Buffer Overflows

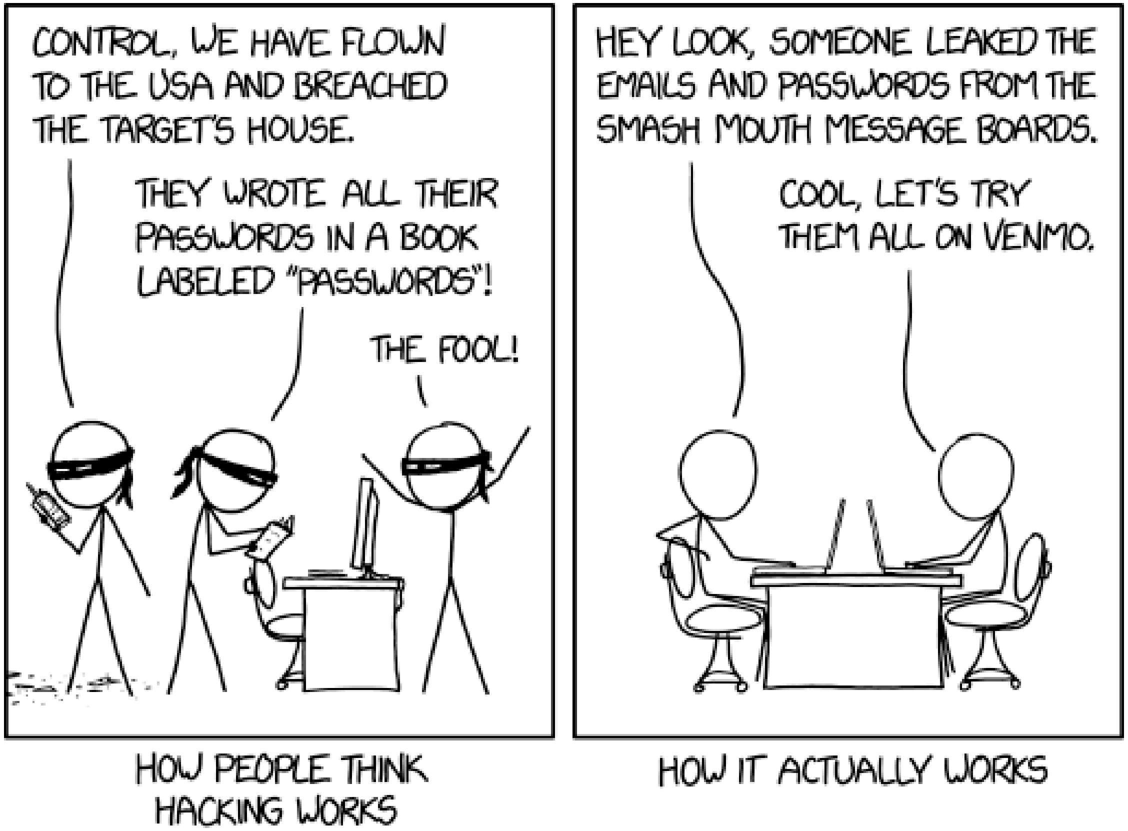
CSE 351 Autumn 2024

Instructor:

Ruth Anderson

Teaching Assistants:

Alexandra Michael
Connie Chen
Chloe Fong
Chendur Jayavelu
Joshua Tan
Nikolas McNamee
Nahush Shrivatsa
Naama Amiel
Neela Kausik
Renee Ruan
Rubee Zhao
Samantha Dreussi
Sean Siddens
Waleed Yagoub



https://xkcd.com/2176

Relevant Course Information

- HW13 due TONIGHT, Monday (10/28) @ 11:59 pm
- HW14 due Wednesday (10/30) @ 11:59 pm
- No Lecture on Fri 11/01 (No HW/Reading due)
  - HW15 not due until next Monday (11/04)
- **Midterm Exam**: https://cs.uw.edu/cse351/exams/
  - Take home, on Gradescope
  - Open: Thursday 10/31 at 5pm; Due: Saturday 11/02 at 11:59pm
  - Review in section this week (10/31)
  - See Ed Post and exams page
- Lab 3 coming soon!
  - You will have everything you need by the end of this lecture

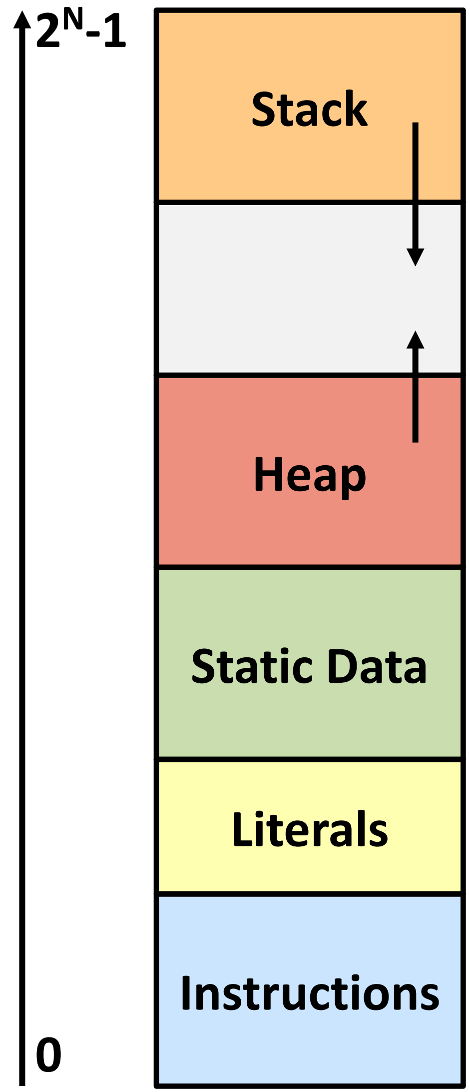
Buffer Overflows

- Address space layout review
- Input buffers on the stack
- Overflowing buffers and injecting code
- Defenses against buffer overflows

not drawn to scale

Review: General Memory Layout

- Stack
  - Local variables (procedure context)
- Heap
  - Dynamically allocated as needed
  - `new`, `malloc()`, `calloc()`, …
- Statically-allocated Data
  - Read/write: global variables (Static Data)
  - Read-only: string literals (Literals)
- Code/Instructions
  - Executable machine instructions
  - Read-only



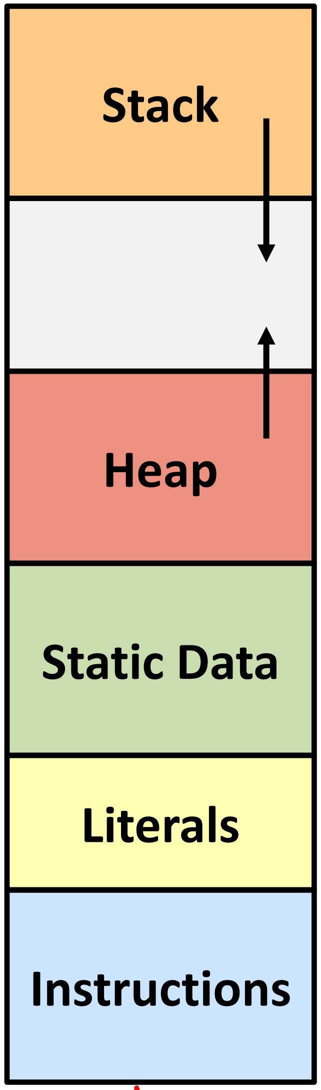
Memory Allocation Example

```
char big_array[1L<<24];  /* 16 MB */

int global = 0;

int useless() { return 0; }

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

Where does everything go?

not drawn to scale

high addr

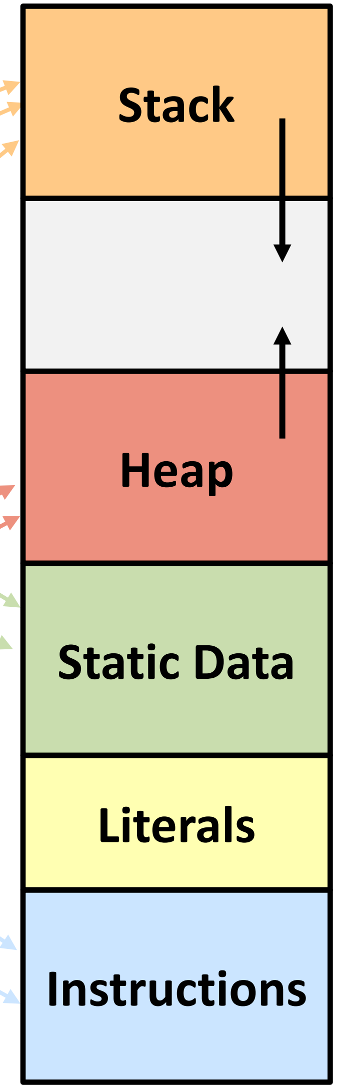


low addr

not drawn to scale

Memory Allocation Example (Answers)

```
char big_array[1L<<24];   /* 16 MB */

int global = 0;

int useless() { return 0; }

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



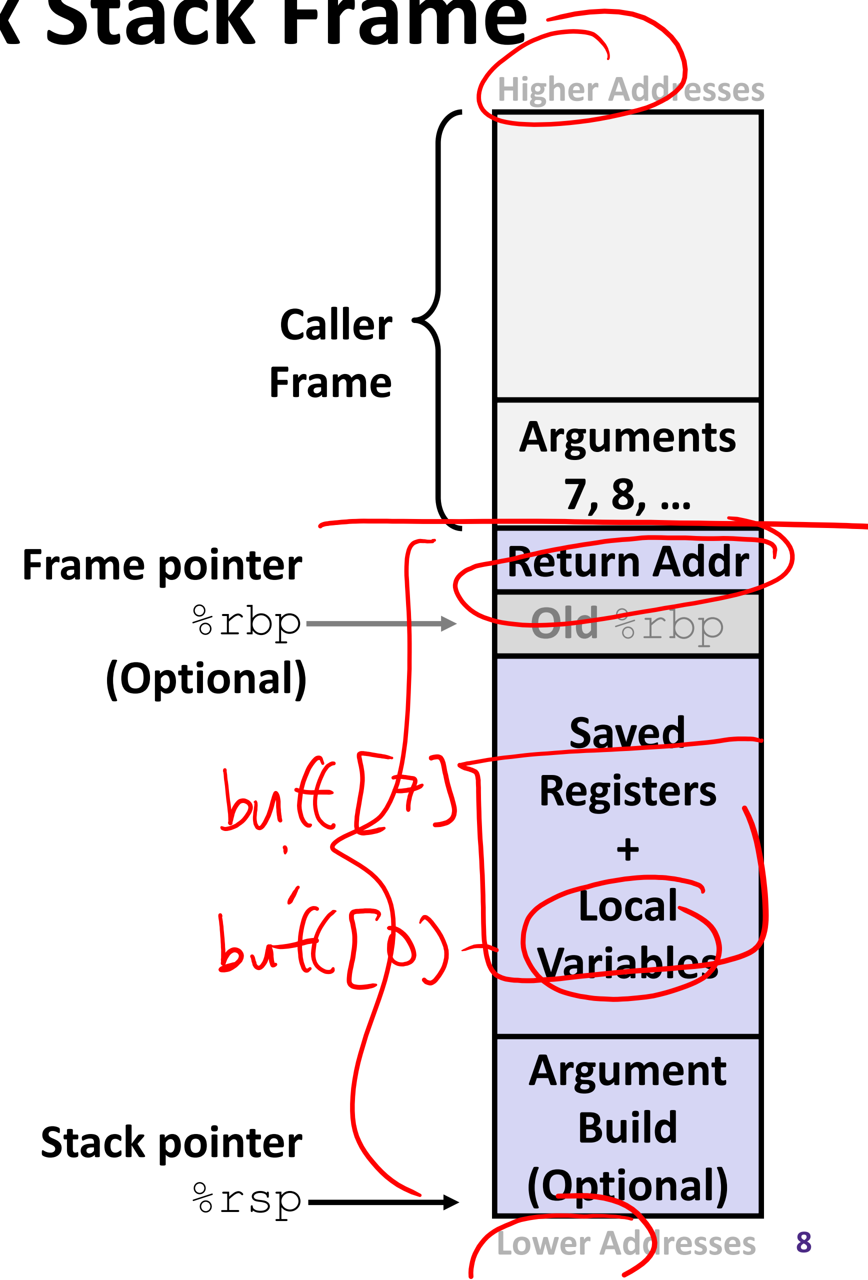
Where does everything go?

What Is a Buffer?

- A buffer is an array used to temporarily store data
- You've probably seen "video buffering…"
  - The video is being written into a buffer before being played
- Buffers can also store user input

Reminder: x86-64/Linux Stack Frame

- Caller's Stack Frame
  - Arguments (if > 6 args) for this call
- Current/ Callee Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Old frame pointer (optional)
  - Callee-saved pushed before using long-term registers
  - Local variables (if can't be kept in registers)
  - Caller-saved pushed before setting up arguments for a function call
  - "Argument build" area (Need to call a function with >6 arguments? Put them here)



Buffer Overflow in a Nutshell

- C does not check array bounds
  - Many Unix/Linux/C functions don't check argument sizes
  - Allows overflowing (writing past the end) of buffers (arrays)
- "Buffer Overflow" = Writing past the end of an array
- 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

Buffer Overflow (1)

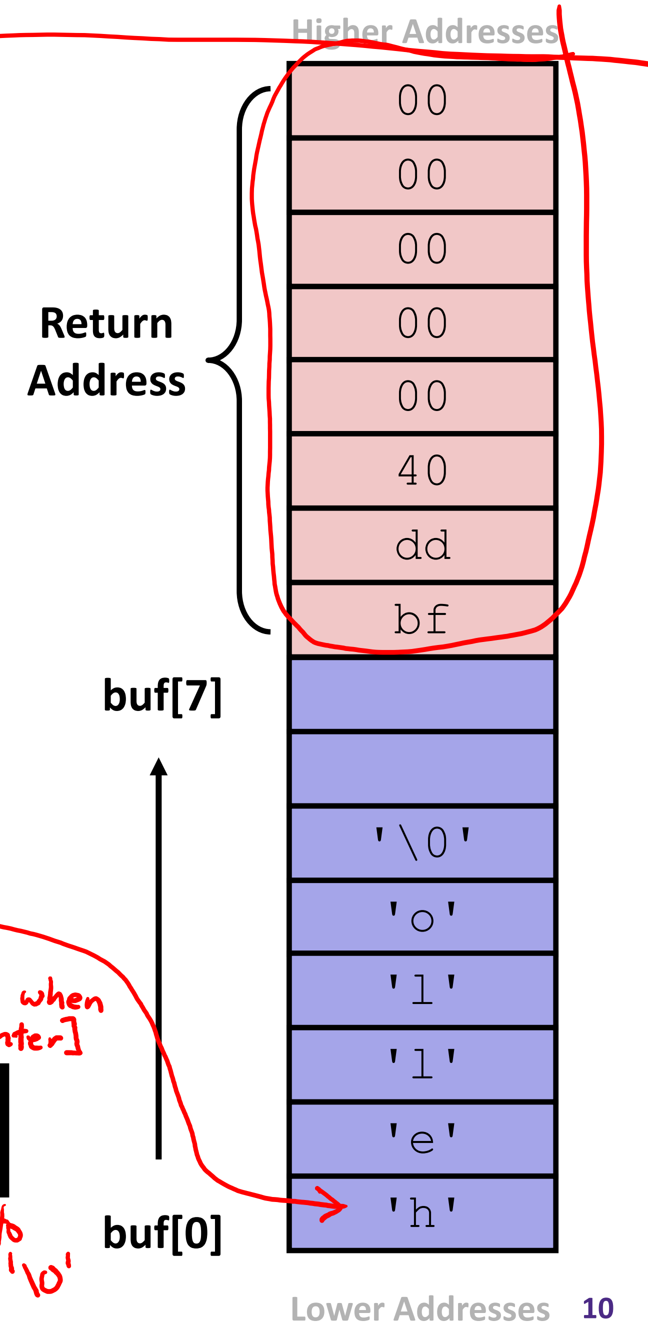
- Stack grows *down* towards lower addresses
- Buffer grows *up* towards higher addresses
- If we write past the end of the array, we overwrite data on the stack!

```
Enter input: hello \n
```

newline inserted when user presses [Enter]

later converted to a null terminator '\0'

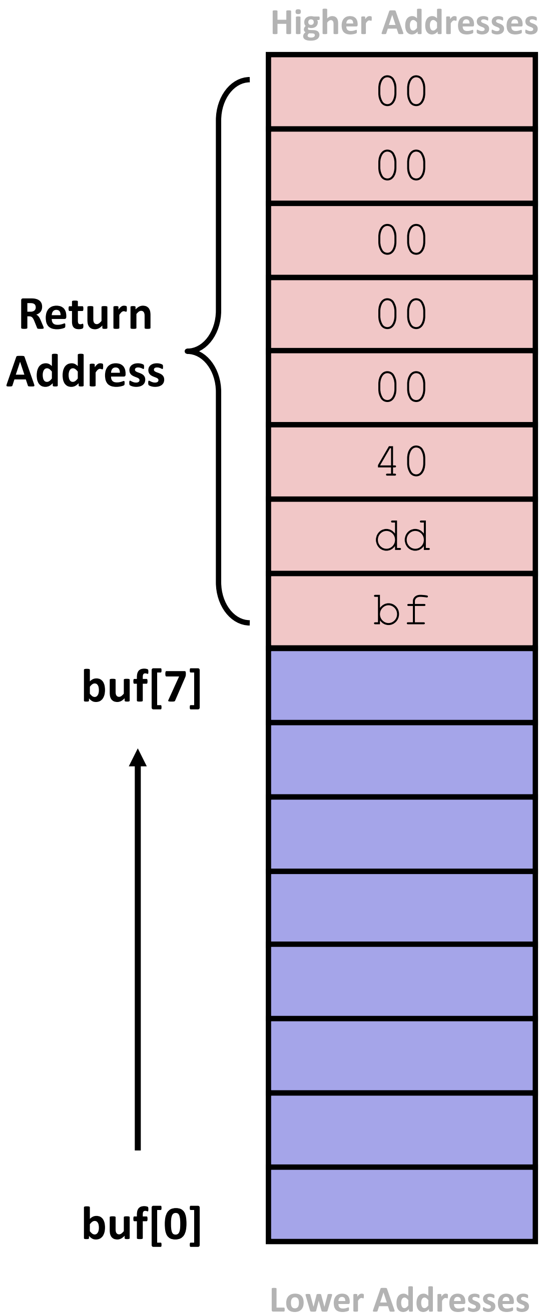
No overflow ☺



Buffer Overflow (2)

- Stack grows *down* towards lower addresses
- Buffer grows *up* towards higher addresses
- If we write past the end of the array, we overwrite data on the stack!

```
Enter input: helloabcdef
```

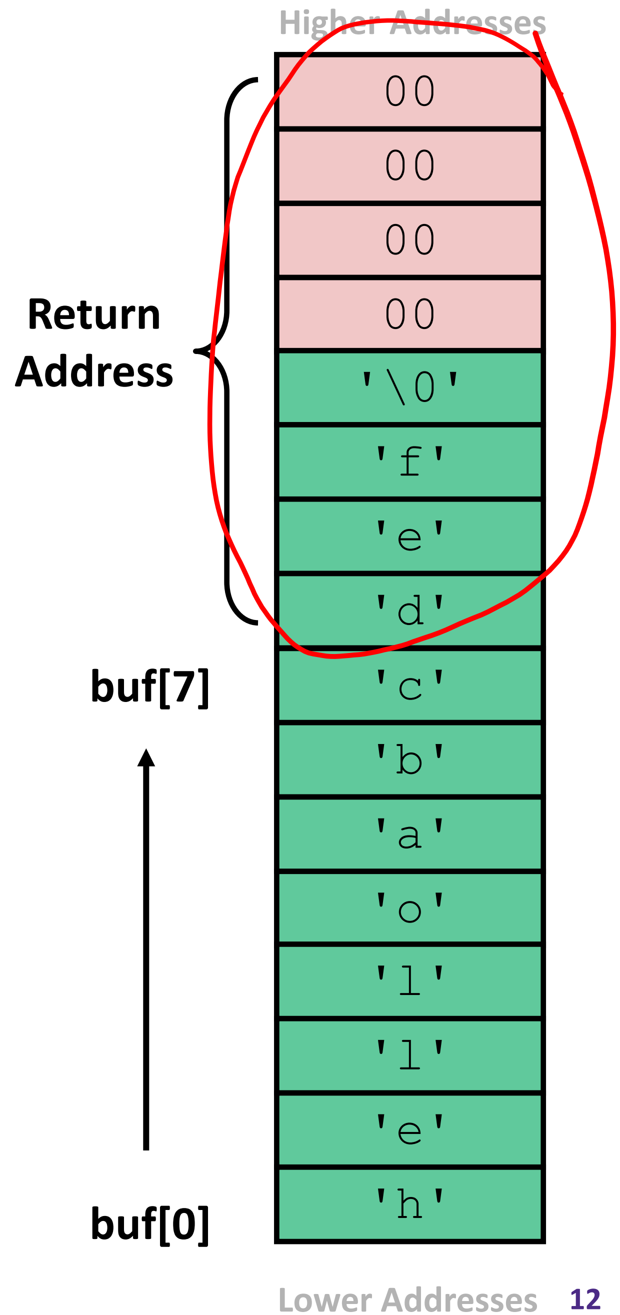


Buffer Overflow (3)

- Stack grows *down* towards lower addresses
- Buffer grows *up* towards higher addresses
- If we write past the end of the array, we overwrite data on the stack!

```
Enter input: helloabcdef
```

Buffer overflow! ☹



Why is Buffer Overflow a Problem?

- Buffer overflows on the stack can overwrite "interesting" data
  - Attackers just choose the right inputs
- Simplest form (sometimes called "stack smashing")
  - Unchecked length on string input into bounded array causes overwriting of stack data
  - Try to change the return address of the current procedure
- Why is this a big deal?
  - It was the #1 *technical* cause of security vulnerabilities
    - #1 *overall* cause is social engineering / user ignorance

String Library Code (1)

- 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 (don't know size!)

same as:

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

- What could go wrong in this code?

String Library Code (2)

- 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;
}
```

gcc: warning: the 'gets' function is dangerous and should not be used.

Try:
`man gets`

  - No way to specify **limit** on number of characters to read
- 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[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

input buffer

read input into buffer

print output from buffer

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

```
unix> ./buf-nsp
Enter string: 123456789012345
123456789012345
```

```
unix> ./buf-nsp
Enter string: 1234567890123456
Segmentation fault (core dumped)
```

Buffer Overflow Disassembly (buf-nsp)

echo:

```
0000000000401146 <echo>:
  401146:  48 83 ec 18          sub    $0x18,%rsp
    ...                          ... calls printf ...
  401159:  48 8d 7c 24 08       lea    0x8(%rsp),%rdi
  40115e:  b8 00 00 00 00       mov    $0x0,%eax
  401163:  e8 e8 fe ff ff       callq  401050 <gets@plt>
  401168:  48 8d 7c 24 08       lea    0x8(%rsp),%rdi
  40116d:  e8 be fe ff ff       callq  401030 <puts@plt>
  401172:  48 83 c4 18          add    $0x18,%rsp
  401176:  c3                   retq
```

call_echo:

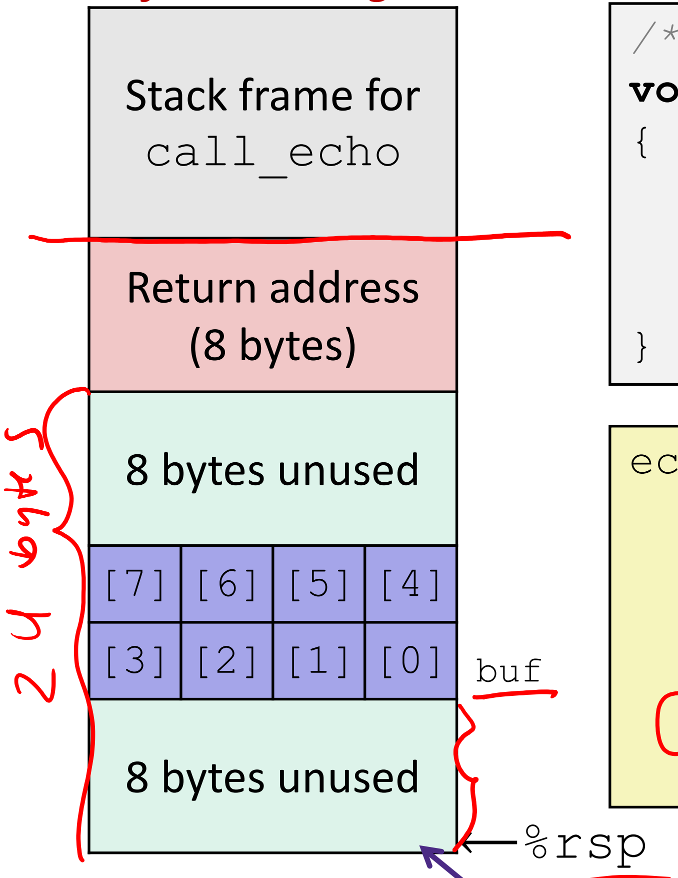
```
0000000000401177 <call_echo>:
  401177:  48 83 ec 08          sub    $0x8,%rsp
  40117b:  b8 00 00 00 00       mov    $0x0,%eax
  401180:  e8 c1 ff ff ff       callq  401146 <echo>
  401185:  48 83 c4 08          add    $0x8,%rsp
  401189:  c3                   retq
```

return address

Higher Addresses

Buffer Overflow Stack

Before call to gets




```
/* Echo Line */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq   $24, %rsp
   ...
  leaq   8(%rsp), %rdi
  mov    $0x0,%eax
  call   gets
   ...
```

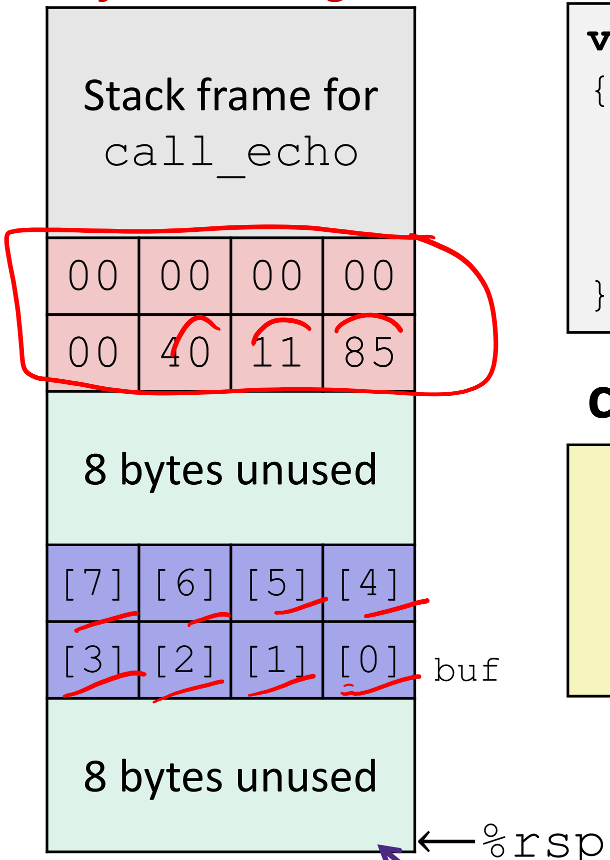
Note: addresses increasing right-to-left, bottom-to-top

Lower Addresses

Higher Addresses

Buffer Overflow Setup

Before call to gets




```
void echo()
{
    char buf[8];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
   ...
  leaq   8(%rsp), %rdi
  mov    $0x0,%eax
  call   gets
   ...
```

call_echo:

```
 . . .
 401180:   callq   401146 <echo>
 401185:   add     $0x8,%rsp
 . . .
```

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

Lower Addresses

Higher Addresses

Buffer Overflow Example #1: 1234567

Before call to gets

Stack frame for `call_echo`

00	00	00	00
00	40	11	85
8 bytes unused			
00	37	36	35
34	33	32	31
8 bytes unused			

buf

←%rsp

null character

0x31 = '1'

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

Lower Addresses

```
void echo()
{
    char buf[8];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
   ...
  leaq   8(%rsp), %rdi
  mov    $0x0,%eax
  call   gets
   ...
```

call_echo:

```
  . . .
  401180:  callq   401146 <echo>
  401185:  add     $0x8,%rsp
  . . .
```

```
unix> ./buf-nsp
Enter string: 1234567
1234567
```

All good! No overflow!

Higher Addresses

Buffer Overflow Example #2: 123456789012345

After call to gets

Stack frame for call_echo			
00	00	00	00
00	40	11	85
00	35	34	33
32	31	30	39
38	37	36	35
34	33	32	31
8 bytes unused			

buf

%rsp

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

Lower Addresses

```
void echo()
{
    char buf[8];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
   ...
  leaq   8(%rsp), %rdi
  mov    $0x0,%eax
  call   gets
   ...
```

call_echo:

```
  . . .
  401180:  callq   401146 <echo>
  401185:  add     $0x8,%rsp
  . . .
```

```
unix> ./buf-nsp
Enter string: 123456789012345
123456789012345
```

Overflowed buffer, but did not corrupt state

Higher Addresses

Buffer Overflow Example #3: 1234567890123456

After call to gets

Stack frame for `call_echo`			
00	00	00	00
00	40	11	00
36	35	34	33
32	31	30	39
38	37	36	35
34	33	32	31
8 bytes unused			

buf

←%rsp

```
void echo()
{
    char buf[8];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
    ...
   leaq   8(%rsp), %rdi
   mov    $0x0,%eax
   call   gets
    ...
```

call_echo:

```
  . . .
  401180:   callq   401146 <echo>
  401185:   add     $0x8,%rsp
  . . .
```

```
unix> ./buf-nsp
Enter string: 1234567890123456
Segmentation fault (core dumped)
```

Overflowed buffer and corrupted return pointer

Lower Addresses

Higher Addresses

Buffer Overflow Example #3 Explained

After return from echo

Stack frame for call_echo				
				← %rsp
00	00	00	00	
00	40	11	00	
36	35	34	33	
32	31	30	39	
38	37	36	35	
34	33	32	31	buf
8 bytes unused				

```
00000000004010d0 <register_tm_clones>:
  4010d0:  lea    0x2f61(%rip),%rdi
  4010d7:  lea    0x2f5a(%rip),%rsi
  4010de:  sub    %rdi,%rsi
  4010e1:  mov    %rsi,%rax
  4010e4:  shr    $0x3f,%rsi
  4010e8:  sar    $0x3,%rax
  4010ec:  add    %rax,%rsi
  4010ef:  sar    %rsi
  4010f2:  je     401108
  4010f4:  mov    0x2efd(%rip),%rax
  4010fb:  test   %rax,%rax
  4010fe:  je     401108
  401100:  jmpq   *%rax
  401102:  nopw   0x0(%rax,%rax,1)
  401108:  retq
```

"Returns" to a valid instruction, but bad indirect jump so program signals `SIGSEGV, Segmentation fault`

Lower Addresses

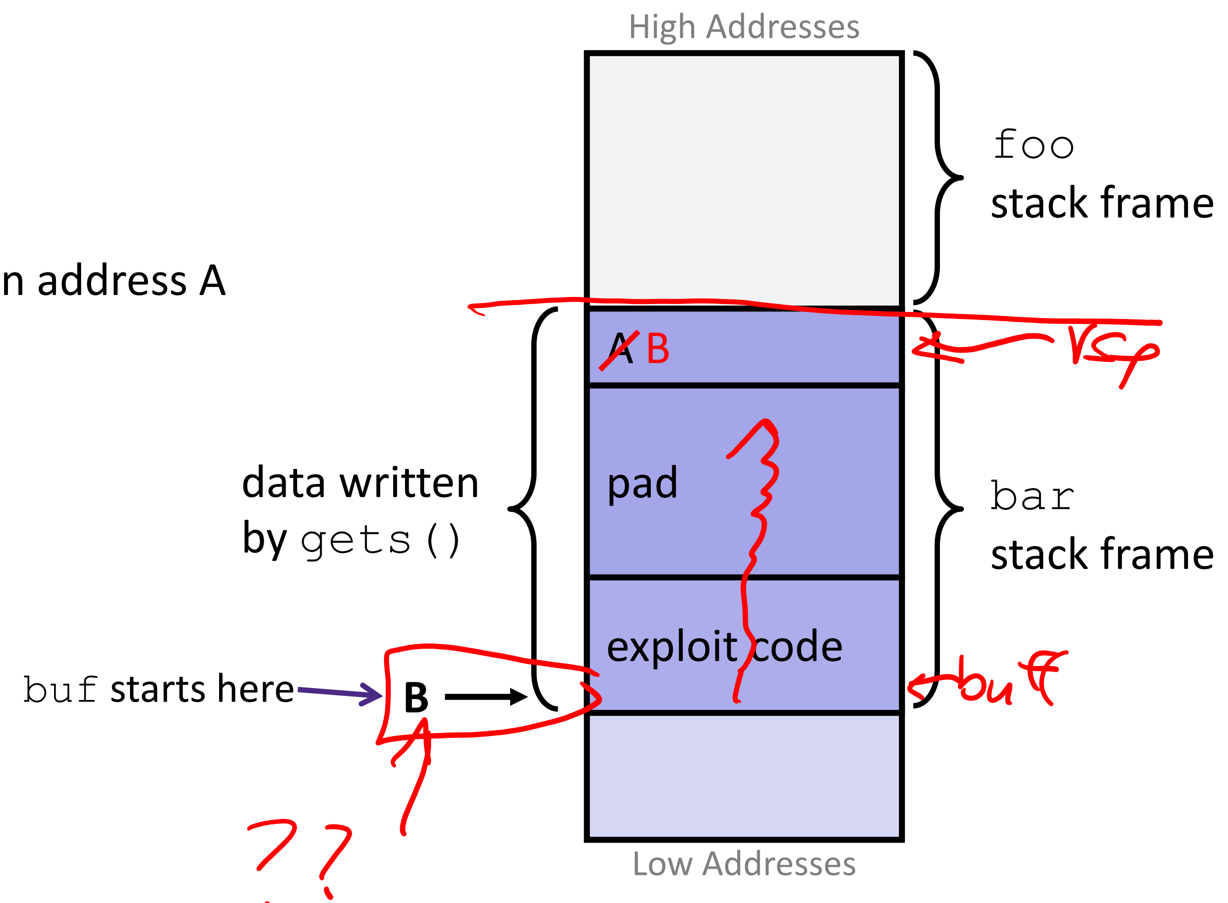
Malicious Use of Buffer Overflow: Code Injection Attacks

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

return address A

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

Stack after call to `gets()`

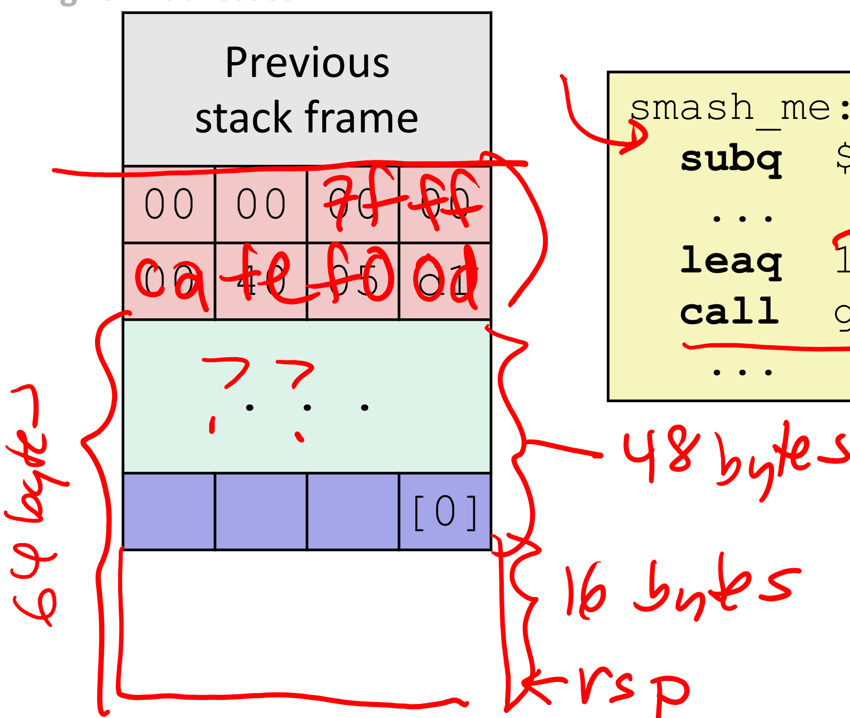


- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- **When `bar()` executes `ret`, will jump to exploit code**

Practice Question

- `smash_me` is vulnerable to stack smashing!
- What is the <u>minimum</u> number of characters that `gets` must read in order for us to change the return address to a <u>stack address</u>? (e.g. 0x00 00 7f ff ca fe f0 0d)

Higher Addresses



64 bytes

```
smash_me:
   subq   $0x40, %rsp
     ...
   leaq   16(%rsp), %rdi
   call   gets
     ...
```

A. 27

B. 30

C. 51

D. 54

E. We're lost…

Lower Addresses

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

Exploits Based on Buffer Overflows

Buffer overflow bugs can allow attackers to execute arbitrary code on victim machines

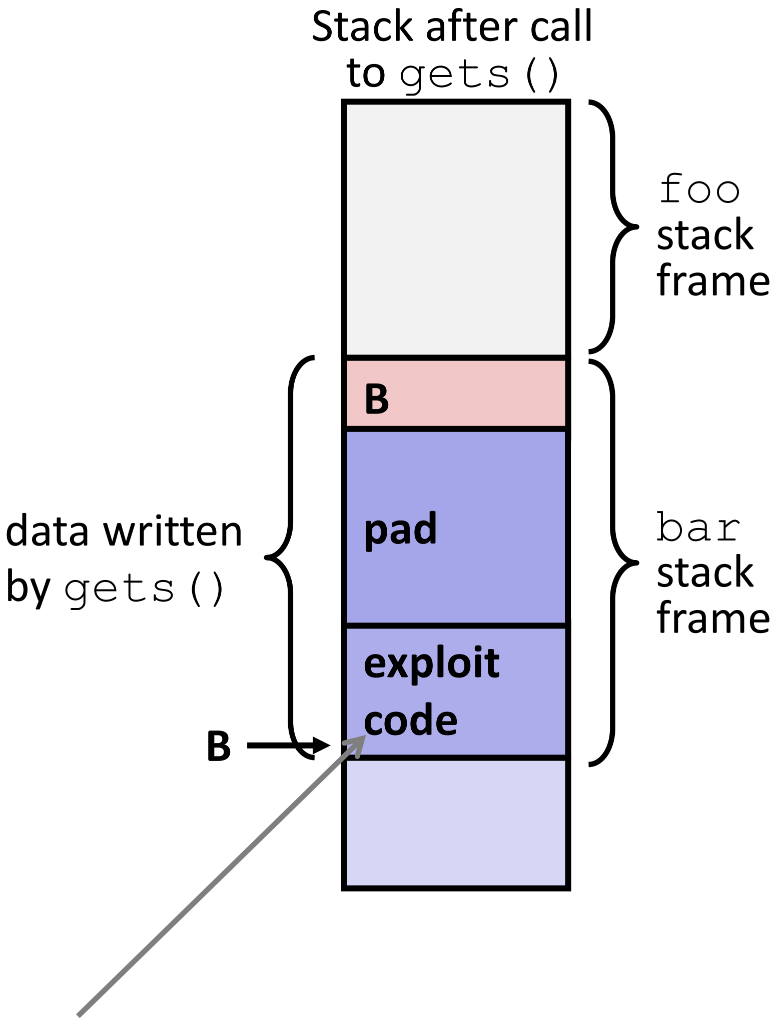
- Distressingly common in real programs
  - Programmers keep making the same mistakes ☹
  - Recent measures make these attacks much more difficult
- Examples across the decades
  - Original “Internet worm” (1988)
  - Heartbleed (2014, affected 17% of servers)
    - Similar issue in Cloudbleed (2017)
  - Hacking embedded devices
    - Cars, Smart homes, Planes

Dealing with buffer overflow attacks

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

1) System-Level Protections (1)

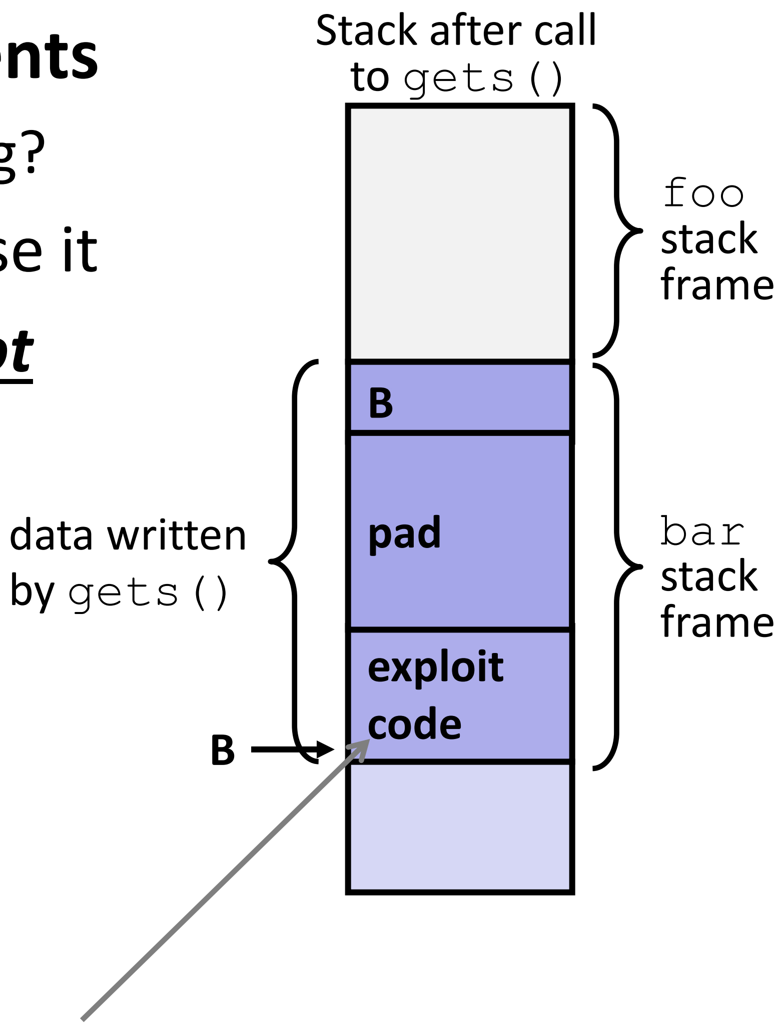
- **Non-executable code segments**
- In traditional x86, can mark region of memory as either "read-only" or "writeable"
  - Can execute anything readable
- x86-64 added explicit "execute" permission
- **Stack marked as non-executable**
  - Do *NOT* execute code in Stack, Static Data, or Heap regions
  - Hardware support needed



Any attempt to execute this code will fail

1) System-Level Protections (2)

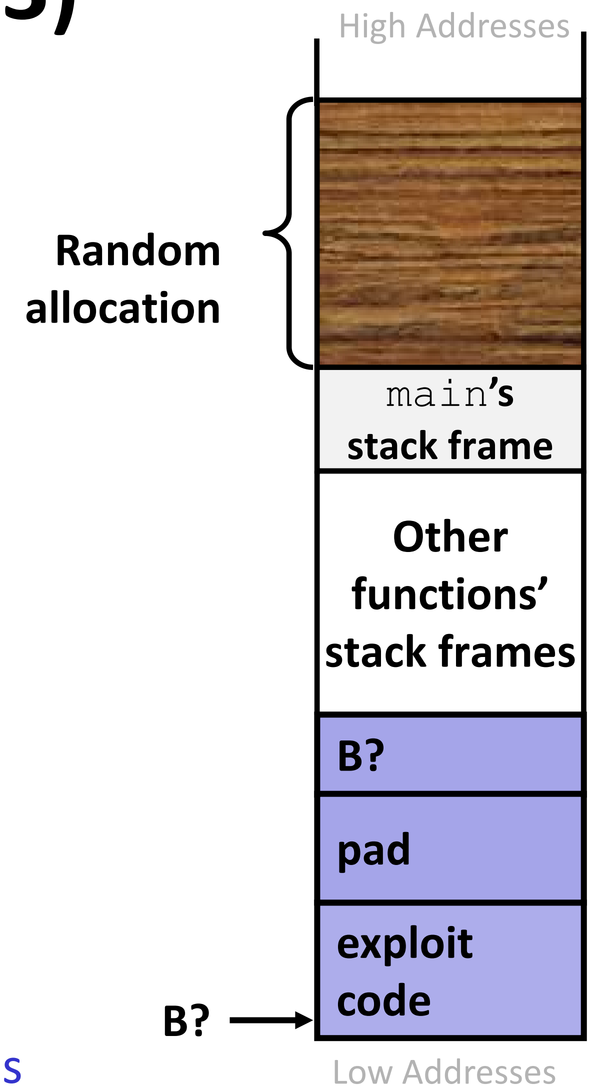
- **Non-executable code segments**
  - Wait, doesn't this fix everything?
- Works well, but can't always use it
- Many embedded devices ***do not*** have this protection
  - *e.g.*, cars, smart homes, pacemakers
- Some exploits still work!
  - Return-oriented programming
  - Return to libc attack
  - JIT-spray attack



Any attempt to execute this code will fail

1) System-Level Protections (3)

- **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: Address of variable `local` for when Slide 5 code executed 3 times:
    - `0x7ffd19d3f8ac`
    - `0x7ffe8a462c2c`
    - `0x7ffe927c905c`
  - Stack repositioned each time program executes



2) Stack Canaries

- Basic Idea: place special value ("canary") on stack just beyond buffer
  - ***Secret*** value that is randomized before main()
  - Placed <u>between</u> buffer and return address
  - Check for corruption before exiting function
- GCC implementation
  - `-fstack-protector`

```
unix>./buf
Enter string: 12345678
12345678
```

```
unix> ./buf
Enter string: 123456789
*** stack smashing detected ***
```

Protected Buffer Disassembly (buf)

This is extra (non-testable) material

try: diff buf-nsp.s buf.s

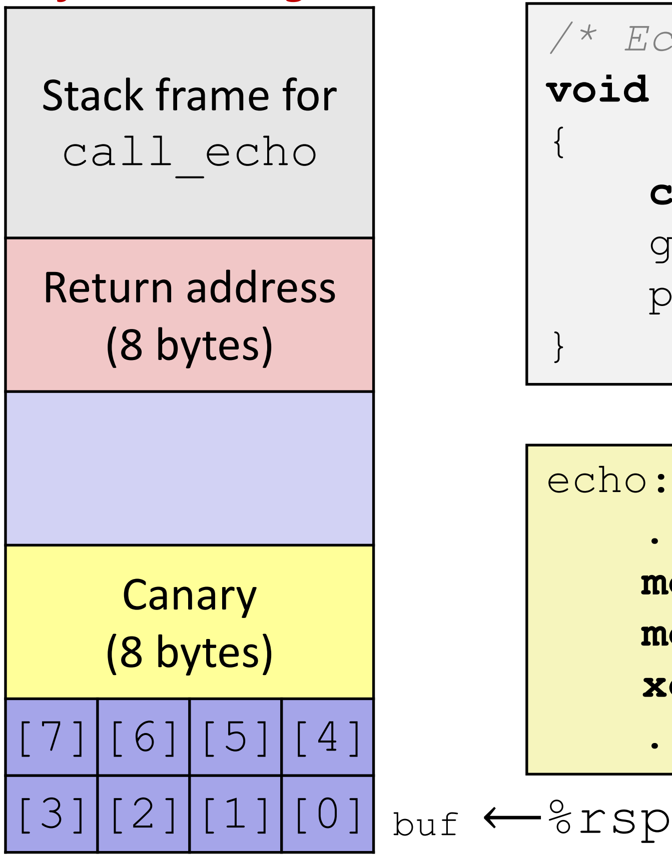
echo:

```
401156:  push   %rbx
401157:  sub    $0x10,%rsp
40115b:  mov    $0x28,%ebx
401160:  mov    %fs:(%rbx),%rax    # read canary value
401164:  mov    %rax,0x8(%rsp)     # store canary on stack
401169:  xor    %eax,%eax          # erase canary from register
...      ... call printf ...
40117d:  callq  401060 <gets@plt>
401182:  mov    %rsp,%rdi
401185:  callq  401030 <puts@plt>
40118a:  mov    0x8(%rsp),%rax     # read current canary on stack
40118f:  xor    %fs:(%rbx),%rax    # compare against original value
401193:  jne    40119b <echo+0x45> # if unchanged, then return
401195:  add    $0x10,%rsp
401199:  pop    %rbx
40119a:  retq
40119b:  callq  401040 <__stack_chk_fail@plt>  # stack smashing detected
```

This is extra (non-testable) material

Setting Up Canary

Before call to gets




```
/* Echo Line */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

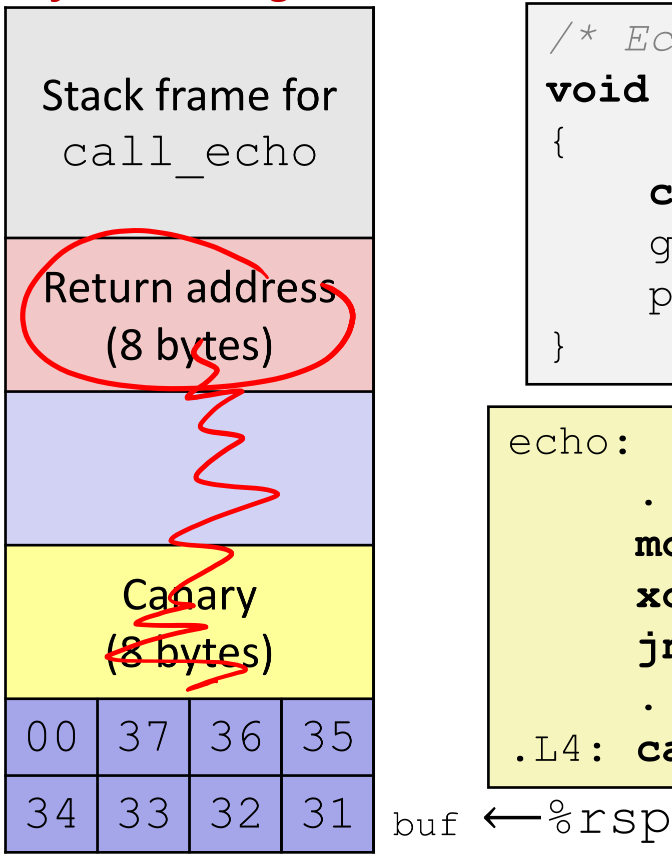
Segment register ***(don't worry about it)***

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

Checking Canary

This is extra (non-testable) material

After call to gets




```
/* Echo Line */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
     . . .
     movq   8(%rsp), %rax     # retrieve from Stack
     xorq   %fs:40, %rax      # compare to canary
     jne    .L4               # if not same, FAIL
     . . .
.L4: call   __stack_chk_fail
```

Input: *1234567*

3) Avoid Overflow Vulnerabilities in Code (1)

```
/* Echo Line */
void echo()
{
    char buf[8];  /* Way too small! */
    fgets(buf, 8, 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 `%ns` where `n` is a suitable integer

3) Avoid Overflow Vulnerabilities in Code (2)

- Alternatively, don't use C - use a language that does array index bounds check
  - Buffer overflow is impossible in Java
    - `ArrayIndexOutOfBoundsException`
  - Rust language was designed with security in mind
    - Panics on index out of bounds, plus more protections

Summary of Prevention Measures

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

2) Have compiler use “stack canaries”

3) Avoid overflow vulnerabilities
   - Use library routines that limit string lengths
   - Use a language that makes them impossible

Think this is cool?

- You'll love Lab 3 😉
  - Some parts *must* be run through GDB to disable certain security features
- Take CSE 484 (Security)
  - Several different kinds of buffer overflow exploits
  - Many ways to counter them
- Nintendo fun!
  - Using glitches to rewrite code: https://www.youtube.com/watch?v=TqK-2jUQBUY
  - Flappy Bird in Mario: https://www.youtube.com/watch?v=hB6eY73sLV0

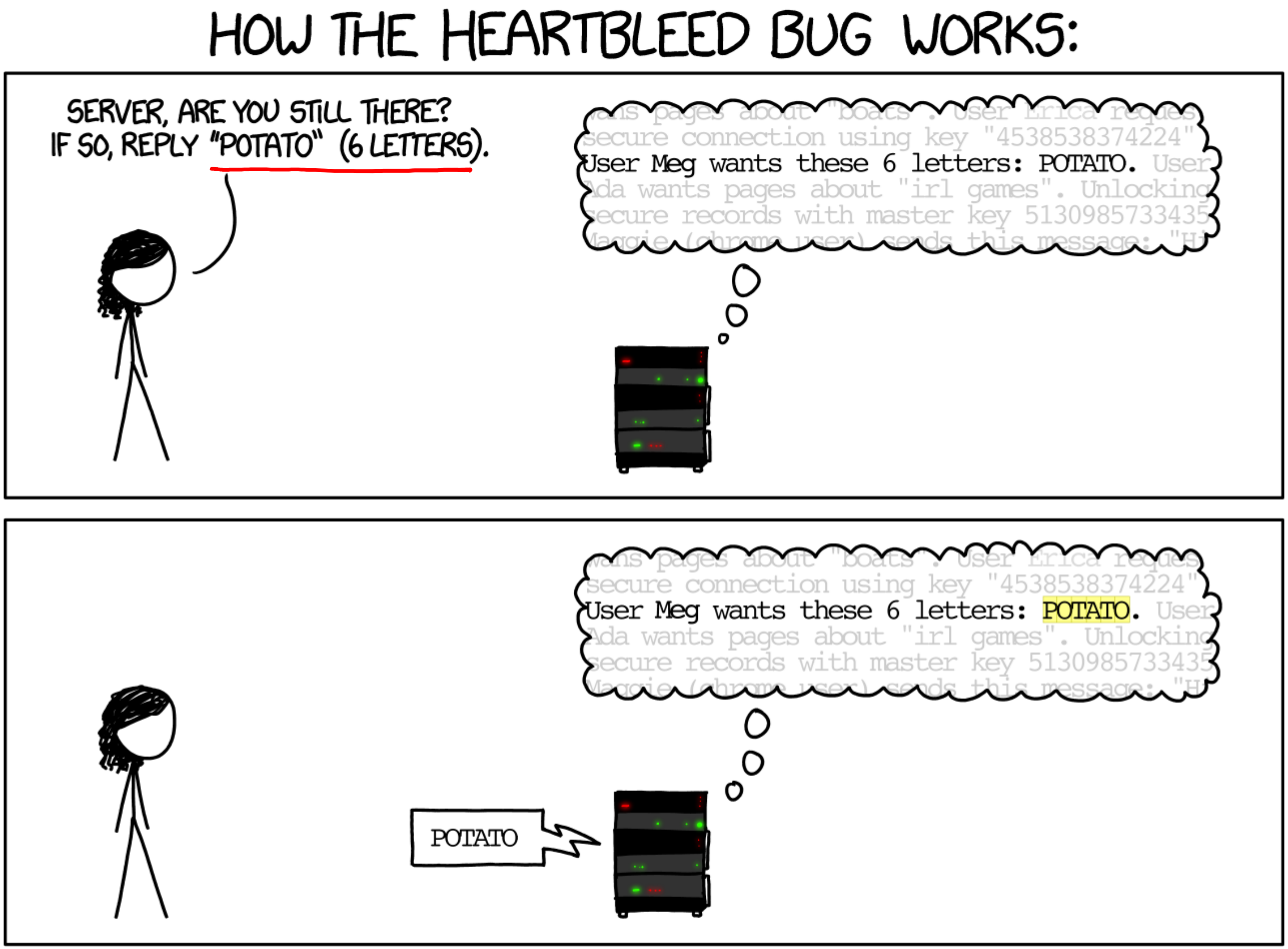
Example: the original Internet worm (1988)

- 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 with phony argument:
    - `finger "`*`exploit-code padding new-return-addr`*`"`
    - Exploit code: executed a root shell on the victim machine with a direct connection to the attacker
- 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 author of the worm (Robert Morris*) was prosecuted…

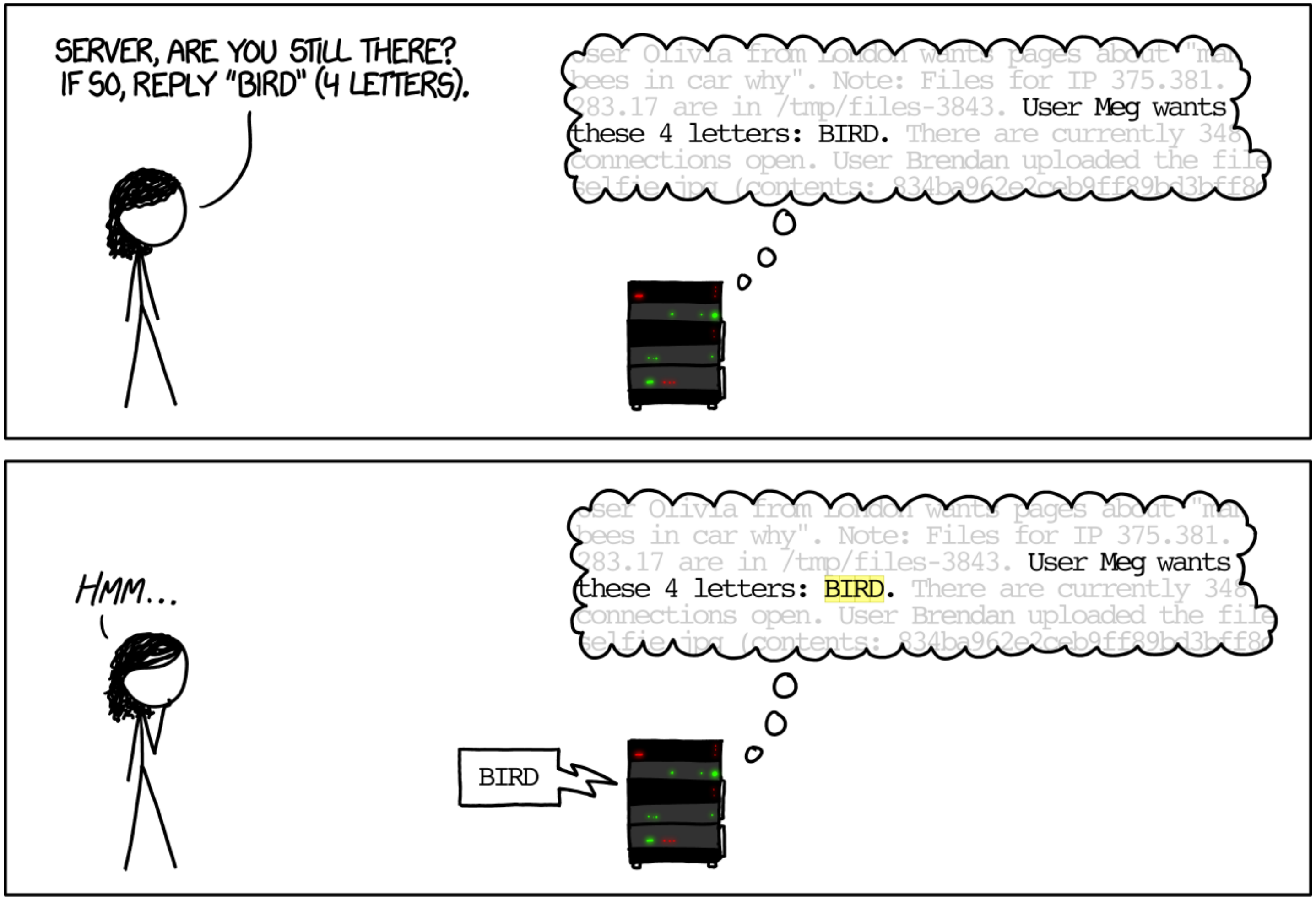
Hacking Cars (2010)

- UW CSE research demonstrated wirelessly hacking a car using buffer overflow
  - http://www.autosec.org/pubs/cars-oakland2010.pdf
- Overwrote the onboard control system's code
  - Disable brakes, unlock doors, turn engine on/off

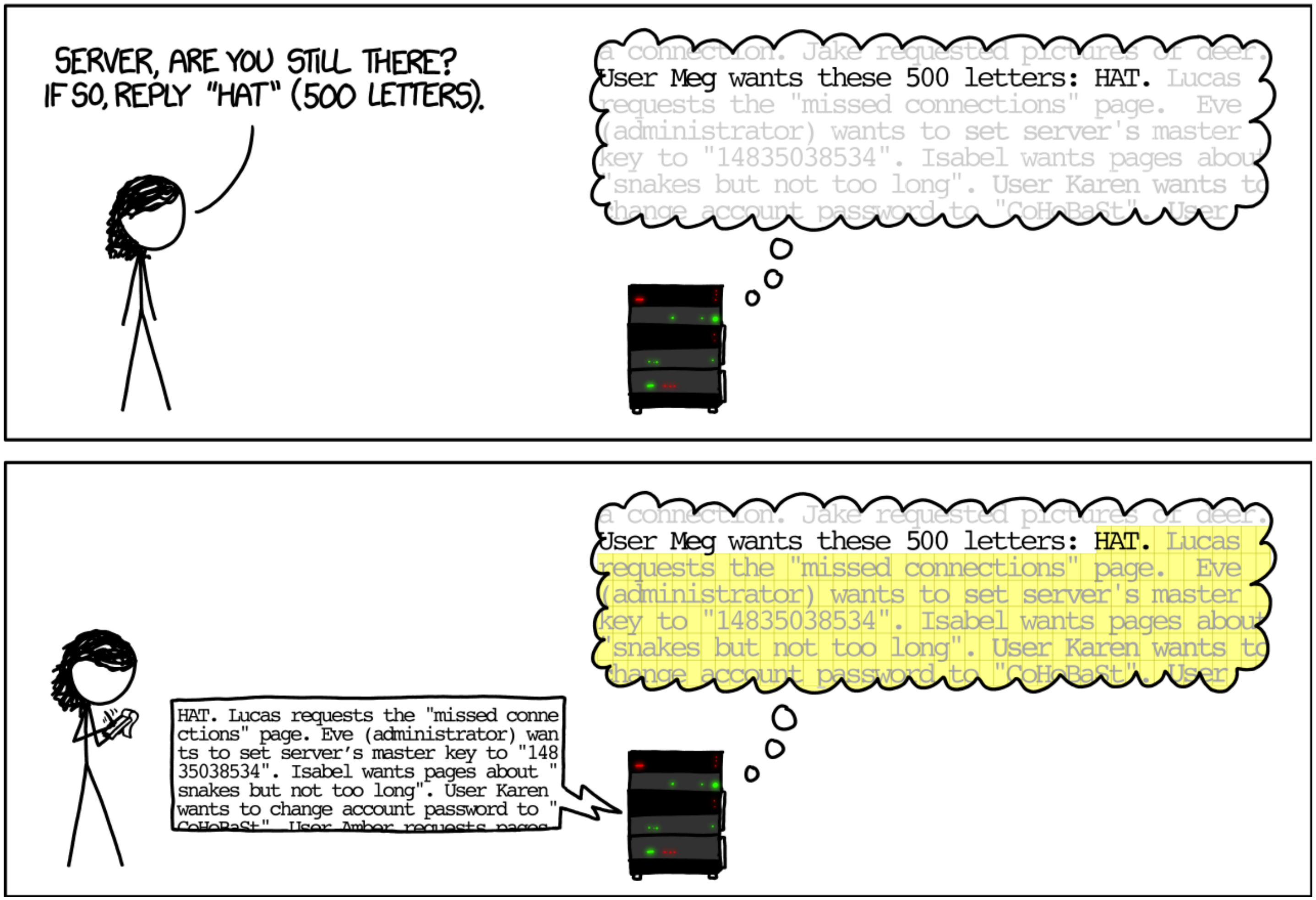
Example: Heartbleed (2014) (1)



Example: Heartbleed (2014) (2)

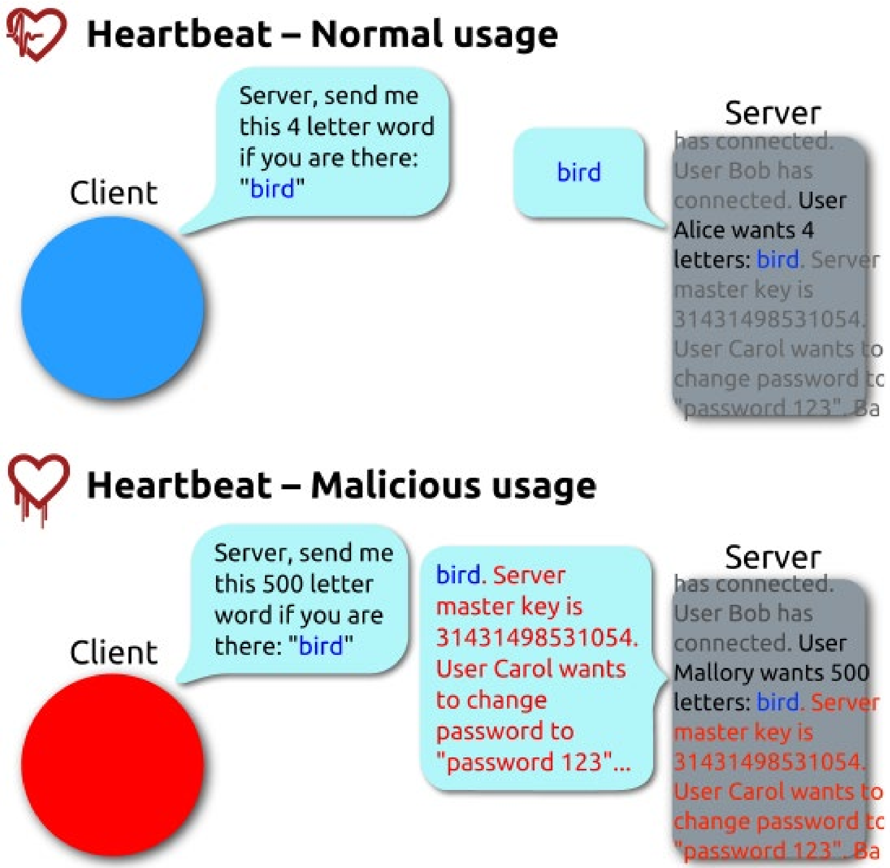


Example: Heartbleed (2014) (3)



Heartbleed Details

- 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, ...



By FenixFeather - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=32276981