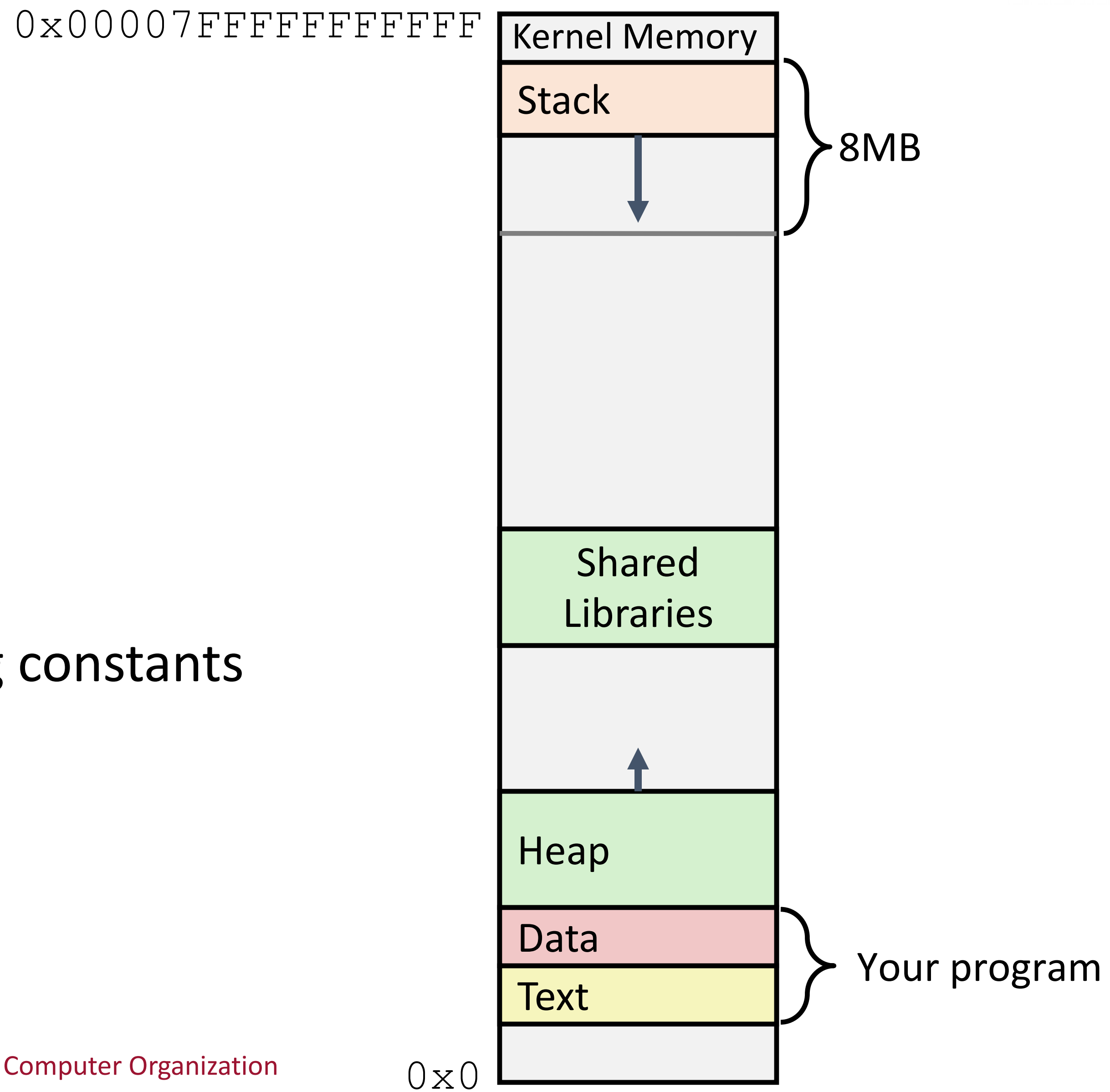
Machine-Level Programming: Buffer Overflow

CMPU 224 – Computer Organization
Jason Waterman

x86-64 Linux Memory Layout

not drawn to scale

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



String Library Code

- Implementation of Linux function `gets()`
  - `gets()` reads a line from `stdin` into the buffer pointed to by `s` until either a terminating newline or `EOF`, which it replaces with a null byte
  - 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

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

GETS(3) Linux Programmer's Manual GETS(3)

NAME

gets - get a string from standard input (DEPRECATED)

SYNOPSIS

```
#include <stdio.h>
char *gets(char *s);
```

DESCRIPTION

Never use this function.

gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or **EOF**, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

RETURN VALUE

gets() returns s on success, and NULL on error or when end of file occurs while no characters have been read. However, given the lack of buffer overrun checking, there can be no guarantees that the function will even return.

BUGS

Never use **gets**(). Because it is impossible to tell without knowing the data in advance how many characters **gets**() will read, and because **gets**() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use **fgets**() instead.

Vulnerable Buffer Code

```
#include <stdio.h>

// Compile with:
// gcc -Og -fno-stack-protector -o bufdemo bufdemo.c

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

void call_echo(void) {
  echo();
}

void main(void) {
  puts('Type a string:');
  call_echo();
}
```

← btw, how big is big enough?

```
Linux>./bufdemo
Type a string:
012
012
```

```
Linux>./bufdemo
Type a string:
01234567890123456789012
01234567890123456789012
```

```
Linux>./bufdemo
Type a string:
0123456789012345678901234
Segmentation fault
```

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

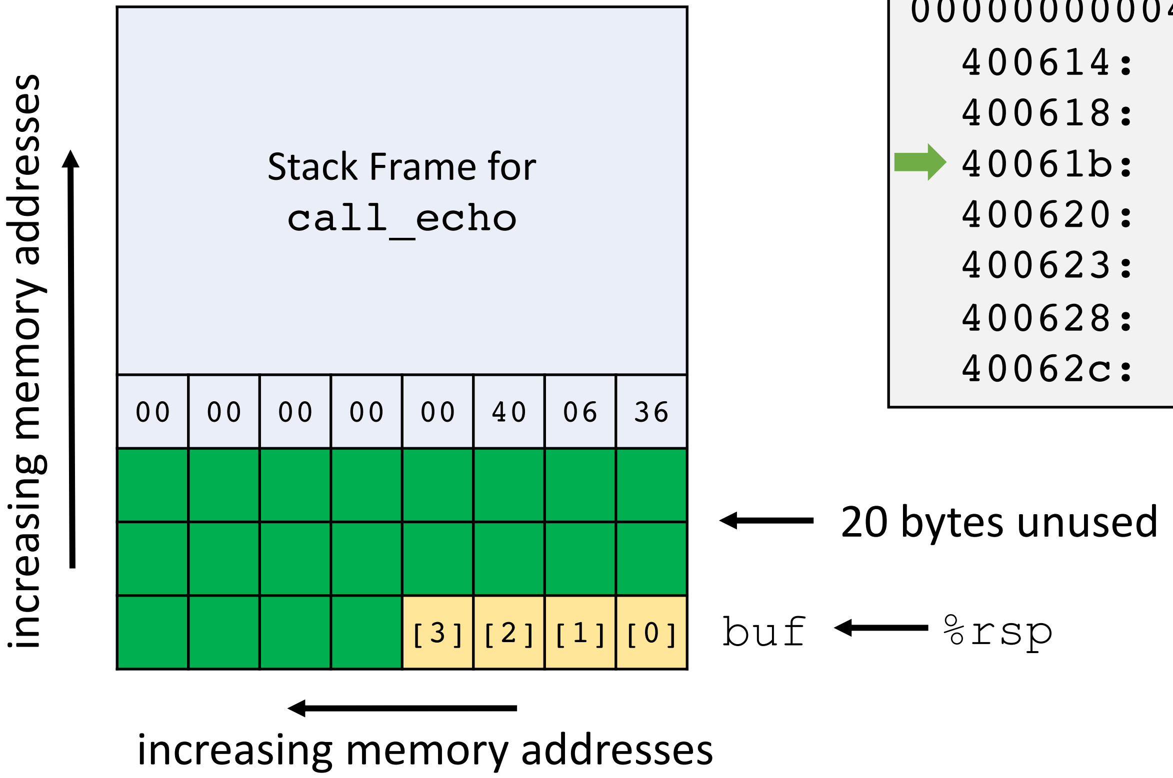
Buffer Overflow Disassembly

```
#include <stdio.h>

// Compile with:
// gcc -Og -fno-stack-protector –o \
   bufdemo bufdemo.c

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

void call_echo(void) {
  echo();
}

void main(void) {
  puts("Type a string:");
  call_echo();
}
```

```
0000000000400614 <echo>:
  400614:        48 83 ec 18        sub    $0x18,%rsp
  400618:        48 89 e7           mov    %rsp,%rdi
  40061b:        e8 a6 ff ff ff     callq  4005c6 <gets>
  400620:        48 89 e7           mov    %rsp,%rdi
  400623:        e8 68 fe ff ff     callq  400490 <puts@plt>
  400628:        48 83 c4 18        add    $0x18,%rsp
  40062c:        c3                 retq

000000000040062d <call_echo>:
  40062d:        48 83 ec 08        sub    $0x8,%rsp
  400631:        e8 de ff ff ff     callq  400614 <echo>
  400636:        48 83 c4 08        add    $0x8,%rsp
  40063a:        c3                 retq

000000000040063b <main>:
  40063b:        48 83 ec 08        sub    $0x8,%rsp
  40063f:        bf e4 06 40 00     mov    $0x4006e4,%edi
  400644:        e8 47 fe ff ff     callq  400490 <puts@plt>
  400649:        e8 df ff ff ff     callq  40062d <call_echo>
  40064e:        48 83 c4 08        add    $0x8,%rsp
  400652:        c3                 retq
```

Buffer Overflow Stack

Stack before call to gets

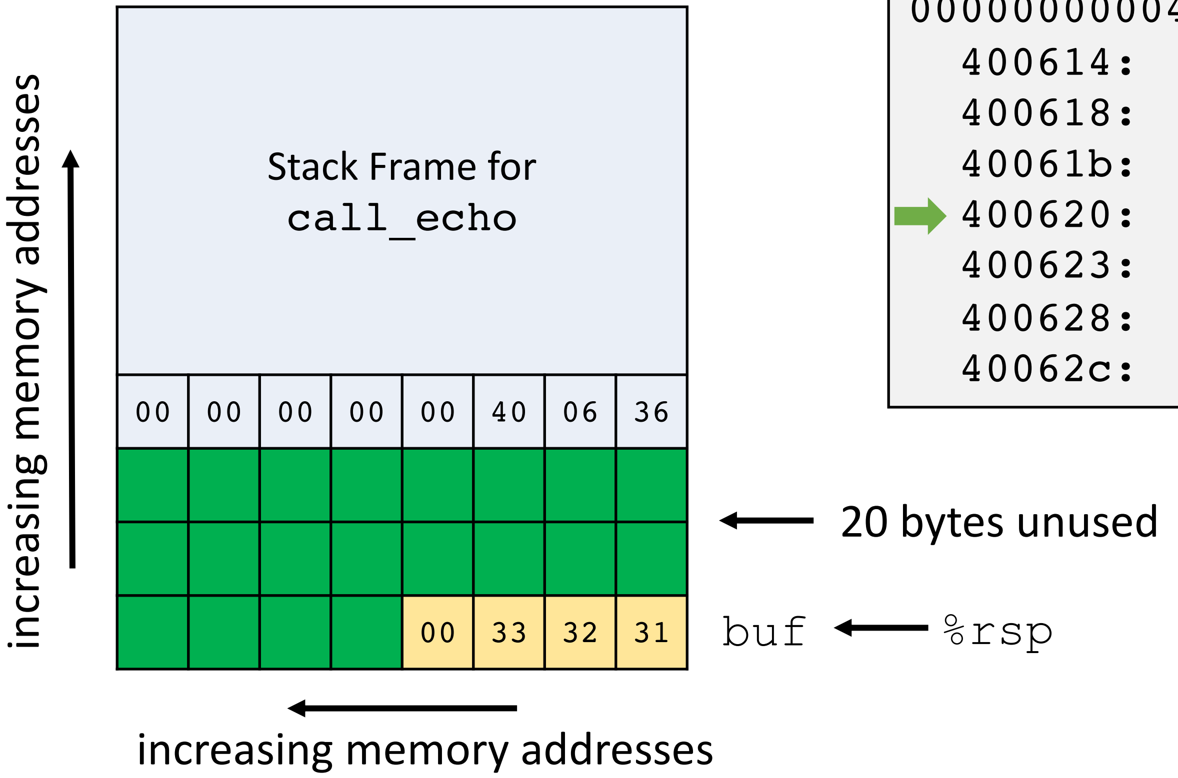



```
0000000000400614 <echo>:
  400614:       48 83 ec 18          sub    $0x18,%rsp
  400618:       48 89 e7             mov    %rsp,%rdi
  40061b:       e8 a6 ff ff ff       callq  4005c6 <gets>
  400620:       48 89 e7             mov    %rsp,%rdi
  400623:       e8 68 fe ff ff       callq  400490 <puts@plt>
  400628:       48 83 c4 18          add    $0x18,%rsp
  40062c:       c3                   retq
```

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

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

Buffer Overflow Stack

Stack after call to gets




```
0000000000400614 <echo>:
  400614:       48 83 ec 18         sub    $0x18,%rsp
  400618:       48 89 e7            mov    %rsp,%rdi
  40061b:       e8 a6 ff ff ff      callq  4005c6 <gets>
➡ 400620:       48 89 e7            mov    %rsp,%rdi
  400623:       e8 68 fe ff ff      callq  400490 <puts@plt>
  400628:       48 83 c4 18         add    $0x18,%rsp
  40062c:       c3                  retq
```

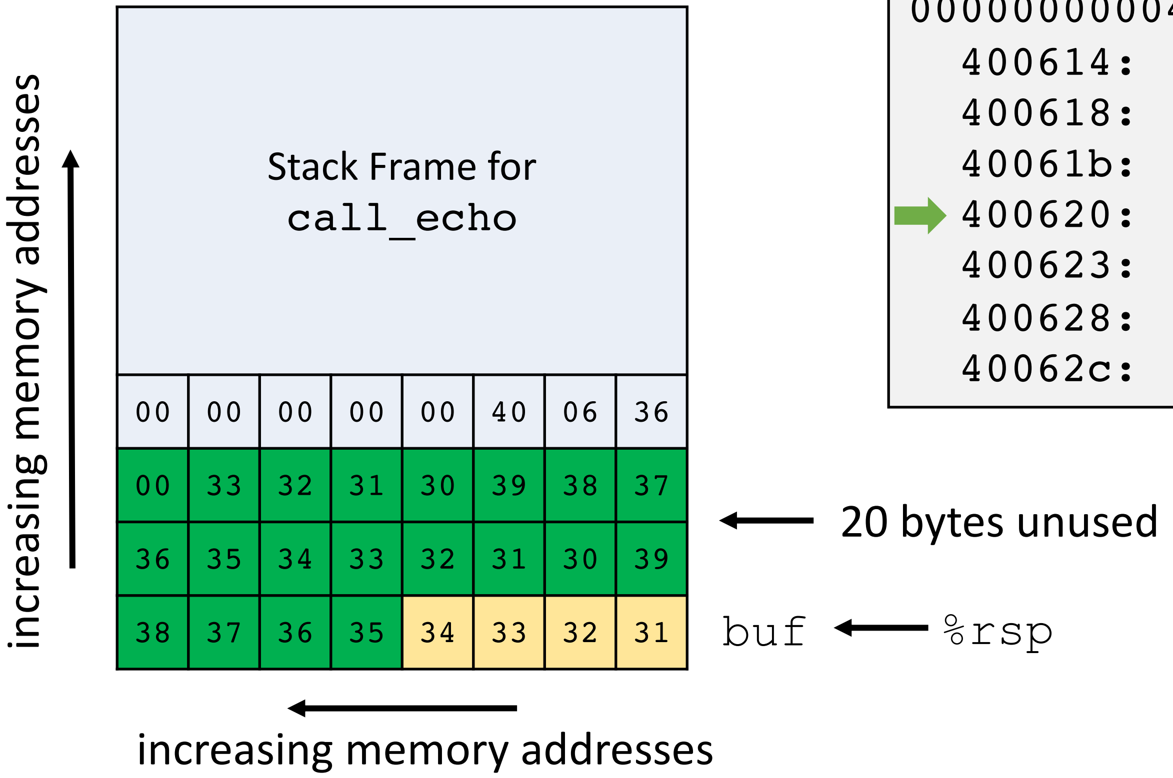
```
void call_echo(void) {
  echo();
}

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

```
Linux>./bufdemo
Type a string:
123
123
```

Buffer Overflow Stack

Overflowed buffer but didn’t corrupt return address

Stack after call to gets




```
0000000000400614 <echo>:
  400614:       48 83 ec 18        sub    $0x18,%rsp
  400618:       48 89 e7           mov    %rsp,%rdi
  40061b:       e8 a6 ff ff ff     callq  4005c6 <gets>
  400620:       48 89 e7           mov    %rsp,%rdi
  400623:       e8 68 fe ff ff     callq  400490 <puts@plt>
  400628:       48 83 c4 18        add    $0x18,%rsp
  40062c:       c3                 retq
```

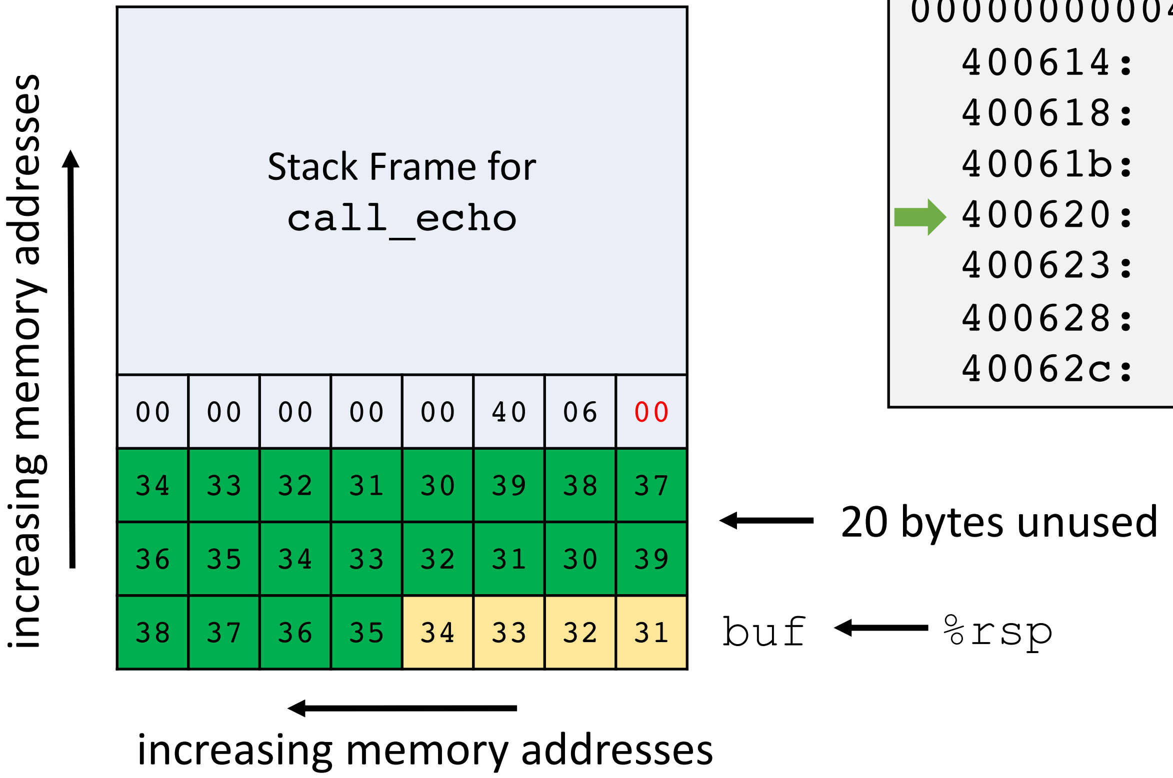
```
void call_echo(void) {
  echo();
}

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

```
Linux>./bufdemo
Type a string:
12345678901234567890123
12345678901234567890123
```

Buffer Overflow Stack

Overflowed buffer and corrupted the return address

Stack after call to gets




```
0000000000400614 <echo>:
  400614:       48 83 ec 18        sub    $0x18,%rsp
  400618:       48 89 e7           mov    %rsp,%rdi
  40061b:       e8 a6 ff ff ff     callq  4005c6 <gets>
  400620:       48 89 e7           mov    %rsp,%rdi
  400623:       e8 68 fe ff ff     callq  400490 <puts@plt>
  400628:       48 83 c4 18        add    $0x18,%rsp
  40062c:       c3                 retq
```

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

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

```
Linux>./bufdemo
Type a string:
12345678901234567890123 4
Segmentation fault
```

Where did we go?

Stack after call to gets

Stack Frame for
`call_echo`

00	00	00	00	00	40	06	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

Actual assembly code near address 0x400600

```
00000000004005c6 <gets>:
  …
  4005ff:      0f 95 c1       setne  %cl
  400602:      40 84 ce       test   %cl,%sil
  400605:      75 d9          jne    4005e0 <gets+0x1a>
  400607:      c6 02 00       movb   $0x0,(%rdx)
  40060a:      48 89 e8       mov    %rbp,%rax
  40060d:      48 83 c4 08    add    $0x8,%rsp
  400611:      5b             pop    %rbx
  400612:      5d             pop    %rbp
  400613:      c3             retq
```

Address 0x400600 interpreted as instructions

program crashes here →

```
400600:      95             xchg   %eax,%ebp
400601:      c1 40 84 ce    roll   $0xce,-0x7c(%rax)
400605:      75 d9          jne    4005e0 <gets+0x1a>
400607:      c6 02 00       movb   $0x0,(%rdx)
40060a:      48 89 e8       mov    %rbp,%rax
...
400613:      c3              retq
```

We now have the ability to hijack the program by jumping to arbitrary code anywhere in the program!

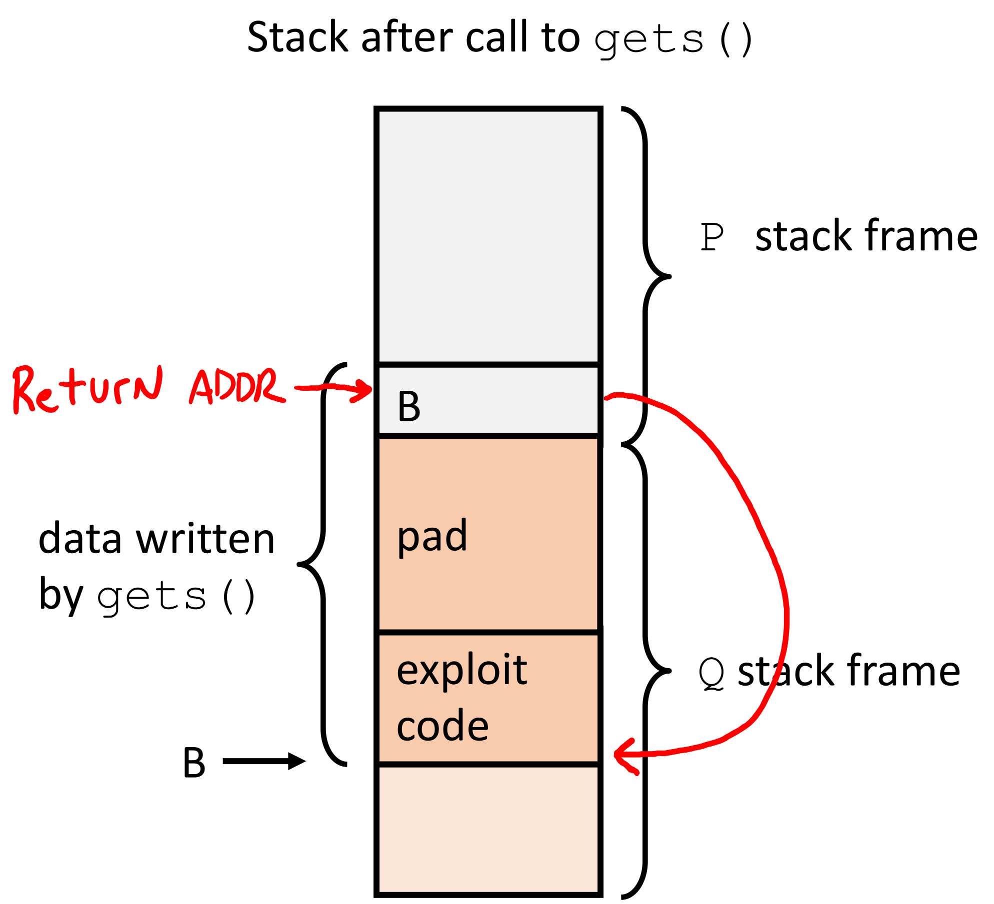
Code Injection Attacks

- 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

```
void P(){
   Q();
   ...
}
```

return address A

```
int Q() {
   char buf[64];
   gets(buf);
   ...
   return ...;
}
```



Exploits Based on Buffer Overflows

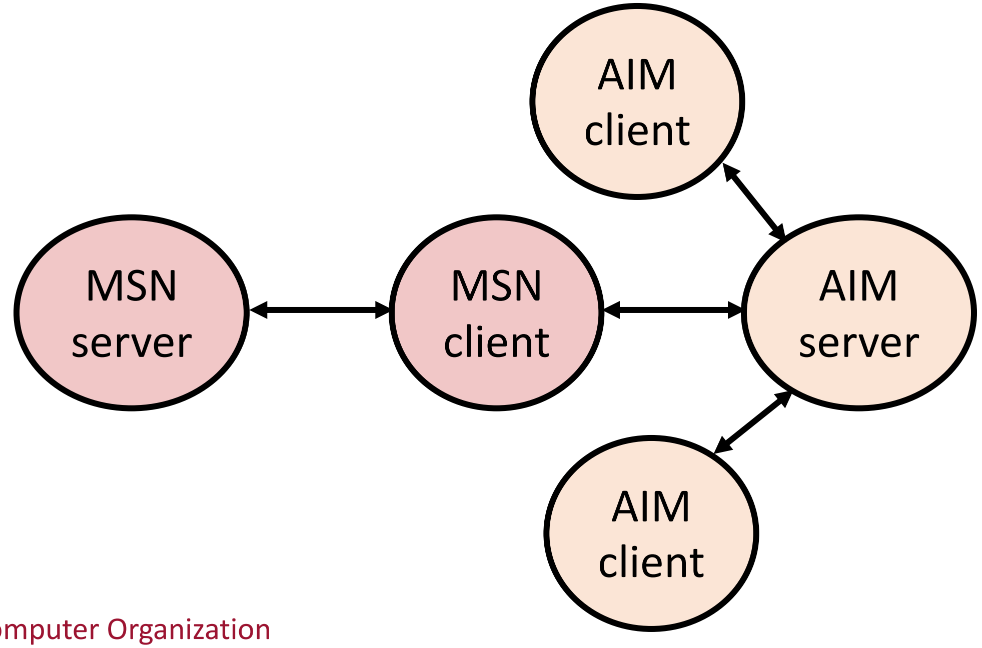
- *Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines*
- Distressingly common in real progams
  - Programmers keep making the same mistakes ☹
  - Recent measures make these attacks much more difficult
- Examples across the decades
  - Original "Internet worm" (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 server program used to see who was logged in to the system used **`gets()`** to read the argument sent by the client:
  - Worm attacked the server program by sending phony argument to the server:
    - ***"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…

Example 2: IM War

- Full account:
  - https://nplusonemag.com/issue-19/essays/chat-wars/
- 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!

OK, what to do about buffer overflow attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
- Let's talk about each of these…

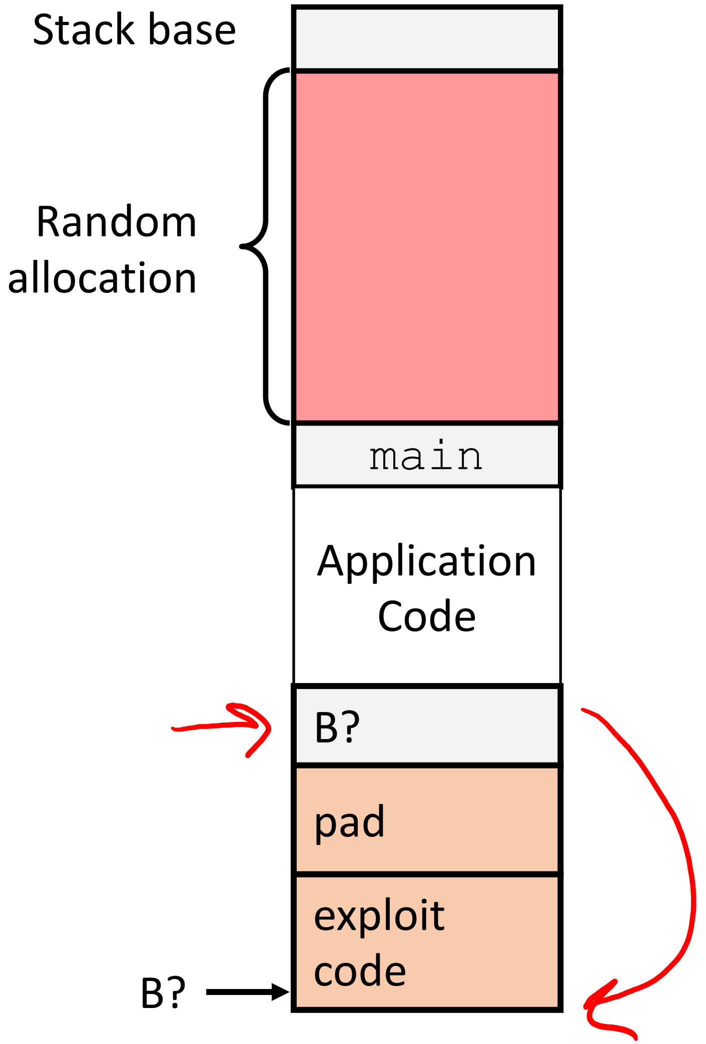
1. Avoid Overflow Vulnerabilities in Code (!)

- 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

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

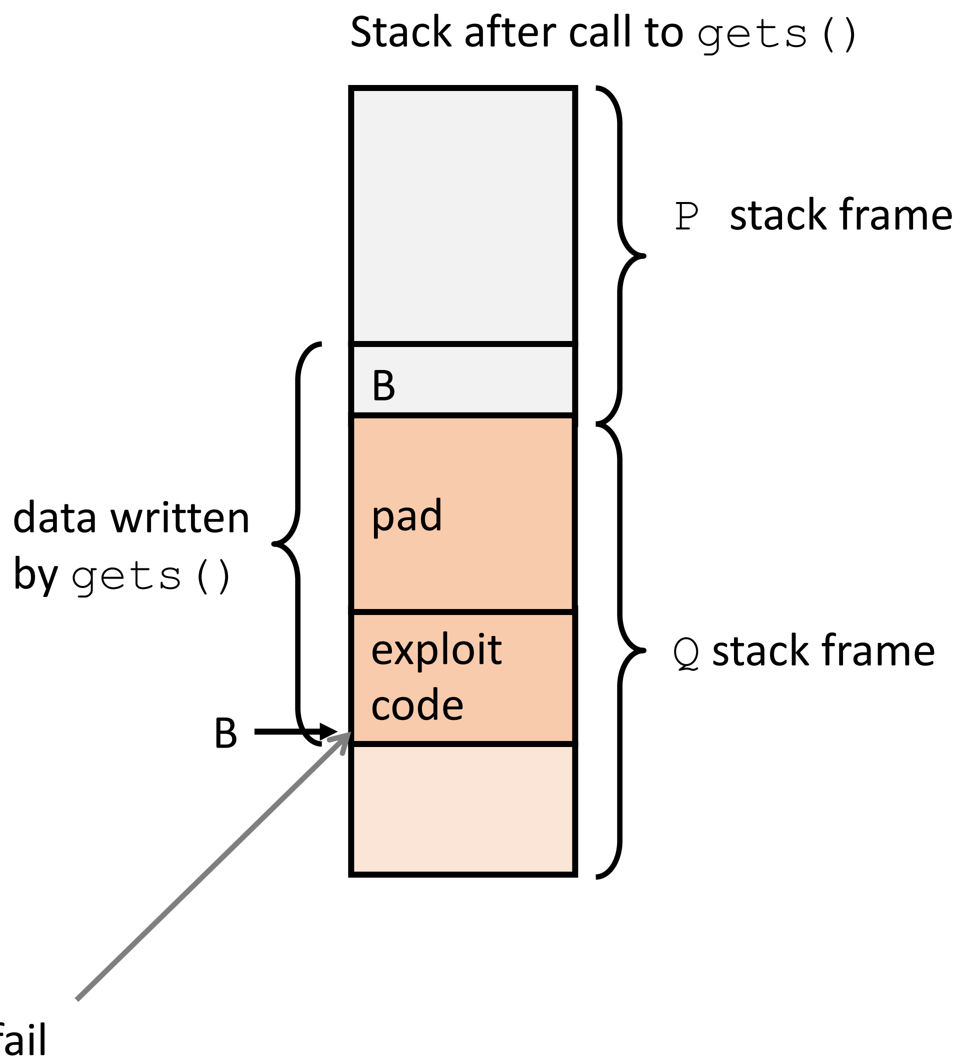
2. System-Level Protections can help

- 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



2. System-Level Protections can help

- Non-executable code segments
  - In traditional x86, can mark region of memory as either "read-only" or "writeable"
    - Can execute anything readable
  - X86-64 added explicit "execute" permission
  - Stack marked as non-executable



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)

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

```
Linux>./bufdemo-sp
Type a string:
01234567
*** buffer overflow detected ***: ./bufdemo-sp terminated
```

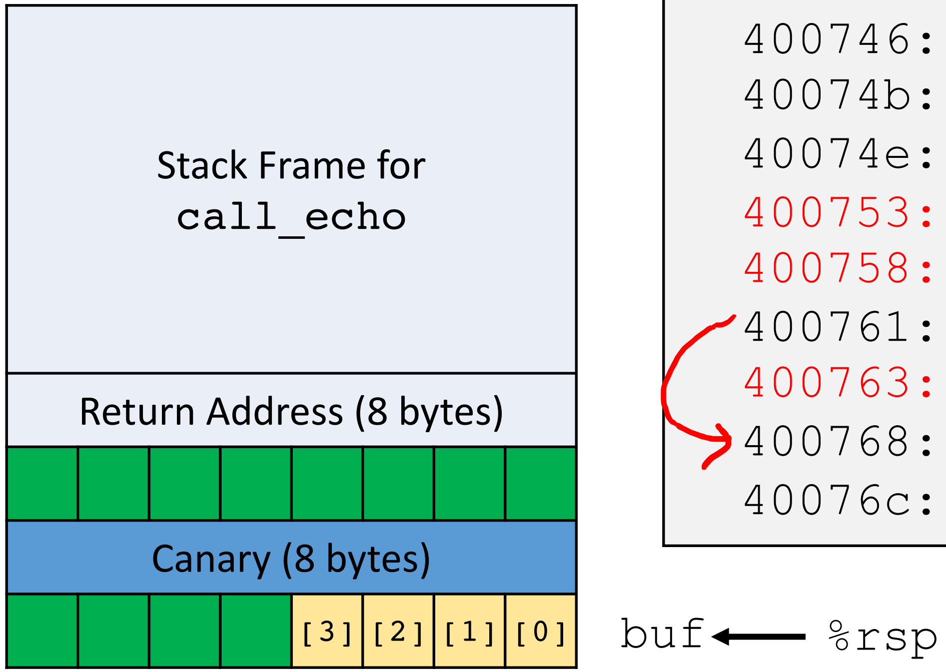
Protected Buffer Disassembly

echo:

```
40072f:  sub    $0x18,%rsp
400733:  mov    %fs:0x28,%rax   # Retrieve canary
40073c:  mov    %rax,0x8(%rsp)  # Store on stack
400741:  xor    %eax,%eax       # Zero out Register
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax  # Retrieve canary
400758:  xor    %fs:0x28,%rax   # Compare to stored value
400761:  je     400768 <echo+0x39>
400763:  callq  400580 <__stack_chk_fail@plt> # Stack bad!
400768:  add    $0x18,%rsp
40076c:  retq
```

Zero when equal

Stack before call to gets



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
    - 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
```

rax ← rdi + rdx

Gadget address = `0x4004d4`

- Use tail end of existing functions

Gadget Example #2

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

Encodes `movq %rax, %rdi`

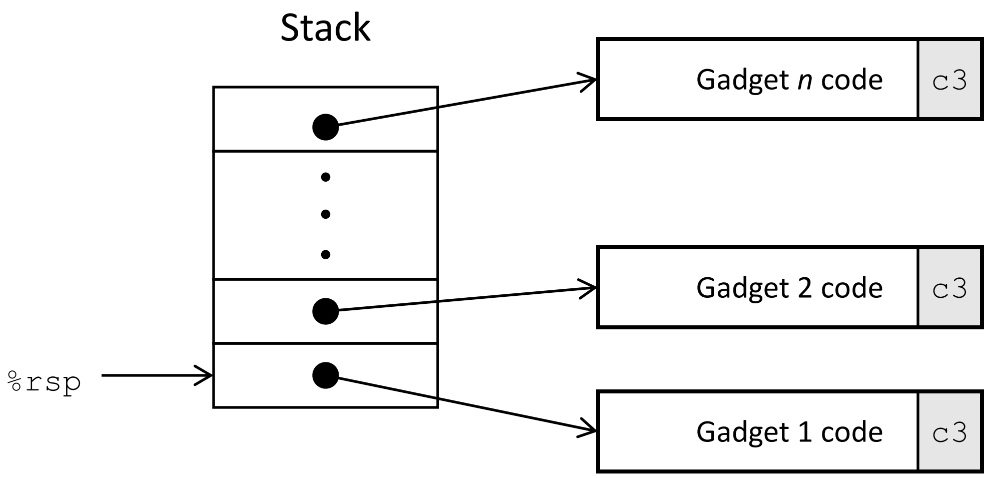
```
<setval>:
  4004d9:  c7 07 d4 48 89 c7   movl  $0xc78948d4,(%rdi)
  4004df:  c3                  retq
```

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