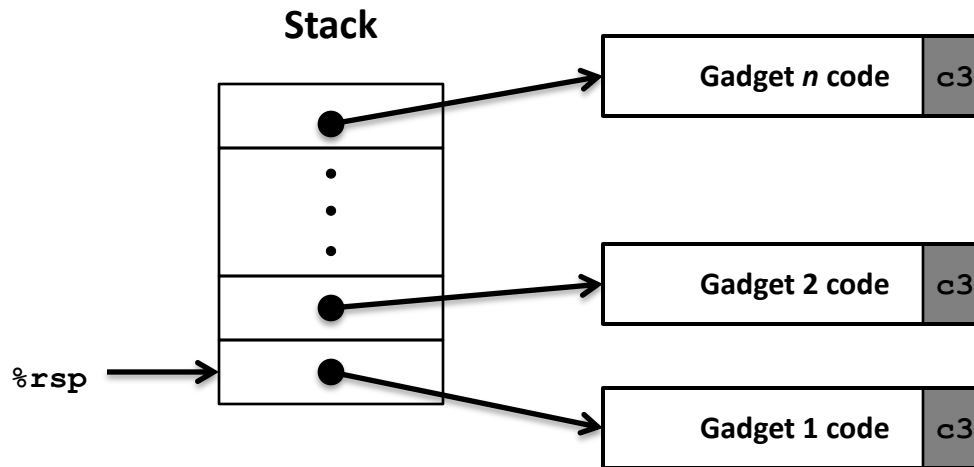
# Gadget Example #2

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



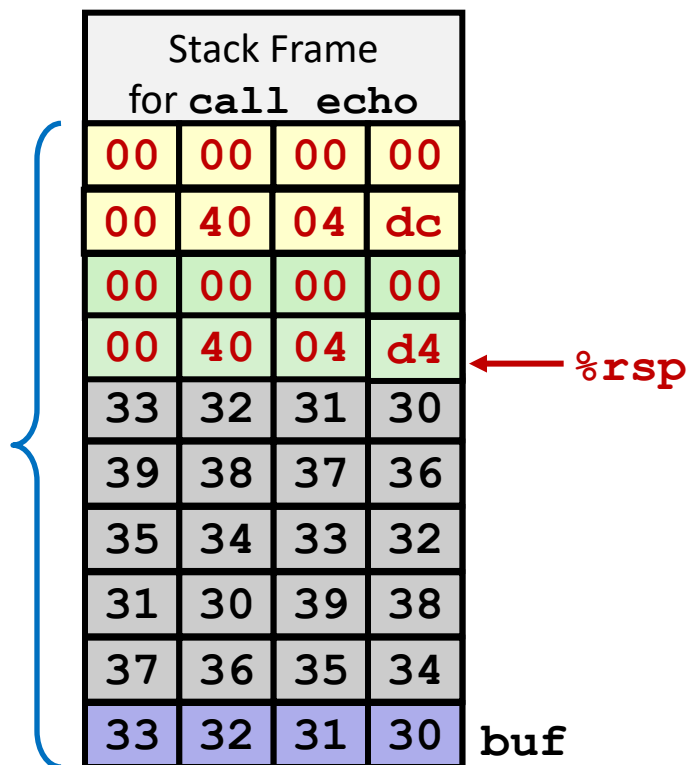
- Repurpose byte codes

# ROP Execution



- **Trigger with `ret` instruction**
  - Will start executing Gadget 1
- **Final `ret` in each gadget will start next one**
  - `ret`: pop address from stack and jump to that address

# Crafting an ROP Attack String



## ■ Gadget #1

■ `0x4004d4`  $\text{rax} \leftarrow \text{rdi} + \text{rdx}$

## ■ Gadget #2

■ `0x4004dc`  $\text{rdi} \leftarrow \text{rax}$

## ■ Combination

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

## Attack String (Hex)

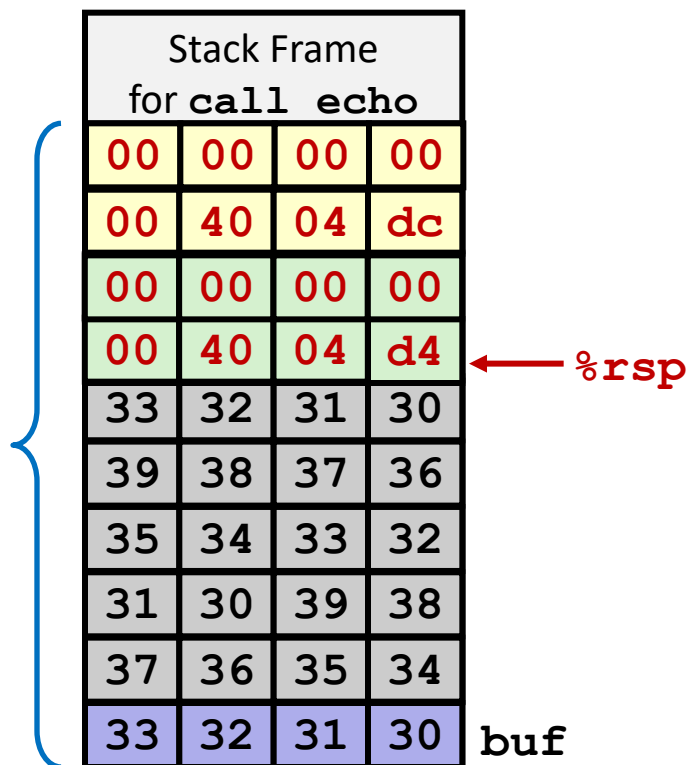
```

30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
d4 04 40 00 00 00 00 00 dc 04 40 00 00 00 00 00 00

```

Multiple gadgets will corrupt stack upwards

# What Happens When echo Returns?



1. Echo executes `ret`
  - Starts Gadget #1
2. Gadget #1 executes `ret`
  - Starts Gadget #2
3. Gadget #2 executes `ret`
  - Goes off somewhere ...

Multiple gadgets will corrupt stack upwards

# Today

- Memory Layout
- **Buffer Overflow**
  - Vulnerability
  - Protection
  - Bypassing Protection