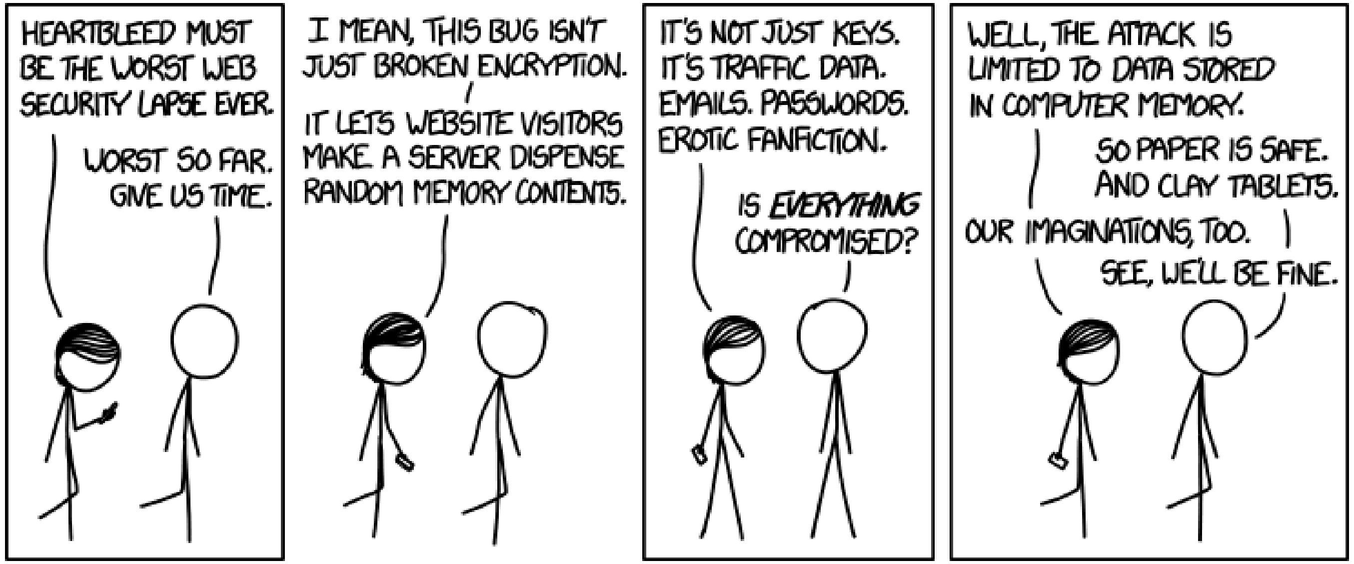
# The Hardware/Software Interface

## Buffer Overflow

**Instructors:**
Amber Hu, Justin Hsia

**Teaching Assistants:**
Anthony Mangus
Grace Zhou
Jiuyang Lyu
Kurt Gu
Mendel Carroll
Naama Amiel
Rose Maresh
Violet Monserate
Divya Ramu
Jessie Sun
Kanishka Singh
Liander Rainbolt
Ming Yan
Pollux Chen
Soham Bhosale




**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/

# Relevant Course Information

- HW13 due tonight, HW14 due Friday, HW15 due Monday
- Mid-quarter survey due tomorrow (10/30)

- Midterm grades to be released next week
  - Solutions posted on website soon; rubric and grades will be found on Gradescope
  - Regrade requests will be open for a short time after grade release
  - Don’t freak out about your grade – a learning opportunity and not a reflection of you as a person

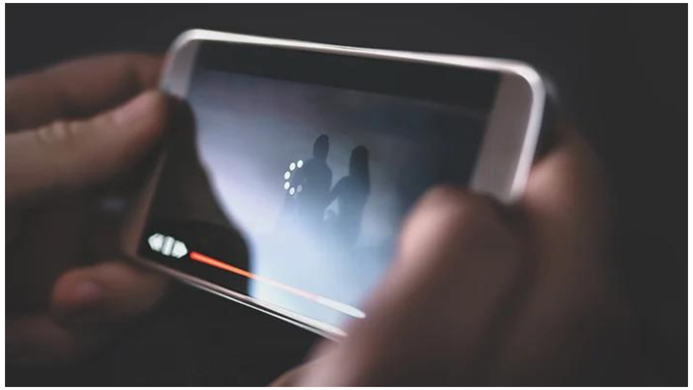
- Lab 3 released today, due next Friday (11/7)
  - You will have everything you need by the end of this lecture

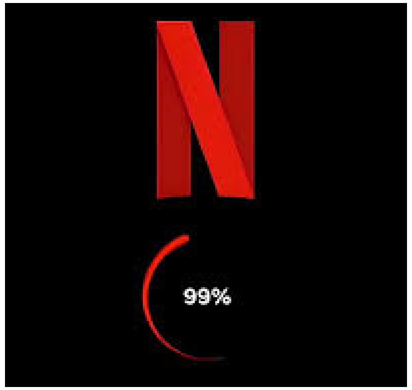
# Lecture Outline (1/5)

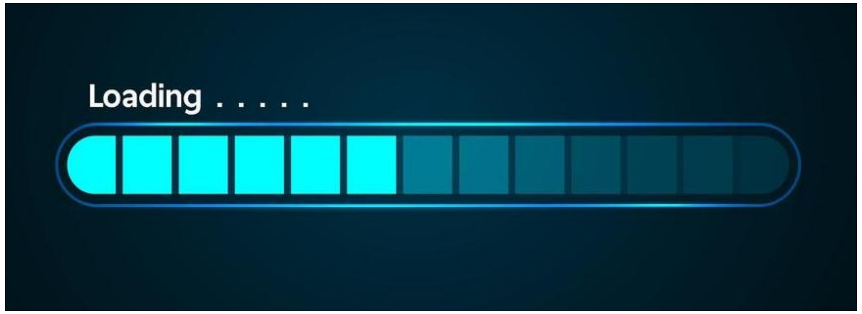
- **Buffer Overflow**
- Vulnerable Code
- Code Injection Attacks
- Dealing with Buffer Overflow Attacks
- Real Life Examples of Buffer Overflow

# What Is a Buffer?

- A buffer is an array used to temporarily store data
- You've probably seen buffering symbols for streaming video and games
  - The video/game data is being written into a buffer before being shown



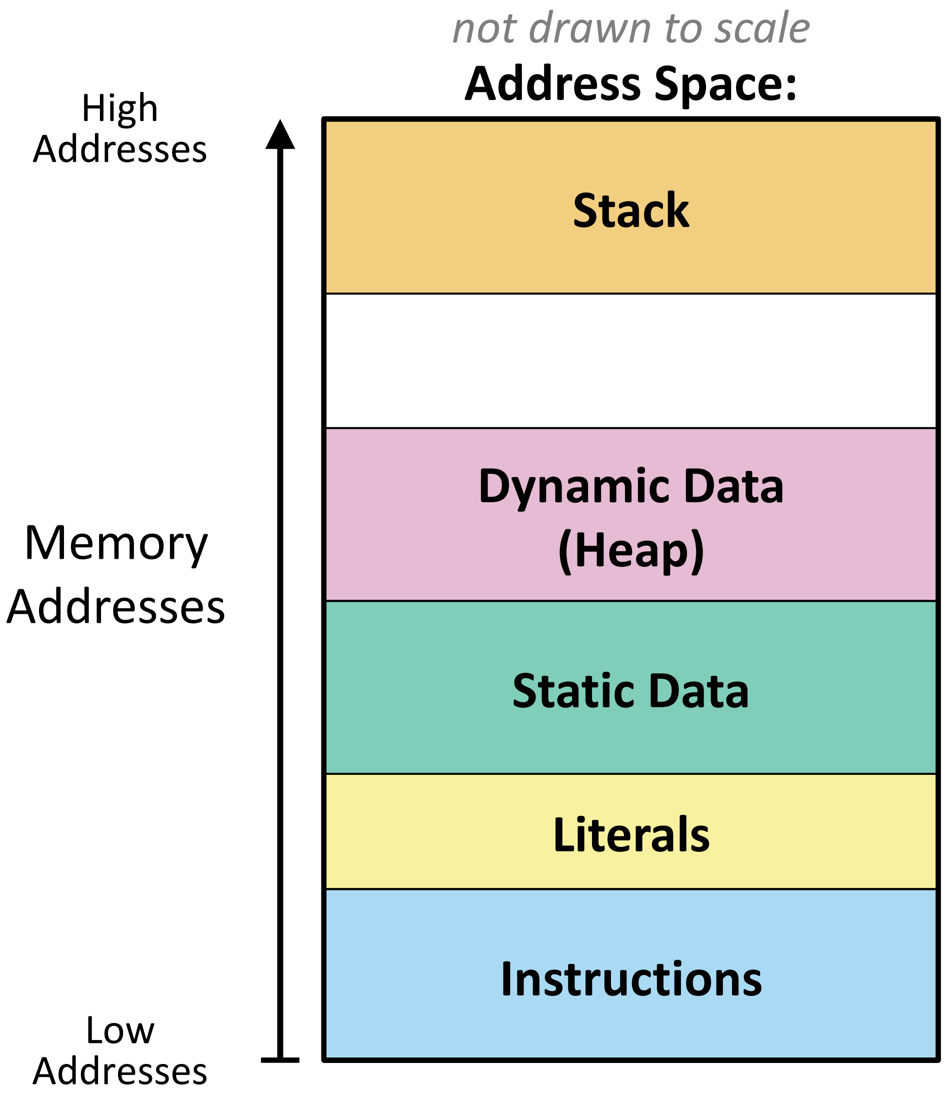



- Buffers can also store user input for programs…

# General Memory Layout (Reminder)

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



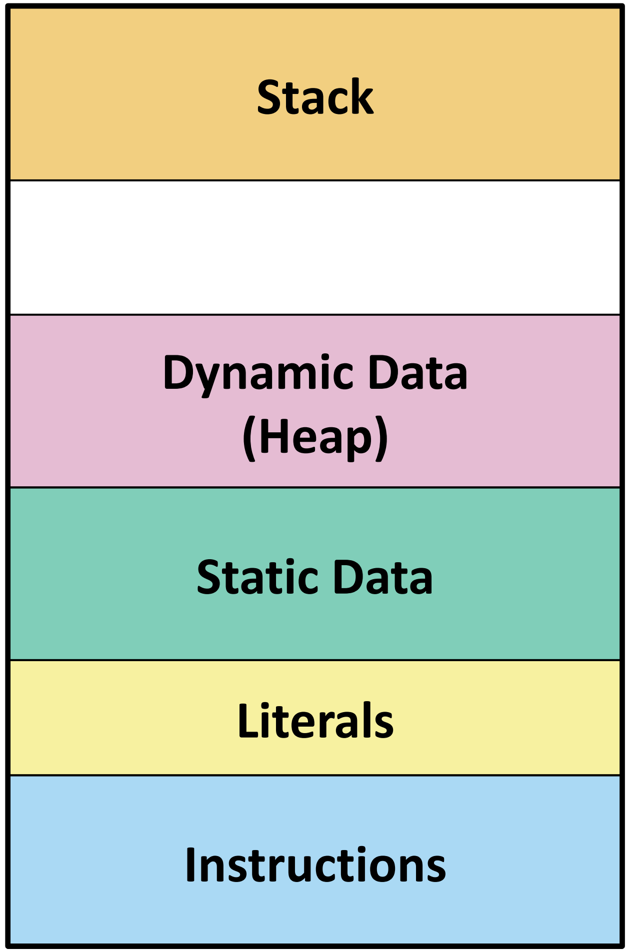
# Memory Allocation Example

- Where does everything go?

```
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 ... */
}
```

*not drawn to scale*

**Address Space:**



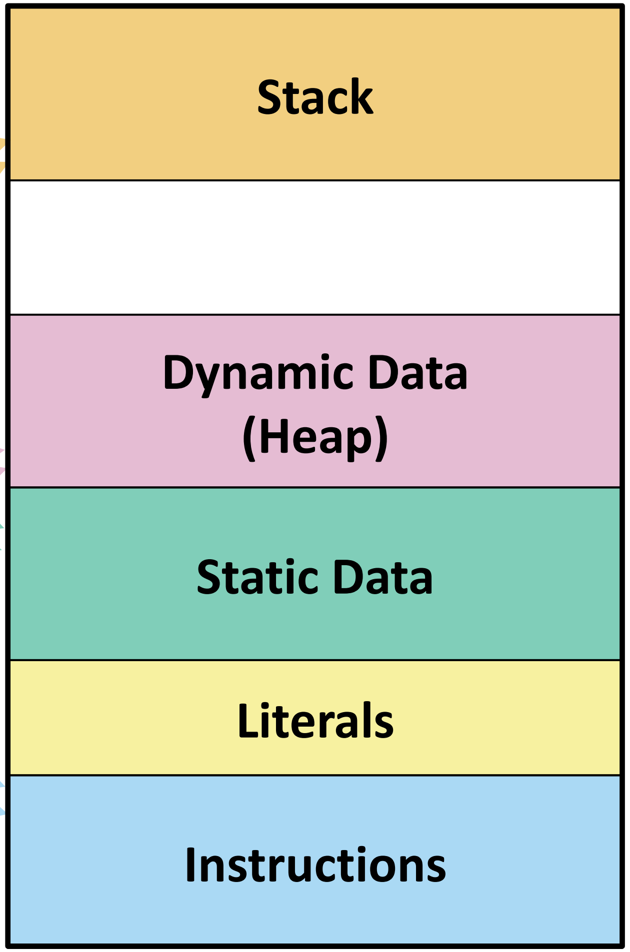
# Memory Allocation Example Solution

- Where does everything go?

```
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 ... */
}
```

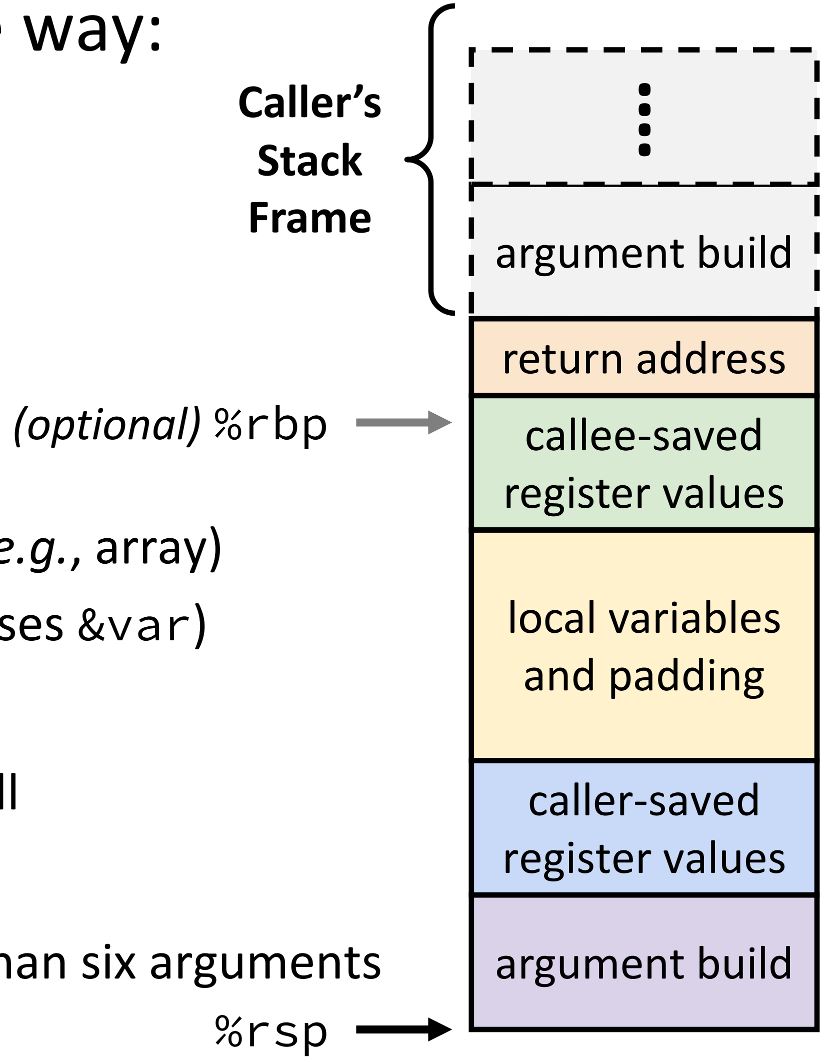
*not drawn to scale*

**Address Space:**



# x86-64 Stack Frame Structure (Reminder)

- Each stack frame organized in the same way:
  1) Return address pushed by `call`
     - The address of the instruction after `call`
  2) Callee-saved registers
     - Only if procedure modifies/uses them
  3) Local variables
     - Unavoidable if variable is too big for a register (*e.g.*, array)
     - Unavoidable if variable needs an address (*i.e.*, uses `&var`)
  4) Caller-saved registers
     - Only if values are needed *across* a procedure call
  5) Argument build
     - Only if procedure calls a procedure with more than six arguments

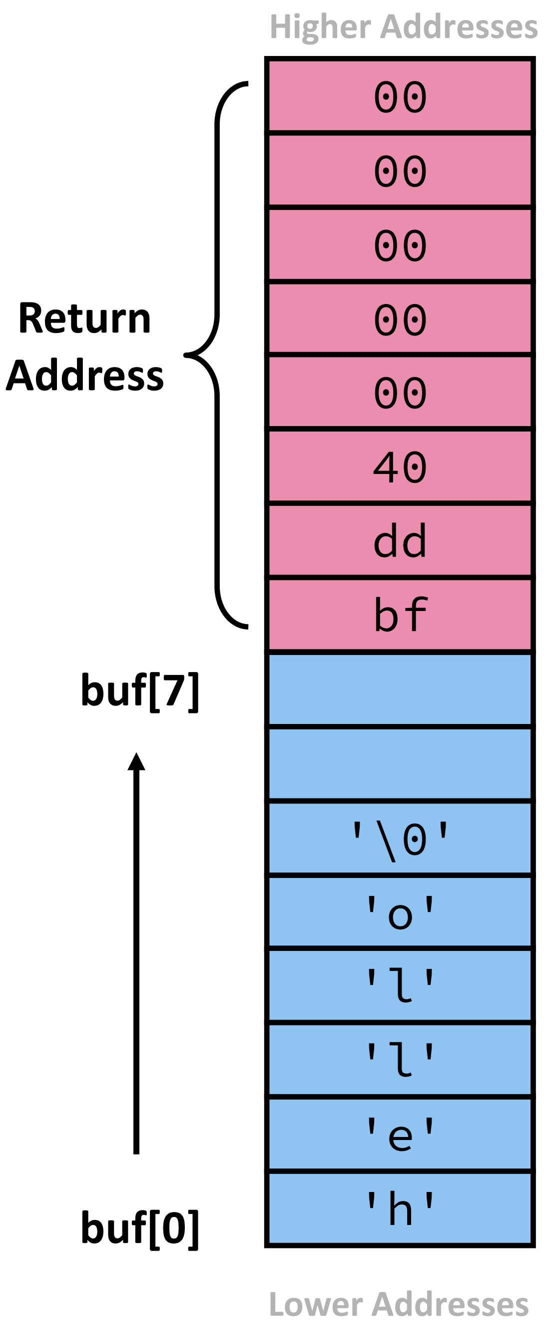


# Buffer Overflow in a Nutshell (Review)

- 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 Example (1/3)

- 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!

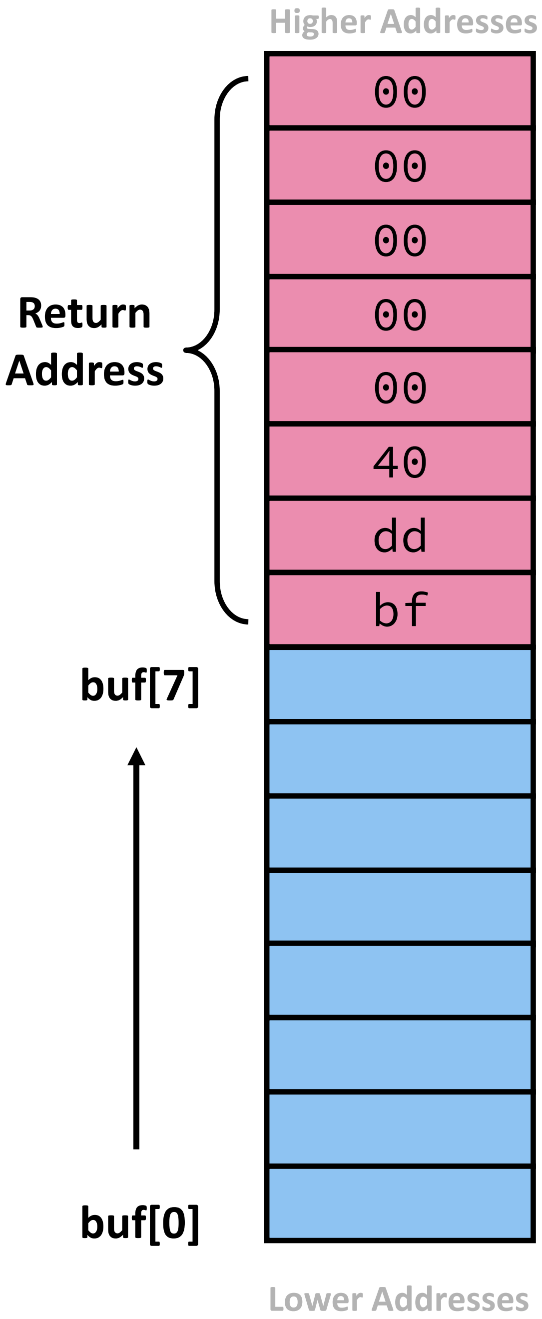



```
Enter input: hello
```

**No overflow ☺**

# Buffer Overflow Example (2/3)

- 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!



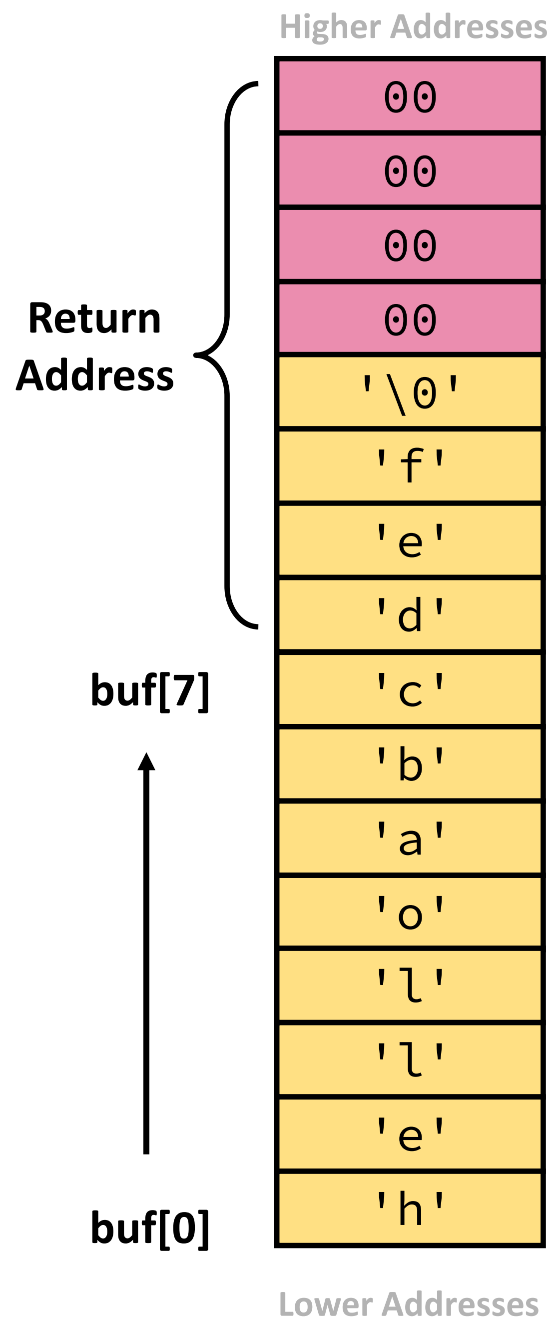

```
Enter input: helloabcdef
```

# Buffer Overflow Example (3/3)

- 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 or user ignorance

# Lecture Outline (2/5)

- Buffer Overflow
- **Vulnerable Code**
- Code Injection Attacks
- Dealing with Buffer Overflow Attacks
- Real Life Examples of Buffer Overflow

# String Library Code (1/2)

- 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;
}
```

pointer to start of an array

same as:

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

  - What could go wrong in this code?

# String Library Code (2/2)

- 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;
}
```

  - 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 destination
  - `scanf`, `fscanf`, `sscanf`, when given %s specifier

# Vulnerable Buffer Code Demonstration

```
/* Echo Line */
void echo() {
    char buf[16];  // Way too small!
    gets(buf);
    puts(buf);
}
```

```
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:

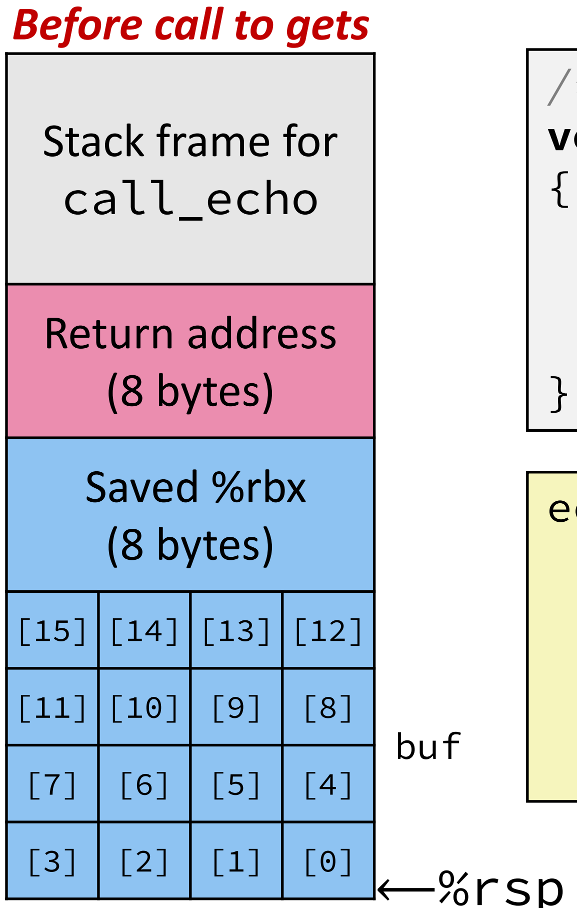
```
0000000000401146 <echo>:
  401146:   53                     push   %rbx
  401147:   48 83 ec 20            sub    $0x10,%rsp
  40114b:   48 89 e7               mov    %rsp,%rdi
  40114e:   e8 fd fe ff ff         call   401050 <gets@plt>
  401153:   48 89 e7               mov    %rsp,%rdi
  401156:   e8 d5 fe ff ff         call   401030 <puts@plt>
  40115b:   48 83 c4 20            add    $0x10,%rsp
  40115f:   5b                     pop    %rbx
  401160:   c3                     ret
```

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

# Buffer Overflow Demo Stack



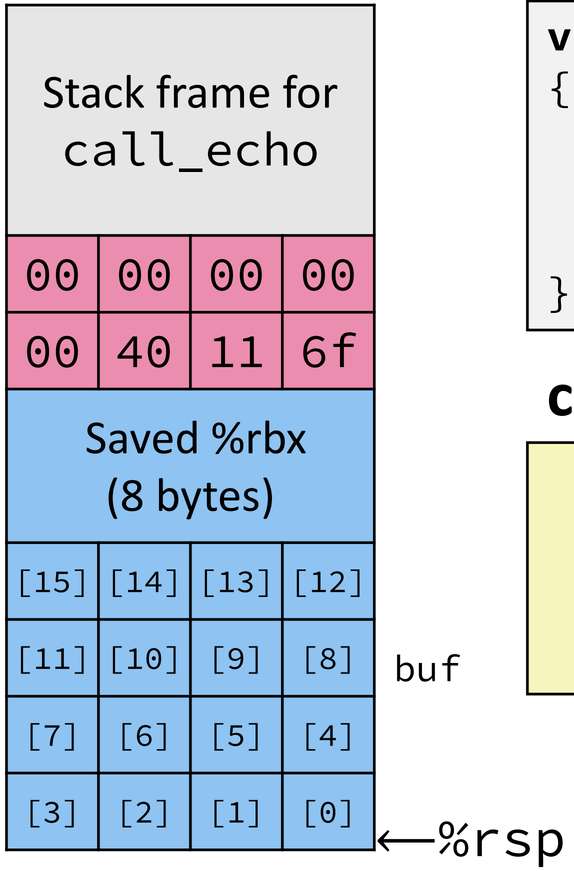

```
/* Echo Line */
void echo()
{
    char buf[16];  // Way too small!
    gets(buf);
    puts(buf);
}
```

```
echo:
  pushq %rbx
  subq  $16, %rsp
  movq  %rsp, %rdi
  call  gets
   ...
```

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

# Buffer Overflow Demo Setup

***Before call to gets***




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

```
echo:
  pushq %rbx
  subq  $16, %rsp
  movq  %rsp, %rdi
  call  gets
   ...
```

**call_echo:**

```
  . . .
  40116a:  callq  401146 <echo>
  40116f:  add    $0x8,%rsp
  . . .
```

# Buffer Overflow Demo Input #1: 24 bytes

*After call to gets*

| Stack frame for call_echo | | | |
|---|---|---|---|
| 00 | 00 | 00 | 00 |
| 00 | 40 | 11 | 6f |
| 00 | 33 | 32 | 31 |
| 30 | 39 | 38 | 37 |
| 36 | 35 | 34 | 33 |
| 32 | 31 | 30 | 39 |
| 38 | 37 | 36 | 35 |
| 34 | 33 | 32 | 31 |

buf

←%rsp

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

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

```
echo:
  pushq %rbx
  subq  $16, %rsp
  movq  %rsp, %rdi
  call  gets
  ...
```

**call_echo:**

```
  . . .
  40116a:  callq   401146 <echo>
  40116f:  add     $0x8,%rsp
  . . .
```

```
unix> ./buf-nsp
Enter string: 12345678901234567890123
12345678901234567890123
```

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

# Buffer Overflow Demo Input #2: 1234567890123456

*After call to gets*

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

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

```
echo:
  pushq %rbx
  subq  $16, %rsp
  movq  %rsp, %rdi
  call  gets
   ...
```

## call_echo:

```
   . . .
   40116a:  callq   401146 <echo>
   40116f:  add     $0x8,%rsp
   . . .
```

```
unix> ./buf-nsp
Enter string: 12345678901234567890123**4**
Segmentation fault (core dumped)
```

**Overflowed buffer and corrupted return pointer**

# Buffer Overflow Demo Input #2 Explained

*After return from echo*

| | | | | |
|---|---|---|---|---|
| Stack frame for call_echo | | | | |
| 00 | 00 | 00 | 00 | ←%rsp |
| 00 | 40 | 11 | **00** | |
| 34 | 33 | 32 | 31 | |
| 30 | 39 | 38 | 37 | |
| 36 | 35 | 34 | 33 | |
| 32 | 31 | 30 | 39 | buf |
| 38 | 37 | 36 | 35 | |
| 34 | 33 | 32 | 31 | |

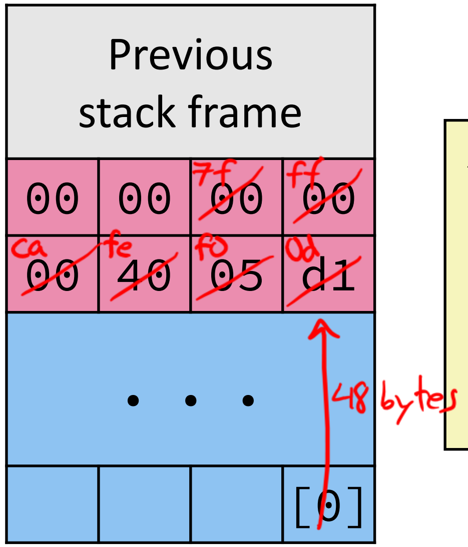
```
00000000004010d0 <register_tm_clones>:
  4010d0:   lea    0x2f61(%rip),%rdi
  4010d7:   lea    0x2f5a(%rip),%rsi
  4010de:   sub    %rdi,%rsi
  4010e1:   mov    %rsi,%rax
  4010e4:   shr    $0x3f,%rsi
  4010e8:   sar    $0x3,%rax
  4010ec:   add    %rax,%rsi
  4010ef:   sar    %rsi
  4010f2:   je     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`

# Polling Question

- `smash_me` is vulnerable to stack smashing!
- What is the minimum number of characters that `gets` must read in order for us to change the return address to a stack address?
  - For example: (`0x 00 00 7f ff ca fe f0 0d`)

always 0's    6 bytes of data




get to ret addr
64 bytes − 16 + 6
overwrite ret addr

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

A. 27
B. 30
C. 51
D. $\boxed{54}$
E. We're lost…

# Lecture Outline (3/5)

- Buffer Overflow
- Vulnerable Code
- **Code Injection Attacks**
- Dealing with Buffer Overflow Attacks
- Real Life Examples of Buffer Overflow

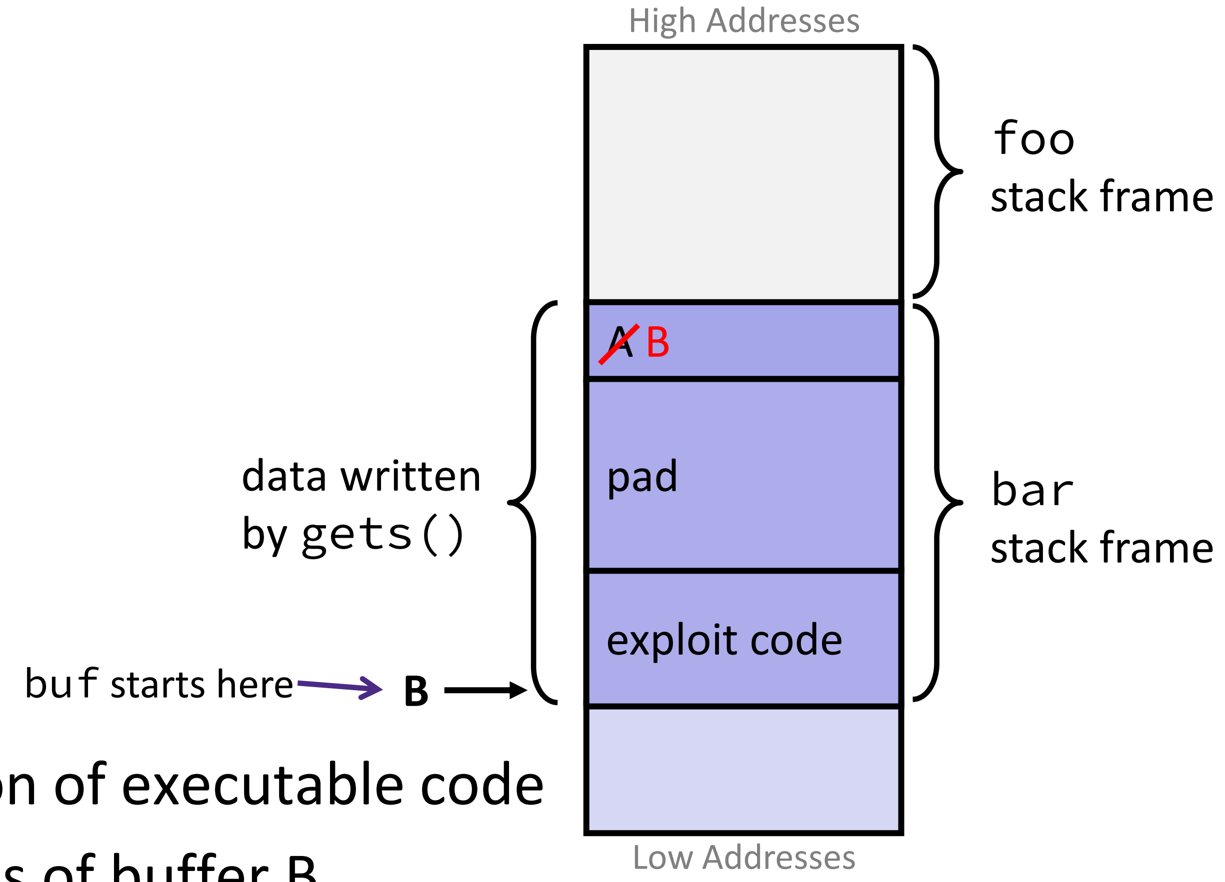
# Code Injection Attacks (Review)

- Malicious use of buffer overflow!

```
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

# Lecture Outline (4/5)

- Buffer Overflow
- Vulnerable Code
- Code Injection Attacks
- **Dealing with Buffer Overflow Attacks**
- Real Life Examples of Buffer Overflow

# 1) Avoid Overflow Vulnerabilities in Code – Functions

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

- 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 (*e.g.*, `%10s`)

# 1) Avoid Overflow Vulnerabilities in Code – Language

- Avoid C – use a language that checks array index bounds
  - Java ("high-level"): buffer overflow impossible because of `ArrayIndexOutOfBoundsException`
  - Rust ("low-level"): designed with safety and concurrency in mind

- If you need to use C to manually manipulate memory (*e.g.*, microprocessors or embedded systems), critical systems often have other methods of formally verifying program behavior

# 2) Stack Canaries

- Idea: place special value ("canary") on stack just beyond the buffer
  - *Secret* value that is randomized before `main()` executes and placed between the buffer and the stack frame's return address
  - Check for corruption before exiting function!

https://archive-share.america.gov/english-idiom-canary-coal-mine/index.html

- GCC implementation uses `-fstack-protector` compiler flag

```
unix>./buf
Enter string: 123456789012345
12345678
```

```
unix> ./buf
Enter string: 1234567890123456
*** stack smashing detected ***
```

# Stack Canaries: Protected Buffer Disassembly (buf)

This is extra (non-testable) material

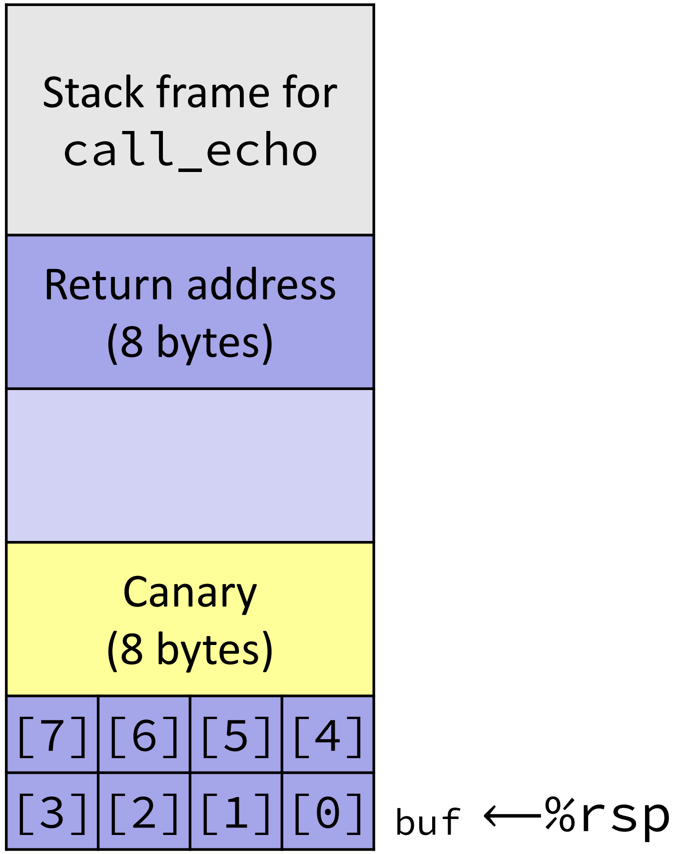
**echo:**

```
401156:  push   %rbx
401157:  sub    $0x10,%rsp
40115b:  mov    $0x28,%ebx
401160:  mov    %fs:(%rbx),%rax
401164:  mov    %rax,0x8(%rsp)
401169:  xor    %eax,%eax
 ...     ... call printf ...
40117d:  callq  401060 <gets@plt>
401182:  mov    %rsp,%rdi
401185:  callq  401030 <puts@plt>
40118a:  mov    0x8(%rsp),%rax
40118f:  xor    %fs:(%rbx),%rax
401193:  jne    40119b <echo+0x45>
401195:  add    $0x10,%rsp
401199:  pop    %rbx
40119a:  retq
40119b:  callq  401040 <__stack_chk_fail@plt>
```

# Setting Up Canary

This is extra (non-testable) material

*Before call to gets*




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

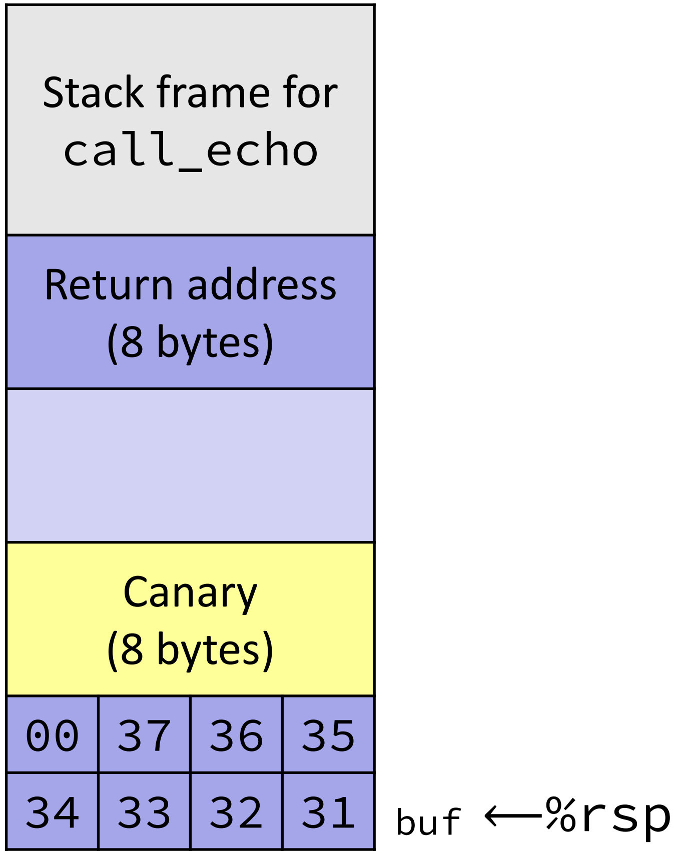
**Segment register stores canary** ***(don't worry about why)***

```
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*



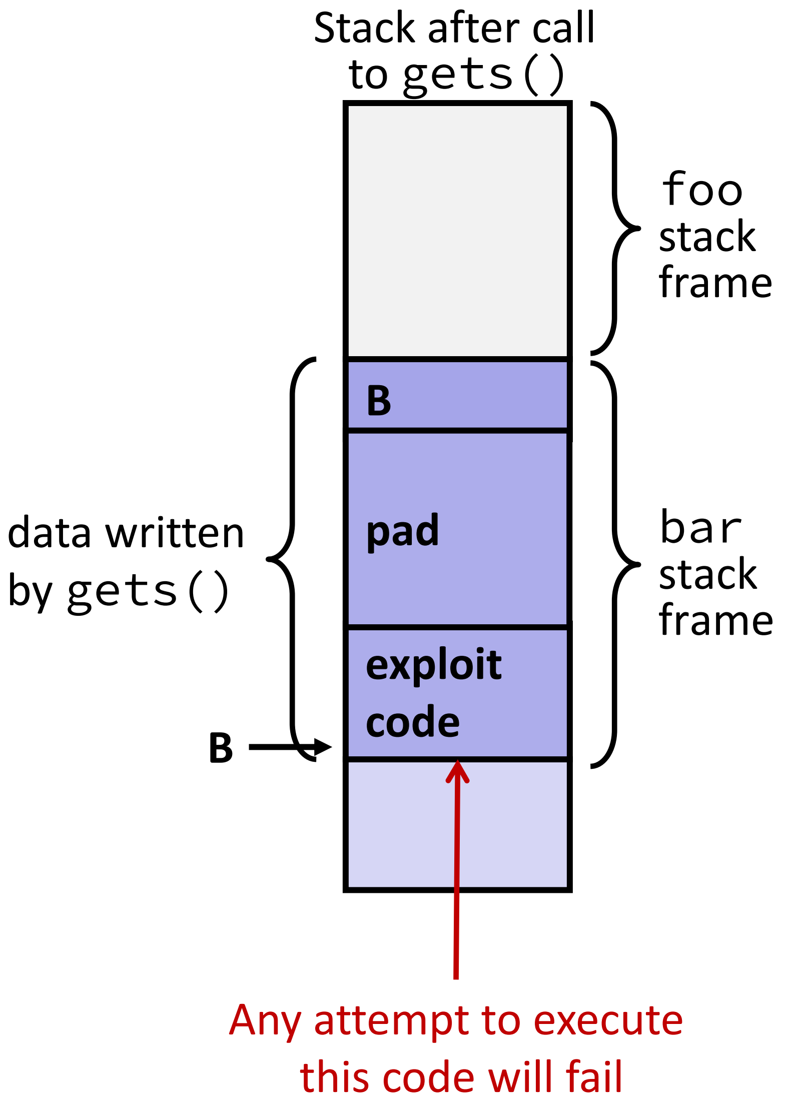

```
/* Echo Line */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
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) System-Level Protections: Non-executable Memory

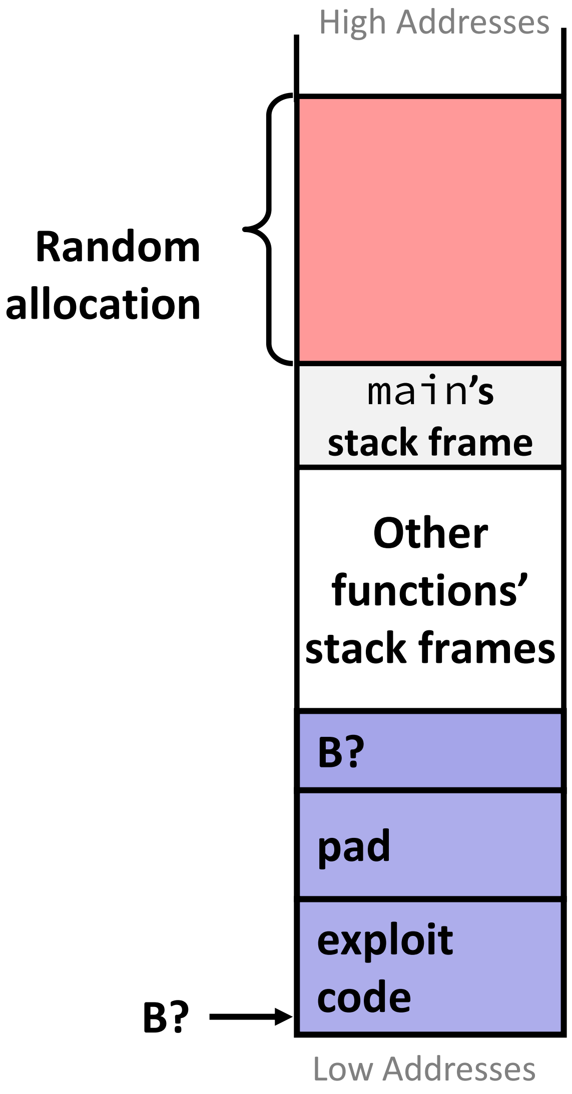
- x86-64 added explicit “execute” memory permission
  - **Stack**, Static Data, and Heap segments marked as non-executable so you cannot execute code from these regions
  - Hardware support needed to make this possible
- Unfortunately, not always available
  - *e.g.*, embedded devices like cars, smart homes
- Can’t stop more sophisticated attacks
  - *e.g.*, return-oriented programming, return to libc attack, JIT-spray attack



# 3) System-Level Protections: Randomized Stack Offsets

- At start of program, allocate a random amount of space on Stack
  - Repositions stack (and stack addresses) each time the program executes
  - Makes it difficult for hacker to predict location of injected code

- Example: Execute code on Slide 6 three times
  - Address of variable `local`:
    - `0x7ffd19d3f8ac`
    - `0x7ffe8a462c2c`
    - `0x7ffe927c905c`



# Lecture Outline (5/5)

- Buffer Overflow
- Vulnerable Code
- Code Injection Attacks
- Dealing with Buffer Overflow Attacks
- **Real Life Examples of Buffer Overflow**

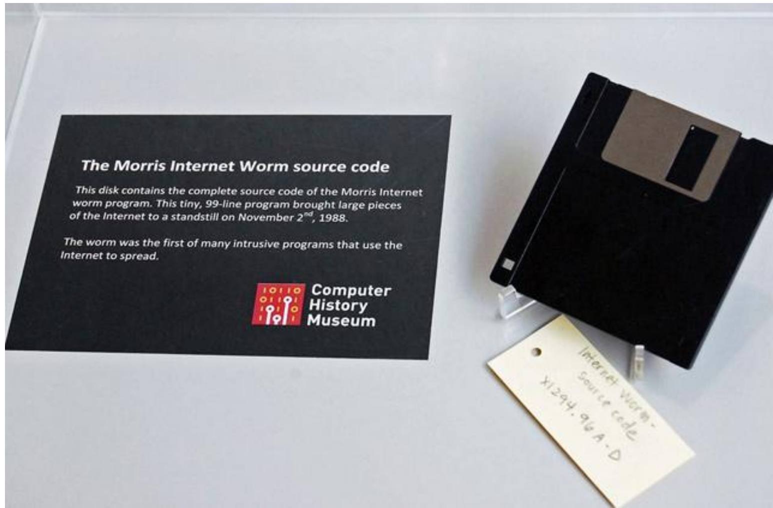
# Exploits Based on Buffer Overflows

- ⚠️ Buffer overflow bugs can allow attackers to execute arbitrary code on victim machines ☠️
  - Most commonly executing a “root shell” – terminal with elevated privileges

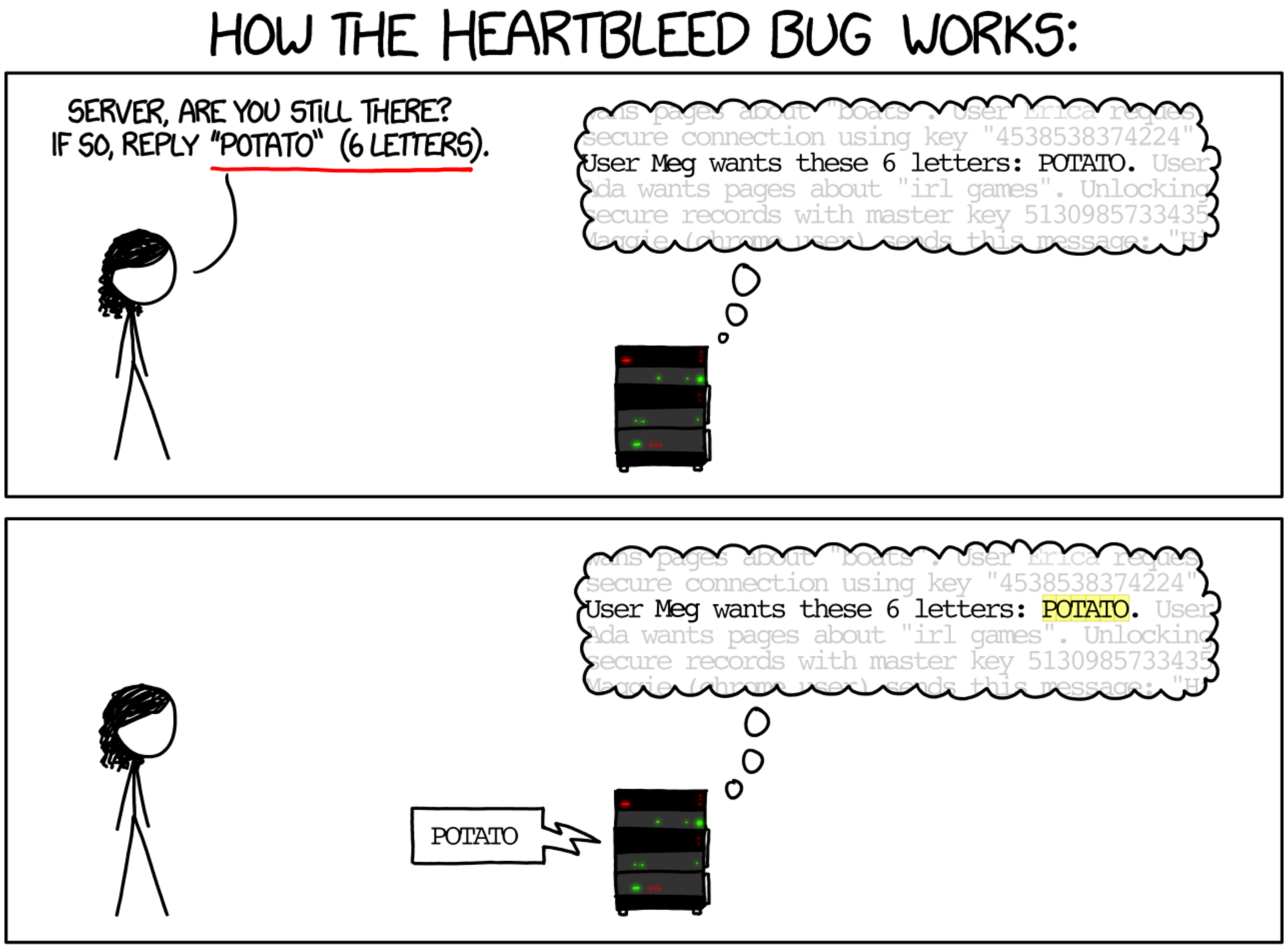
- Distressingly common in real programs
  - Original “Internet worm” (1988)
  - Heartbleed (2014) & Cloudbleed (2017)
  - Hacking embedded devices (*e.g.*, cars, smart home devices, Internet of Things)

# The Morris Worm (1988)

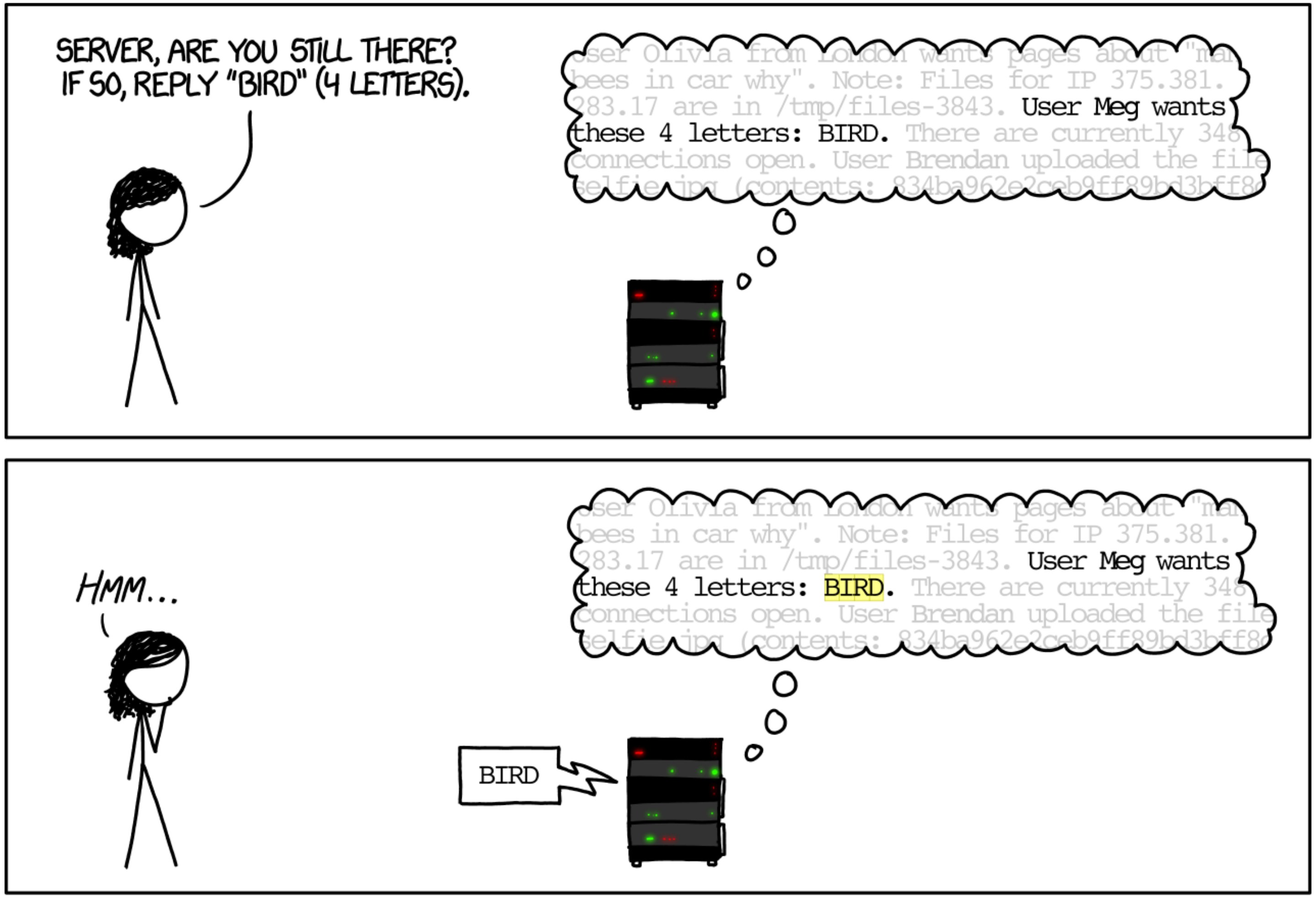
- Early versions of the finger server (`fingerd`) used `gets` to read the argument sent by the client
  - *e.g.*, `finger droh@cs.cmu.edu`
- The Morris Worm attacked `fingerd` server with phony argument:
  - `finger "`*`exploit-code padding new-return-addr`*`"`
  - Exploit code executed a root shell on the victim machine, then scanned for other machines to attack
- Fallout/legacy (1989 article)
  - Invaded ~6000 computers in hours (10% of the Internet)
  - The author, Robert Morris, was prosecuted
    - First conviction under 1986 Computer Fraud and Abuse Act
    - Now an MIT professor



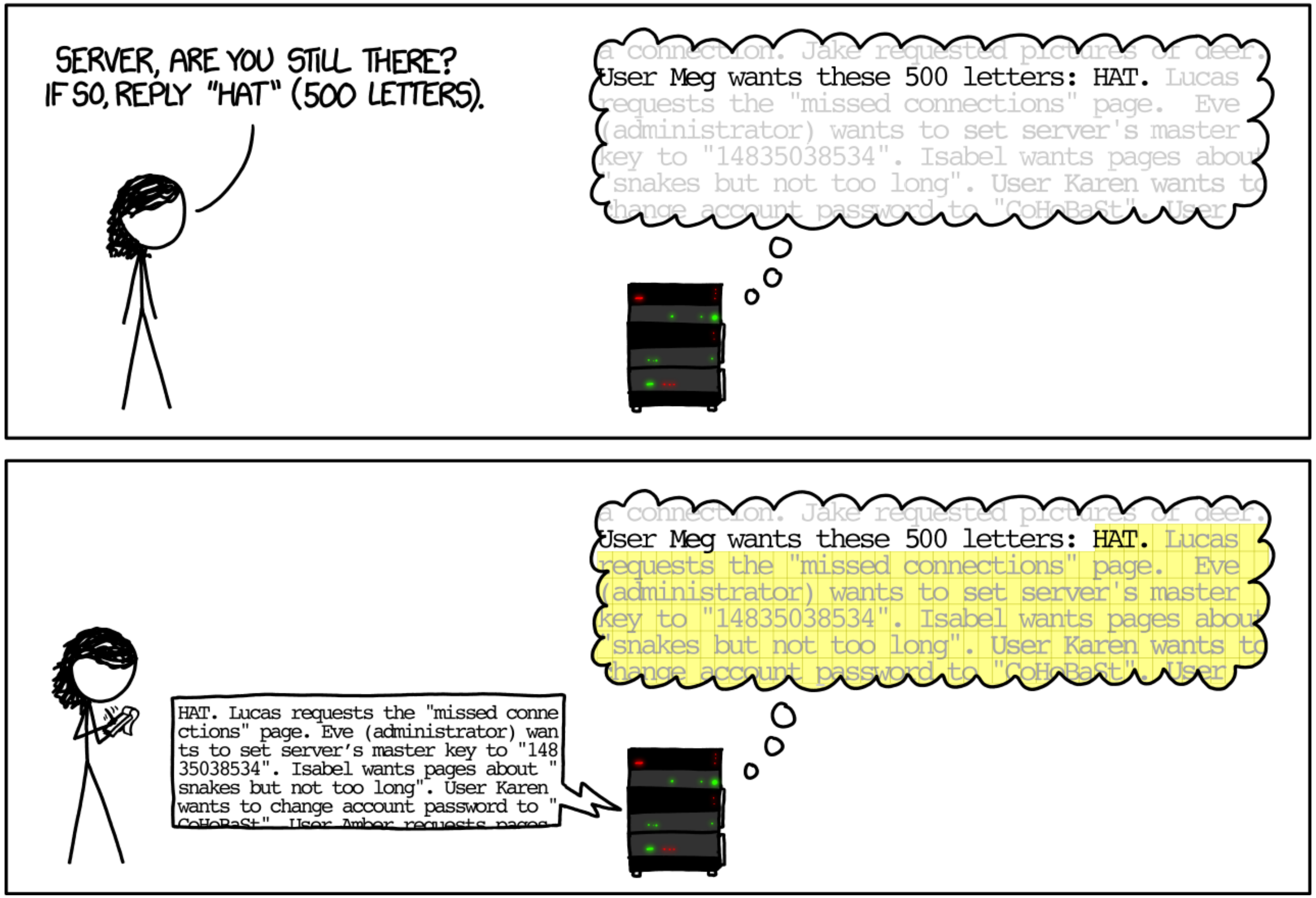
# Heartbleed (2014 – 1/3)



# Heartbleed (2014 – 2/3)

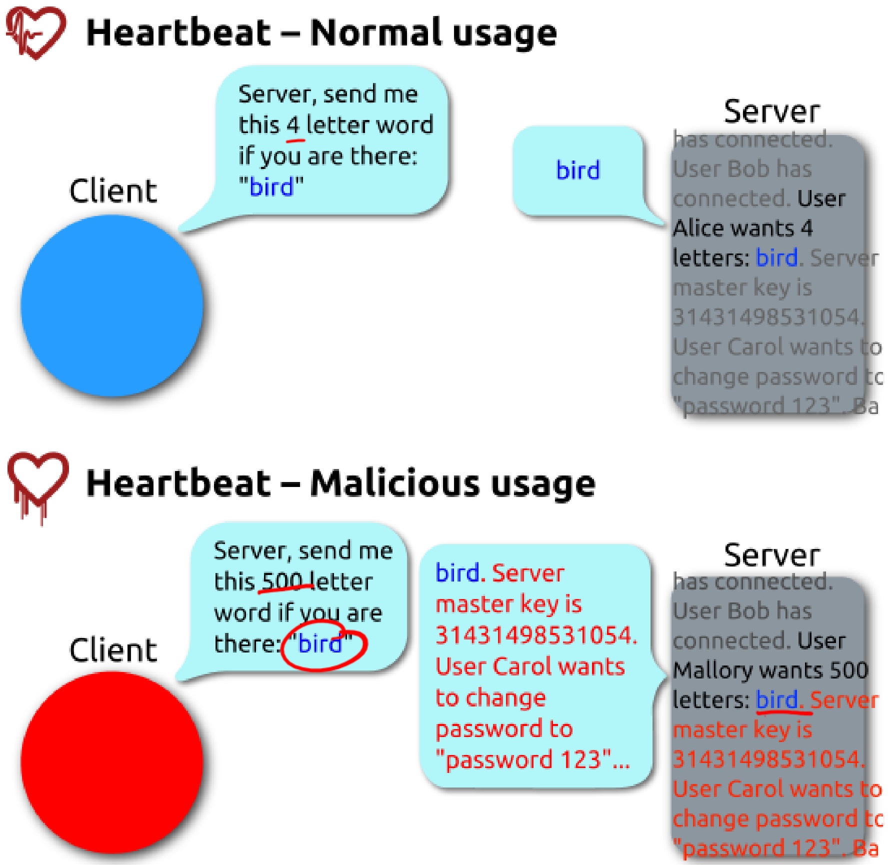


# Heartbleed (2014 – 3/3)



# Heartbleed Details

- Buffer over-read in OpenSSL
  - Open source security library
  - Bug in a small range of versions
- "Heartbeat" packet: message & length
  - Server echoes back data to match length
  - Allowed attackers to read contents of memory
- ~17% of Internet affected
  - *e.g.*, Github, Yahoo, Stack Overflow, Amazon Web Services

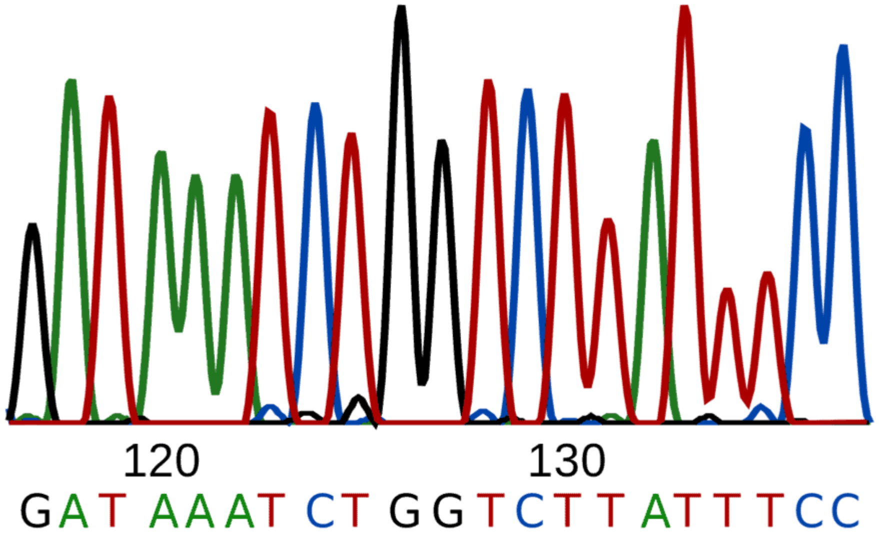


# Hacking Cars (2010)

- 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

# Hacking DNA Sequencing Tech (2017)

- UW CSE project: Computer Security and Privacy in DNA Sequencing
  Ney et al. (2017): https://dnasec.cs.washington.edu/
  - Potential for malicious code to be encoded in DNA!
  - Attacker can gain control of DNA sequencing machine when malicious DNA is read




x86 in DNA!

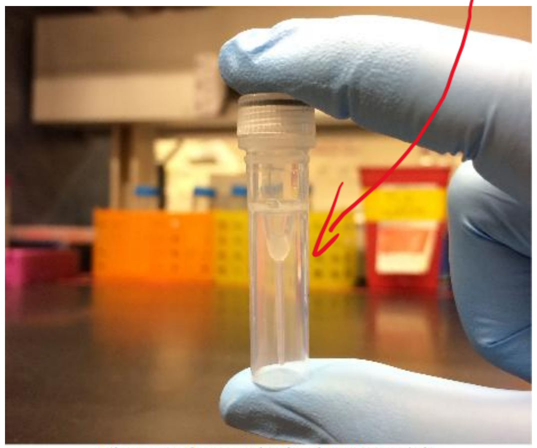


Figure 1: Our synthesized DNA exploit

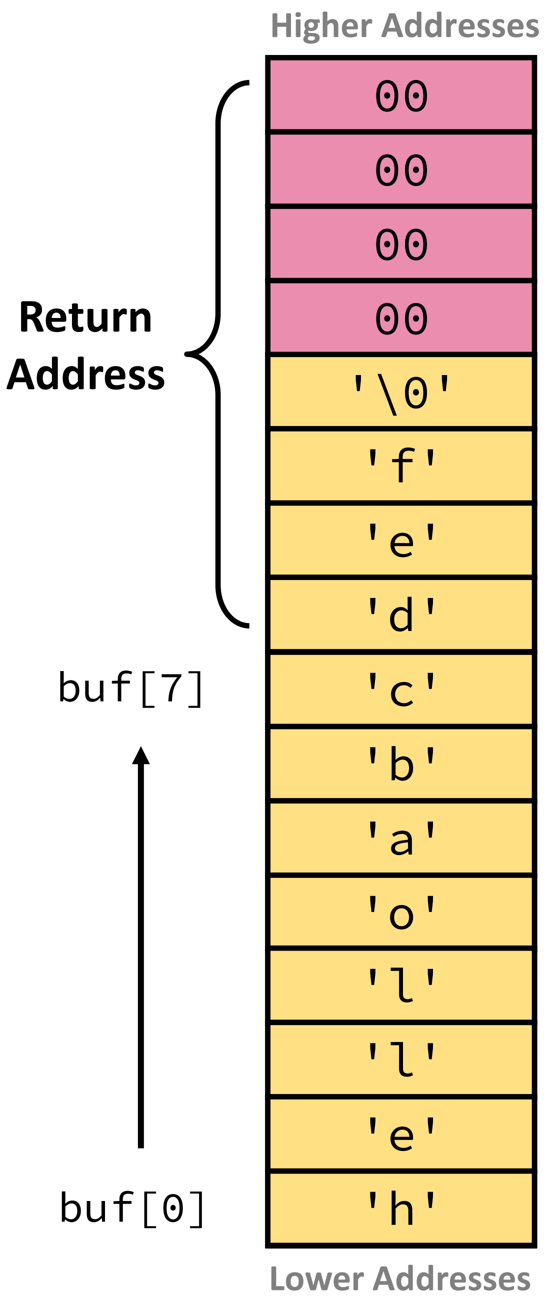
# Think this is cool?

- 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

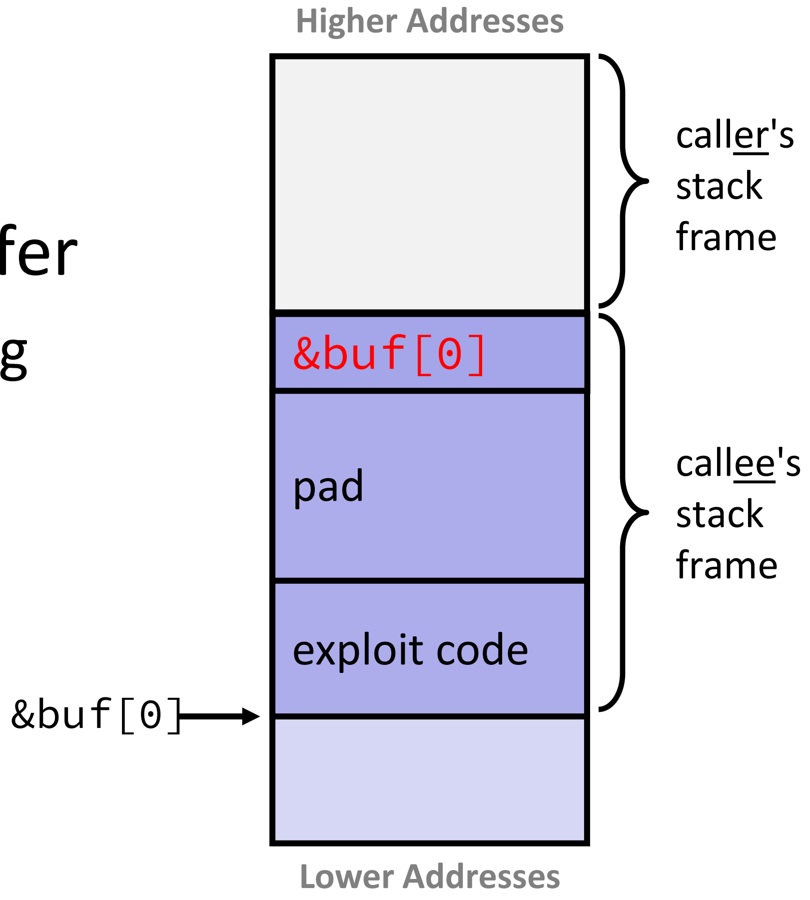
# Summary (1/3)

- A **buffer** is an array that holds temporary data (*e.g.*, user/file/network input)
- **Buffer overflow** is writing past the end of the buffer
  - Common in C/Unix/Linux due to lack of bounds checking
  - Vulnerable functions include `gets`, `strcpy`, `scanf`, `fscanf`, `sscanf`
- Buffer overflow exploit: **stack smashing**
  - Overflow local array to alter stack contents
  - Commonly used to alter procedure return address



# Summary (2/3)

- A **buffer** is an array that holds temporary data (*e.g.*, user/file/network input)
- **Buffer overflow** is writing past the end of the buffer
  - Common in C/Unix/Linux due to lack of bounds checking
  - Vulnerable functions include `gets`, `strcpy`, `scanf`, `fscanf`, `sscanf`
- Buffer overflow exploit: **code injection**
  1) Put exploit/machine code in buffer
  2) Pad to reach stack frame's return address
  3) Replace return address with address of the buffer



# Summary (3/3)

- Dealing with buffer overflow attacks
  - Use array bounds checking
    - Manually (*i.e.*, implement yourself) or automatically (*e.g.*, use safe functions or non-C language)
  - Add a stack canary after the buffer
    - Secret value (changes on each execution) that shouldn't change
  - Randomized stack offsets
    - Makes finding the address of exploit code more difficult
  - Non-executable memory regions (*e.g.*, the stack)
    - Prevent exploit code from being placed and executed there