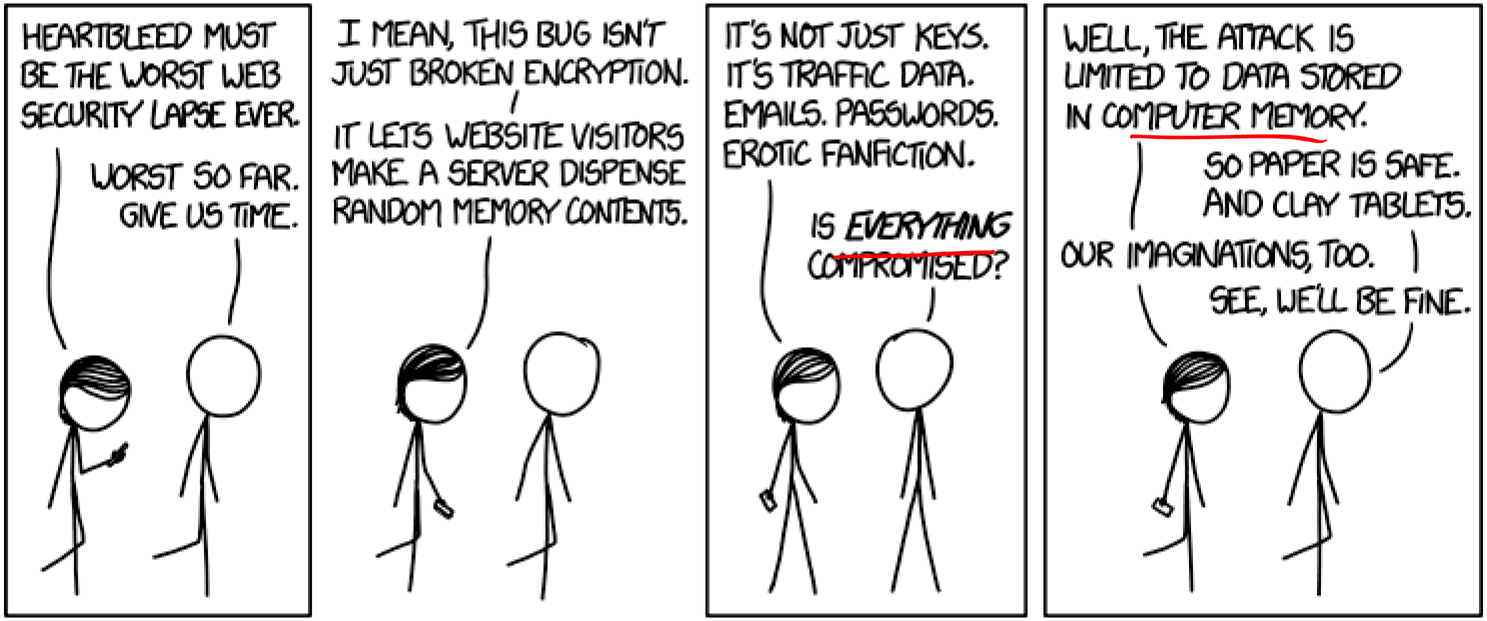
Buffer Overflows

CSE 351 Autumn 2016



Alt text: I looked at some of the data dumps from vulnerable sites, and it was ... bad. I saw emails, passwords, password hints. SSL keys and session cookies. Important servers brimming with visitor IPs. Attack ships on fire off the shoulder of Orion, c-beams glittering in the dark near the Tannhäuser Gate. I should probably patch OpenSSL.

http://xkcd.com/1353/

Instructor:

Justin Hsia

Teaching Assistants:

Chris Ma

Hunter Zahn

John Kaltenbach

Kevin Bi

Sachin Mehta

Suraj Bhat

Thomas Neuman

Waylon Huang

Xi Liu

Yufang Sun

Administrivia

- Lab 2 due, Homework 2 released today
- **Midterm** on Nov. 2 in lecture
  - Make a cheat sheet! – two-sided letter page, *handwritten*
  - Midterm details Piazza post: @225
    - Past Num Rep and Floating Point questions *and solutions* posted
- **Midterm review session**
  - 5-7pm on Monday, Oct. 31 in EEB 105
- Extra office hours
  - Sachin Fri 10/28, 5-8pm, CSE 218
  - Justin Tue 11/1, 12:30-4:30pm, CSE 438

Buffer overflows

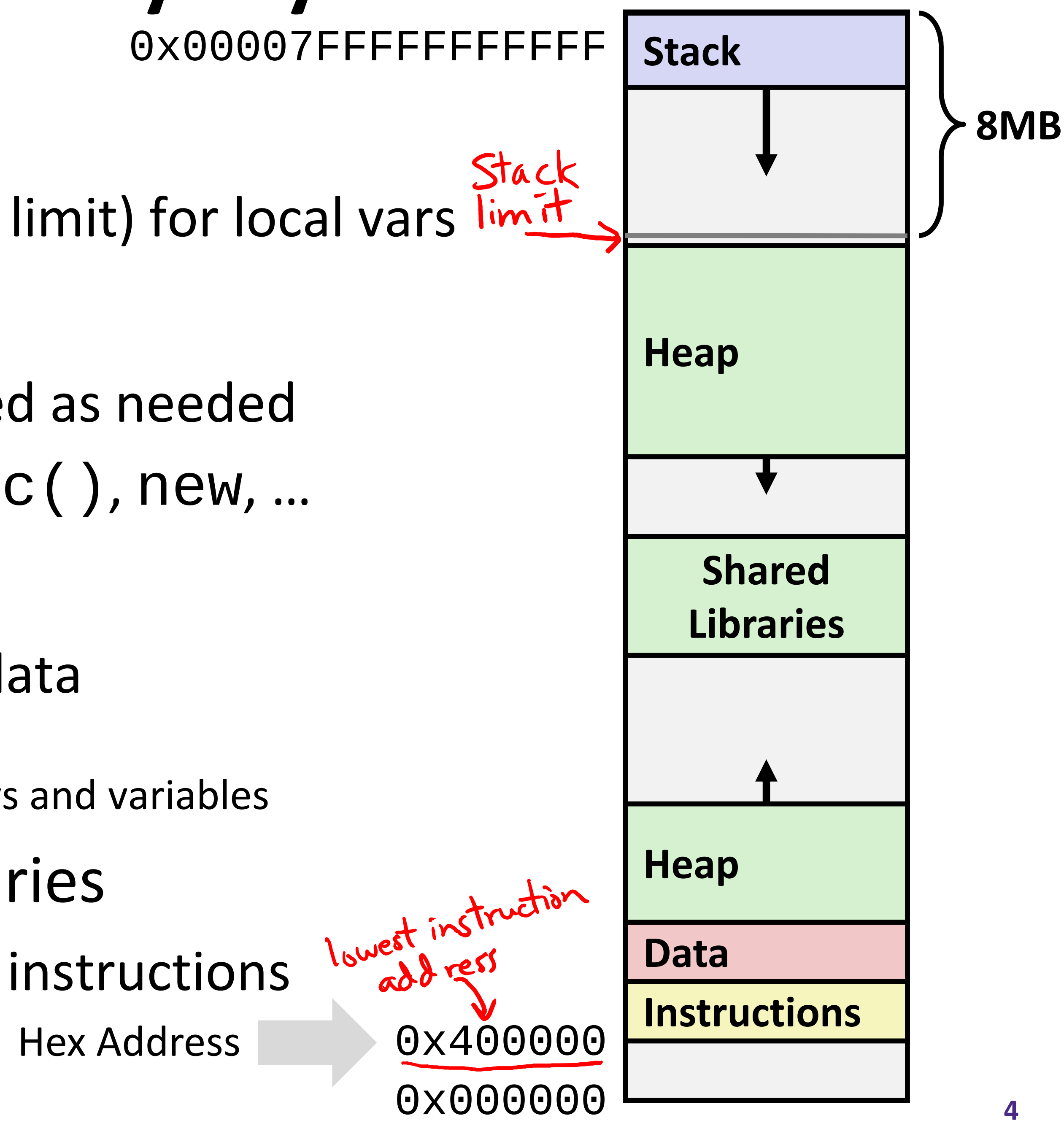
- Buffer overflows are possible because C does not check array boundaries
- Buffer overflows are dangerous because buffers for user input are often stored on the stack

- Specific topics:
  - Address space layout (more details!)
  - Input buffers on the stack
  - Overflowing buffers and injecting code
  - Defenses against buffer overflows

not drawn to scale

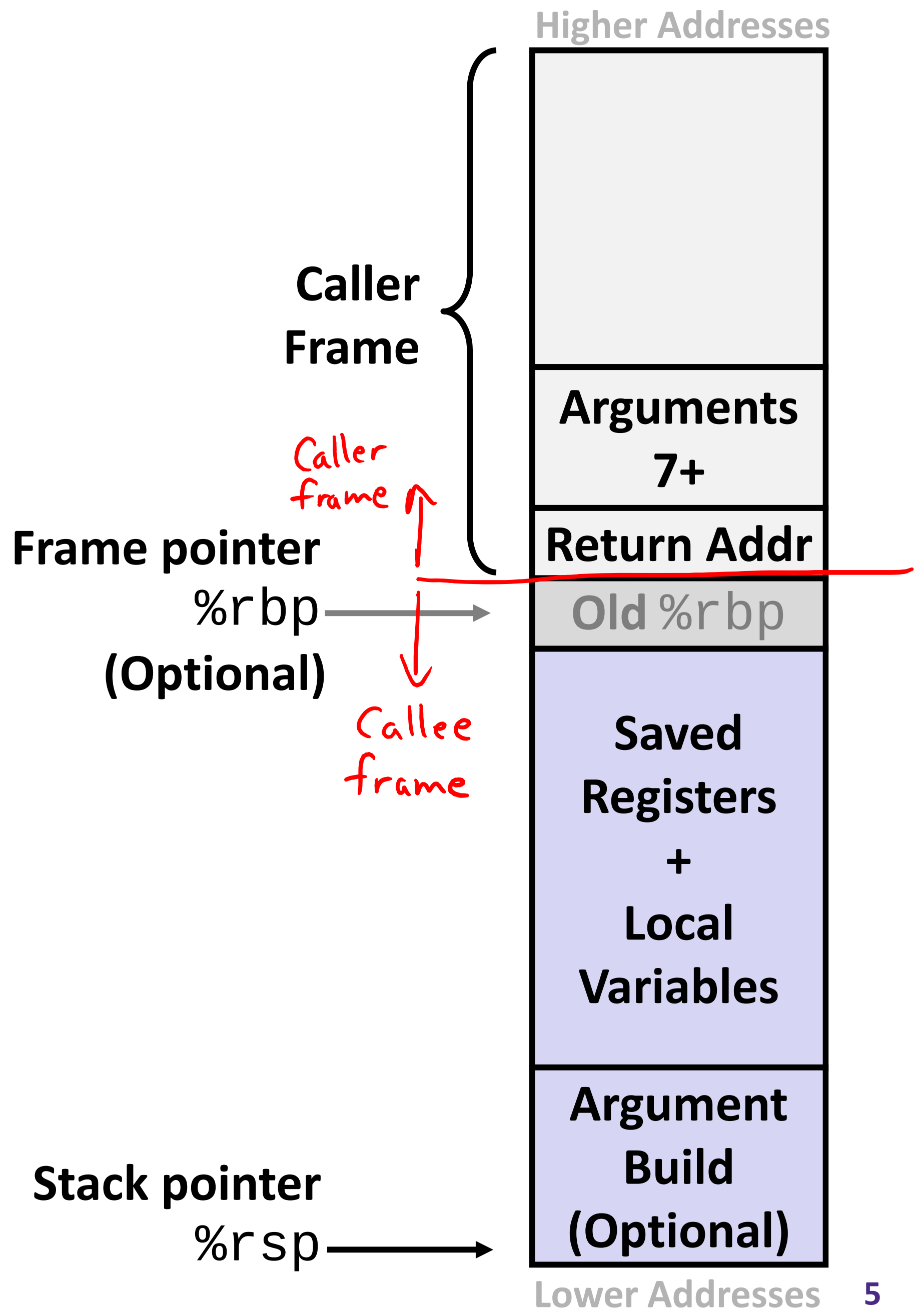
x86-64 Linux Memory Layout

- Stack
  - Runtime stack (8MB limit) for local vars
- Heap
  - Dynamically allocated as needed
  - `malloc()`, `calloc()`, `new`, …
- Data
  - Statically allocated data
    - Read-only: string literals
    - Read/write: global arrays and variables
- Code / Shared Libraries
  - Executable machine instructions
  - Read-only



Reminder: x86-64/Linux Stack Frame

- Caller's Stack Frame
  - Arguments (if > 6 args) for this call
  - Return address
    - Pushed by `call` instruction
- Current/ Callee Stack Frame
  - Old frame pointer (optional)
  - Saved register context (when reusing registers)
  - Local variables (if can't be kept in registers)
  - "Argument build" area (If callee needs to call another function -parameters for function about to call, if needed)



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 ... */
}
```

global vars

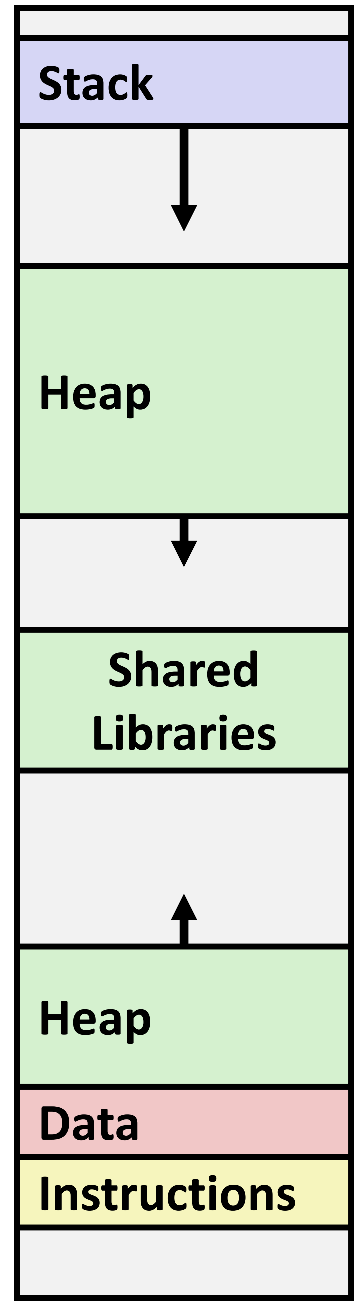
functions

local vars

dynamically allocated memory

Where does everything go?

not drawn to scale



not drawn to scale

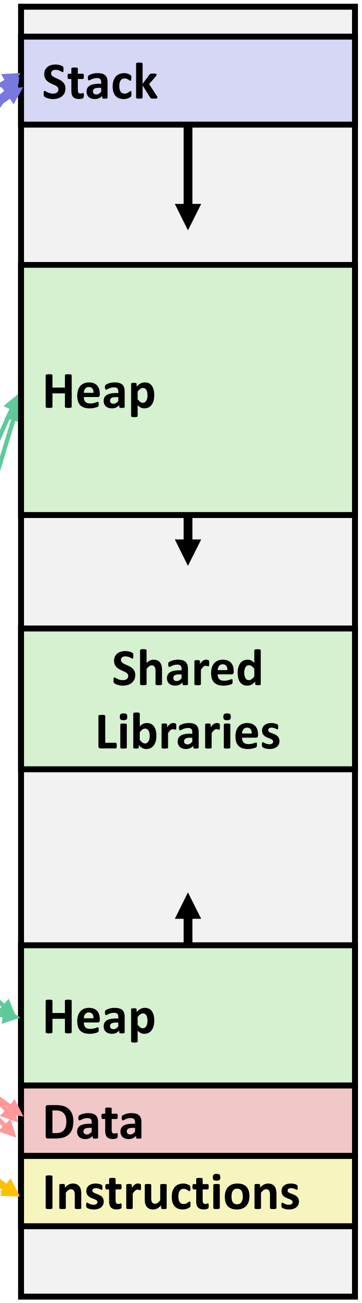
Memory Allocation Example

```
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?

Buffer overflows

- Buffer overflows are possible because C does not check array boundaries
- Buffer overflows are dangerous because buffers for user input are often stored on the stack

- Specific topics:
  - Address space layout (more details!)
  - Input buffers on the stack
  - Overflowing buffers and injecting code
  - Defenses against buffer overflows

Internet Worm

- These characteristics of the traditional Linux memory layout provide opportunities for malicious programs
  - Stack grows “backwards” in memory
  - Data and instructions both stored in the same memory
- November, 1988
  - Internet Worm attacks thousands of Internet hosts.
  - How did it happen?
- *Stack buffer overflow* exploits!

Buffer Overflow in a nutshell

- Many Unix/Linux/C functions don't check argument sizes
- C does not check array bounds
  - Allows overflowing (writing past the end of) buffers (arrays)
- Overflows of buffers on the stack overwrite "interesting" data
  - Attackers just choose the right inputs
- Why a big deal?
  - It is (was?) the #1 *technical* cause of security vulnerabilities
    - #1 *overall* cause is social engineering / user ignorance
- Simplest 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;
}
```

end of file

newline

reads character from input stream

pointer to start of an array

same as:

```
*p = c;
p++;
```

- What could go wrong in this code?

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

    ↖ stop condition looking for special characters

- Similar problems with other Unix functions:
  - `strcpy`: Copies string of arbitrary length to a dst
  - `scanf`, `fscanf`, `sscanf`, when given %s specifier

Vulnerable Buffer Code

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

input buffer (points to buf[4])

read input into buffer (points to gets(buf))

print output from buffer (points to puts(buf))

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

```
unix> ./buf-nsp
Enter string: 012345678901234567890123
012345678901234567890123
```

```
unix> ./buf-nsp
Enter string: 0123456789012345678901234
Segmentation Fault
```

Buffer Overflow Disassembly

echo:

```
00000000004006cf <echo>:
 4006cf:  48 83 ec 18          sub    $24,%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    $24,%rsp
 4006e7:  c3                   ret
```

compiler choice

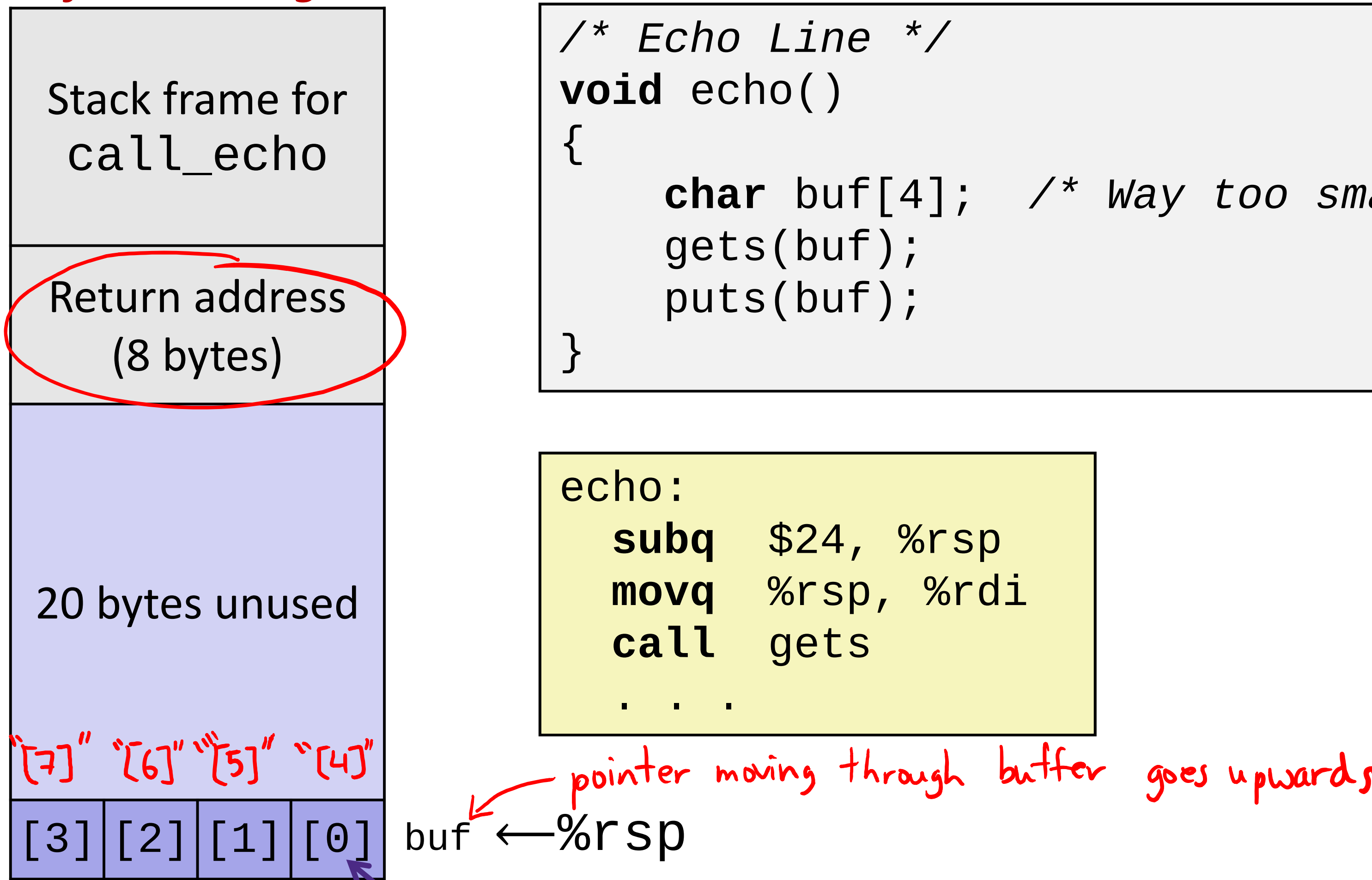
call_echo:

```
 4006e8:  48 83 ec 08          sub    $8,%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    $8,%rsp
 4006fa:  c3                   ret
```

return address placed on stack

Buffer Overflow Stack

Before call to gets



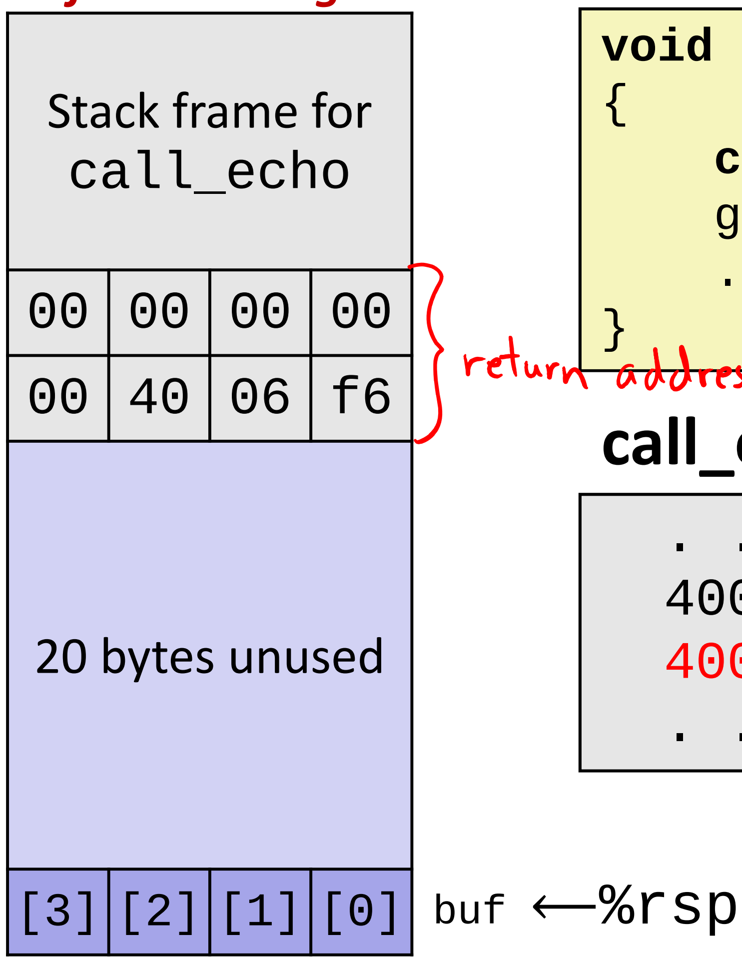

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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

Note: addresses increasing right-to-left, bottom-to-top

Buffer Overflow Example

Before call to gets



return address in 64-bits

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

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

call_echo:

```
  . . .
  4006f1:  callq   4006cf <echo>
  4006f6:  add     $8,%rsp
  . . .
```

Buffer Overflow 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

'8'

'0'

Note: Digit "*N*" is just 0x3*N* in ASCII!

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

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

call_echo:

```
  . . .
  4006f1:   callq   4006cf <echo>
  4006f6:   add     $8,%rsp
  . . .
```

```
unix> ./buf-nsp
Enter string: 01234567890123456789012
01234567890123456789012
```

Overflowed buffer, but did not corrupt state

Buffer Overflow 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    $8,%rsp
. . .
```

```
unix> ./buf-nsp
Enter string: 0123456789012345678901234
Segmentation Fault
```

Overflowed buffer and corrupted return pointer

Buffer Overflow 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     $8,%rsp
  . . .
```

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

Overflowed buffer, corrupted return pointer, but program seems to work! – valid instruction address

Buffer Overflow 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
```

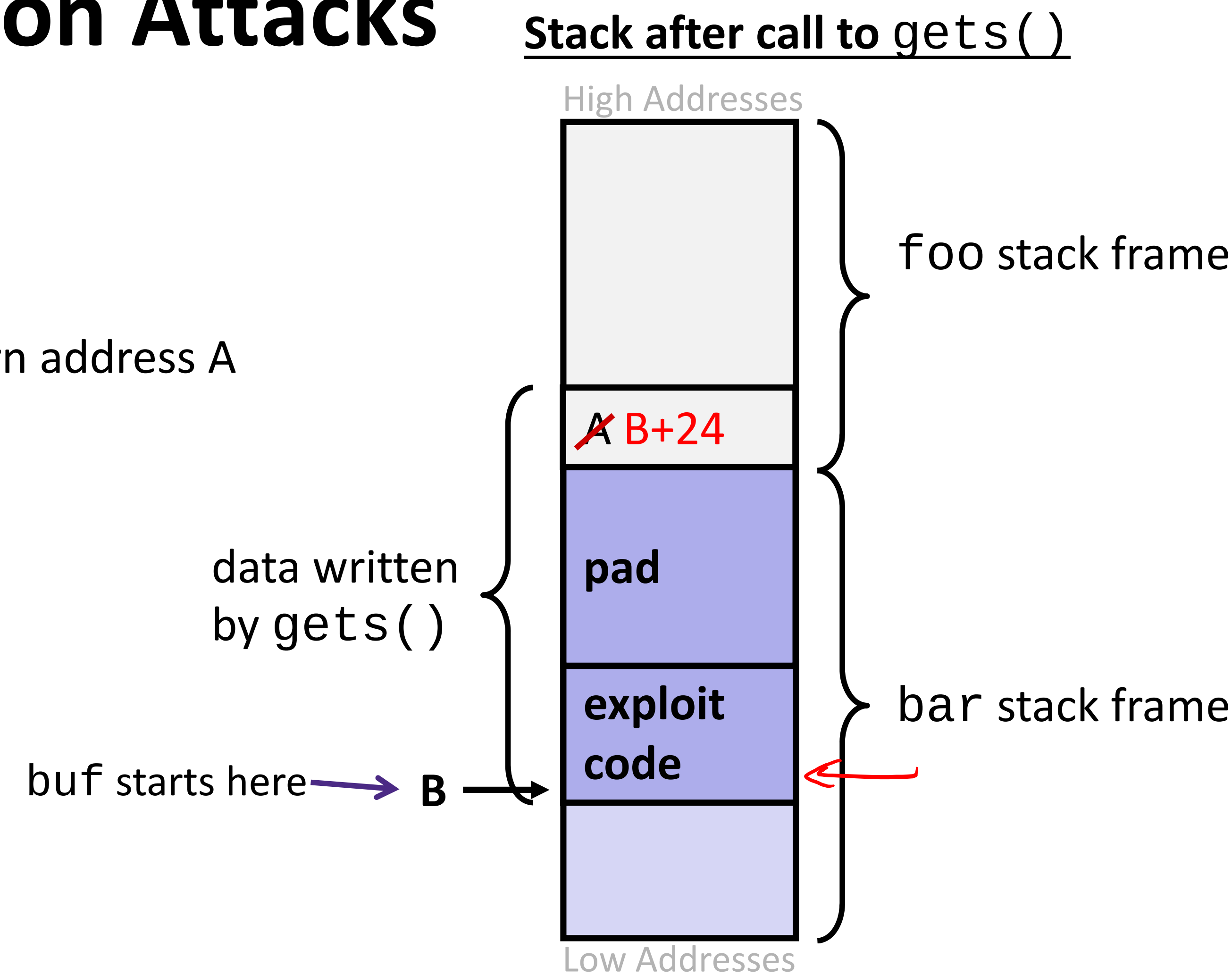
"Returns" to unrelated code.
Lots of things happen, but *without* modifying critical state.
Eventually executes `retq` back to `main`.

Malicious Use of Buffer Overflow: Code Injection Attacks

```
void foo(){
  bar();
A:...
}
```

return address A

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



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

Exploits Based on Buffer Overflows

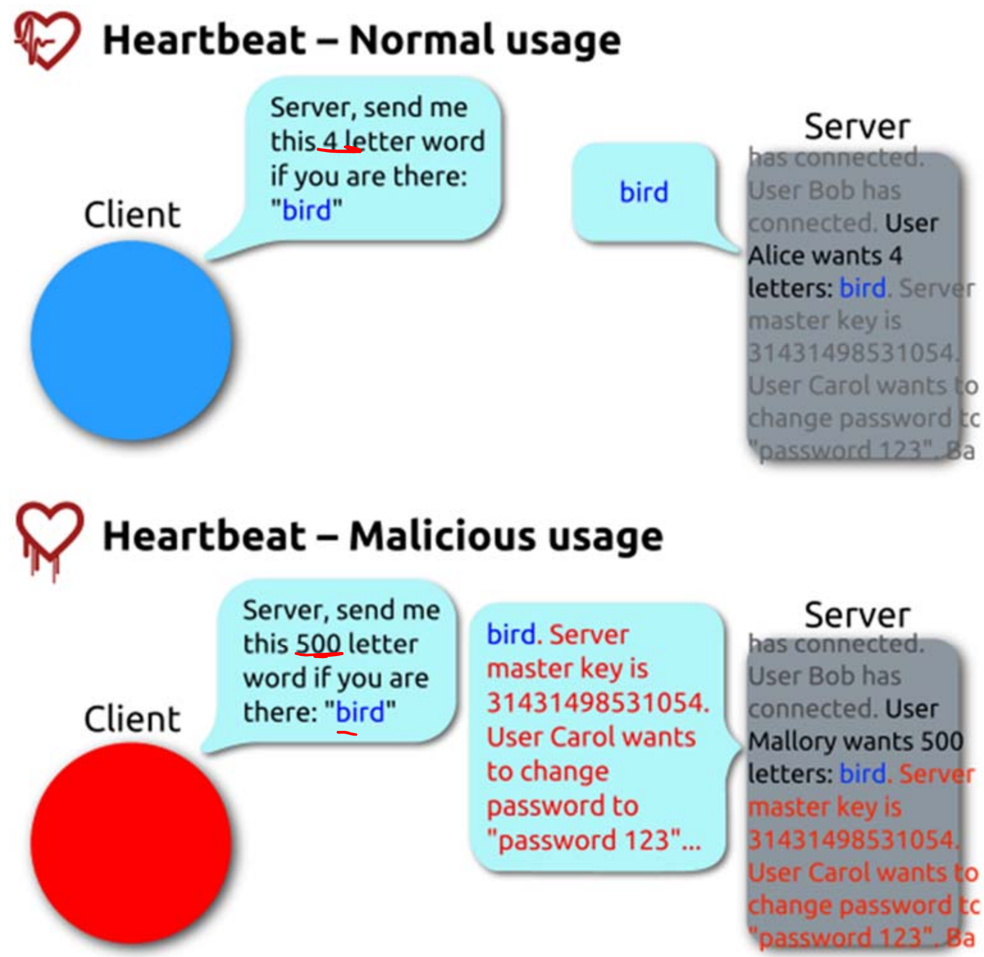
- *Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines*
- Distressingly common in real programs
  - Programmers keep making the same mistakes ☹
  - Recent measures make these attacks much more difficult
- Examples across the decades
  - Original "Internet worm" (1988)
  - *Still happens!!* Heartbleed (2014, affected 17% of servers)
  - *Fun:* Nintendo hacks
    - Using glitches to rewrite code: https://www.youtube.com/watch?v=TqK-2jUQBUY
    - FlappyBird in Mario: https://www.youtube.com/watch?v=hB6eY73sLV0
- You will learn some of the tricks in Lab 3
  - 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@cs.cmu.edu`
  - Worm attacked fingerd server by sending phony argument:
    - `finger` "*`exploit-code padding new-return-addr`*"
    - 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…

Heartbleed (2014!)

- Buffer over-read in OpenSSL
  - Open source security library
  - Bug in a small range of versions
- "Heartbeat" packet
  - Specifies length of message
  - Server echoes it back
  - Library just "trusted" this length
  - Allowed attackers to read contents of memory anywhere they wanted
- Est. 17% of Internet affected
  - "Catastrophic"
  - Github, Yahoo, Stack Overflow, Amazon AWS, ...



Dealing with buffer overflow attacks

1) Avoid overflow vulnerabilities
2) Employ system-level protections
3) Have compiler use “stack canaries”

1) Avoid Overflow Vulnerabilities in Code

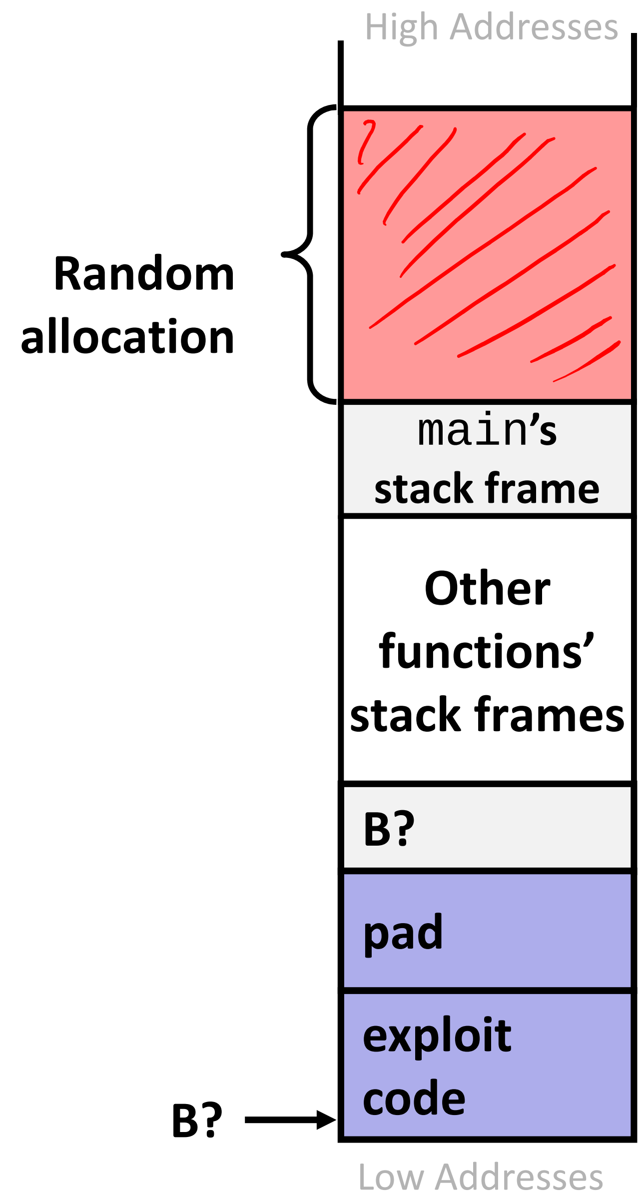
```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

character read limit

- Use library routines that limit string lengths
  - `fgets` instead of `gets` (2nd argument to `fgets` sets limit)
  - `strncpy` instead of `strcpy`
  - Don't use `scanf` with %s conversion specification
    - Use `fgets` to read the string
    - Or use %`n`s where `n` is a suitable integer

2) System-Level Protections

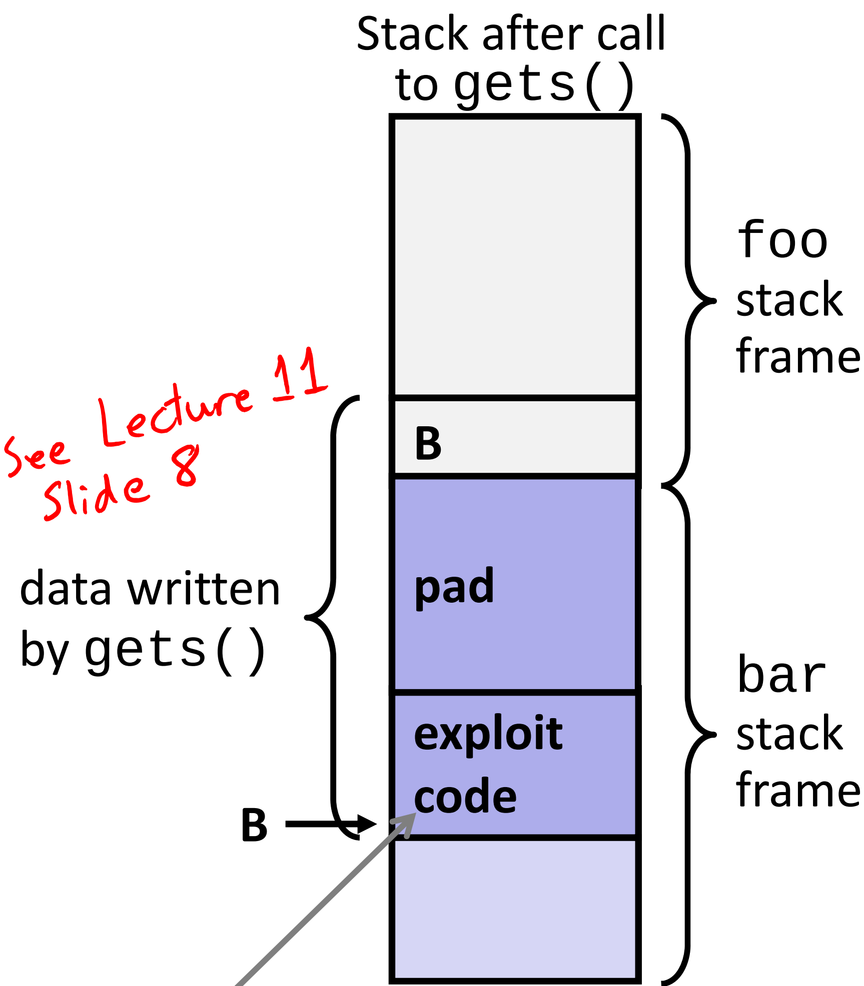
- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
    - Addresses will vary from one run to another
  - Makes it difficult for hacker to predict beginning of inserted code
- Example: Code from Slide 6 executed 5 times; address of variable `local` =
    - `0x7ffe4d3be87c`
    - `0x7fff75a4f9fc`
    - `0x7ffeadb7c80c`
    - `0x7ffeaea2fdac`
    - `0x7ffcd452017c`
  - Stack repositioned each time program executes



2) System-Level Protections

- **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
    - Do *NOT* execute code in Stack, Static Data, or Heap regions
    - Hardware support needed

See Lecture 11 Slide 8



Any attempt to execute this code will fail

3) Stack Canaries

- Basic Idea: place special value ("canary") on stack just beyond buffer
  - *Secret* value known only to compiler
  - "After" buffer but before return address
  - Check for corruption before exiting function
- GCC implementation (now default)
  - `-fstack-protector`
  - Code back on Slide 13 (`buf-nsp`) compiled with `–fno-stack-protector` flag

```
unix>./buf
Enter string: 01234567
01234567
```

```
unix> ./buf
Enter string: 012345678
*** stack smashing detected ***
```

Protected Buffer Disassembly

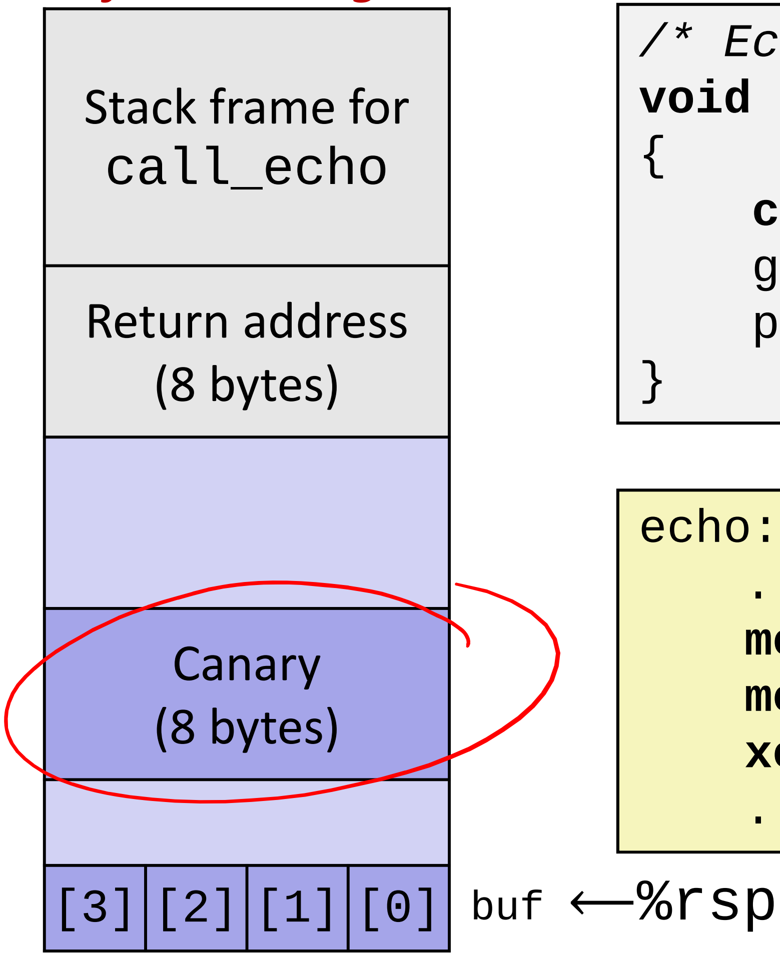
echo:

```
40072f:  sub    $0x18,%rsp
400733:  mov    %fs:0x28,%rax          # read canary value
40073c:  mov    %rax,0x8(%rsp)         # store canary on Stack
400741:  xor    %eax,%eax              # erase canary from register
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax         # read current canary on Stack
400758:  xor    %fs:0x28,%rax          # compare against original value
400761:  je     400768 <echo+0x39>     # if unchanged, then return
400763:  callq  400580 <__stack_chk_fail@plt>  # stack smashing detected
400768:  add    $0x18,%rsp
40076c:  retq
```

try unix> diff buf.s buf-nsp.s

Setting Up Canary

Before call to gets



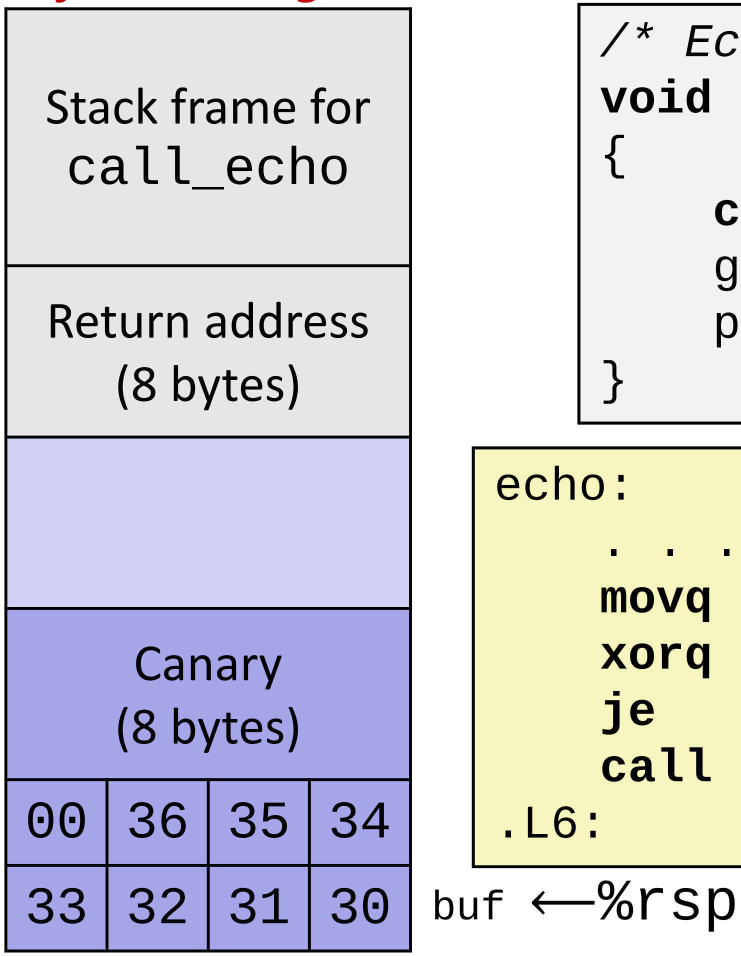

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:40, %rax   # Get canary
    movq    %rax, 8(%rsp)  # Place on stack
    xorl    %eax, %eax     # Erase canary
    . . .
```

Segment register
(don't worry about it)

Checking Canary

After call to gets




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

```
echo:
    . . .
    movq    8(%rsp), %rax      # retrieve from Stack
    xorq    %fs:40, %rax       # compare to canary
    je      .L6                # if same, OK
    call    __stack_chk_fail   # els, FAIL
.L6:        . . .
```

Input: *0123456*

Summary

1) Avoid overflow vulnerabilities
   - Use library routines that limit string lengths

2) Employ system-level protections
   - Randomized Stack offsets
   - Code on the Stack is not executable

3) Have compiler use “stack canaries”