# Buffer Overflows

CSE 351 Spring 2024



## Instructor:

Elba Garza

## Teaching Assistants:

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson


Playlist: CSE 351 24Sp Lecture Tunes!

# Relevant Course Information

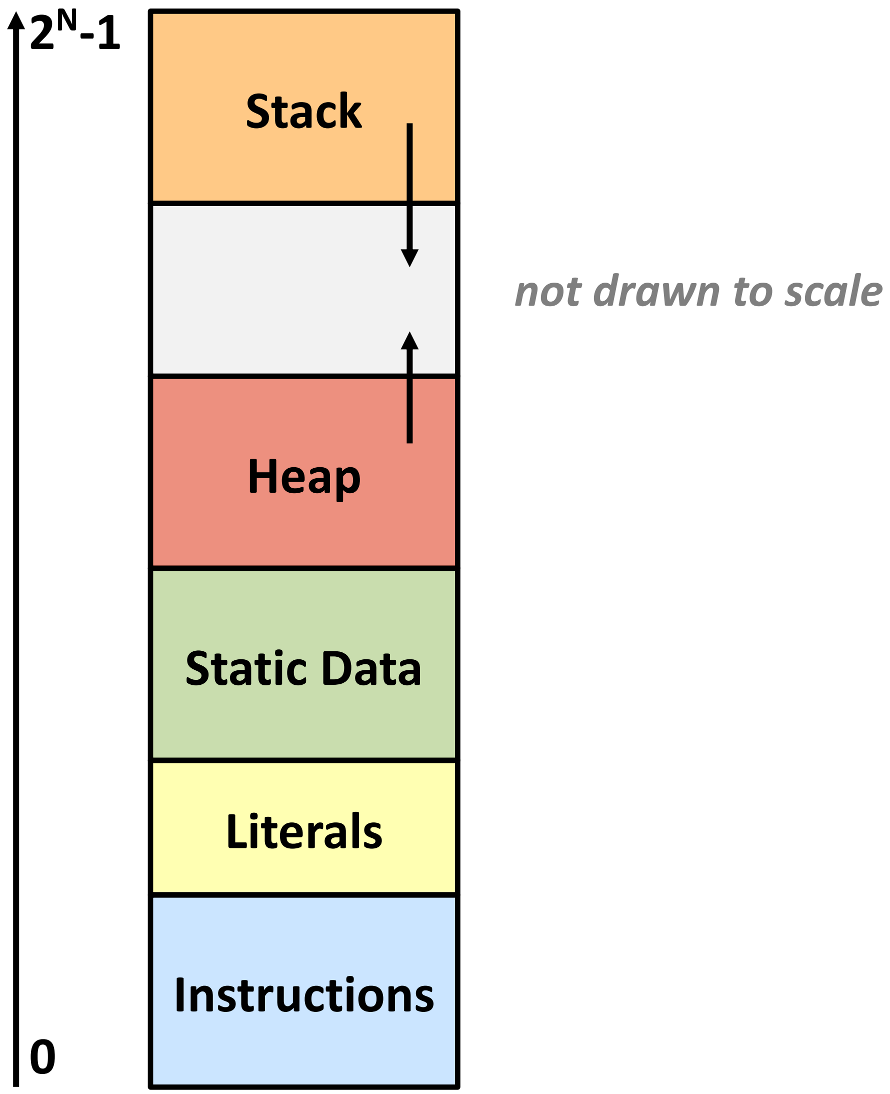
- Lab 2 due tonight & Lab 3 releasing today, due May 8th
  - You will have everything you need for it by the end of this lecture!
- HW13/14 due May 1st, HW15 due May 3rd
- Mid-Quarter Assessment results & write-up coming soon!
- Canvas Mid-Quarter Survey releasing on May 1st
  - Part of EPA grade
  - Particularly focusing on TA feedback
  - Due May 6th

# Buffer Overflows

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

# 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



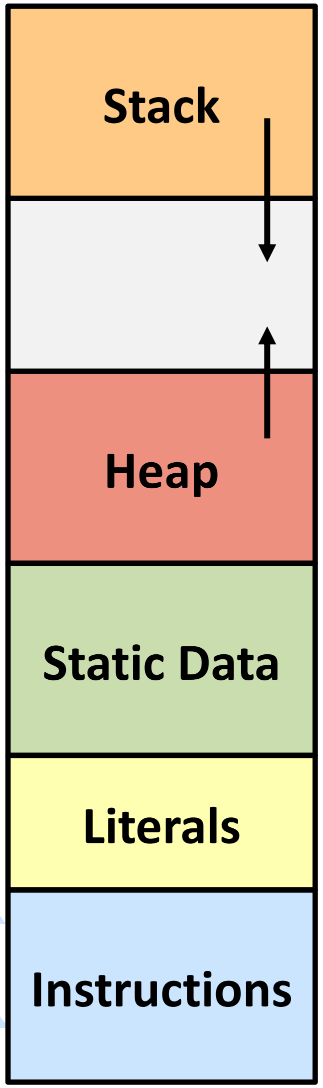
# Memory Allocation Example

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




*not drawn to scale!*

Where does everything go?

# What Is a Buffer?

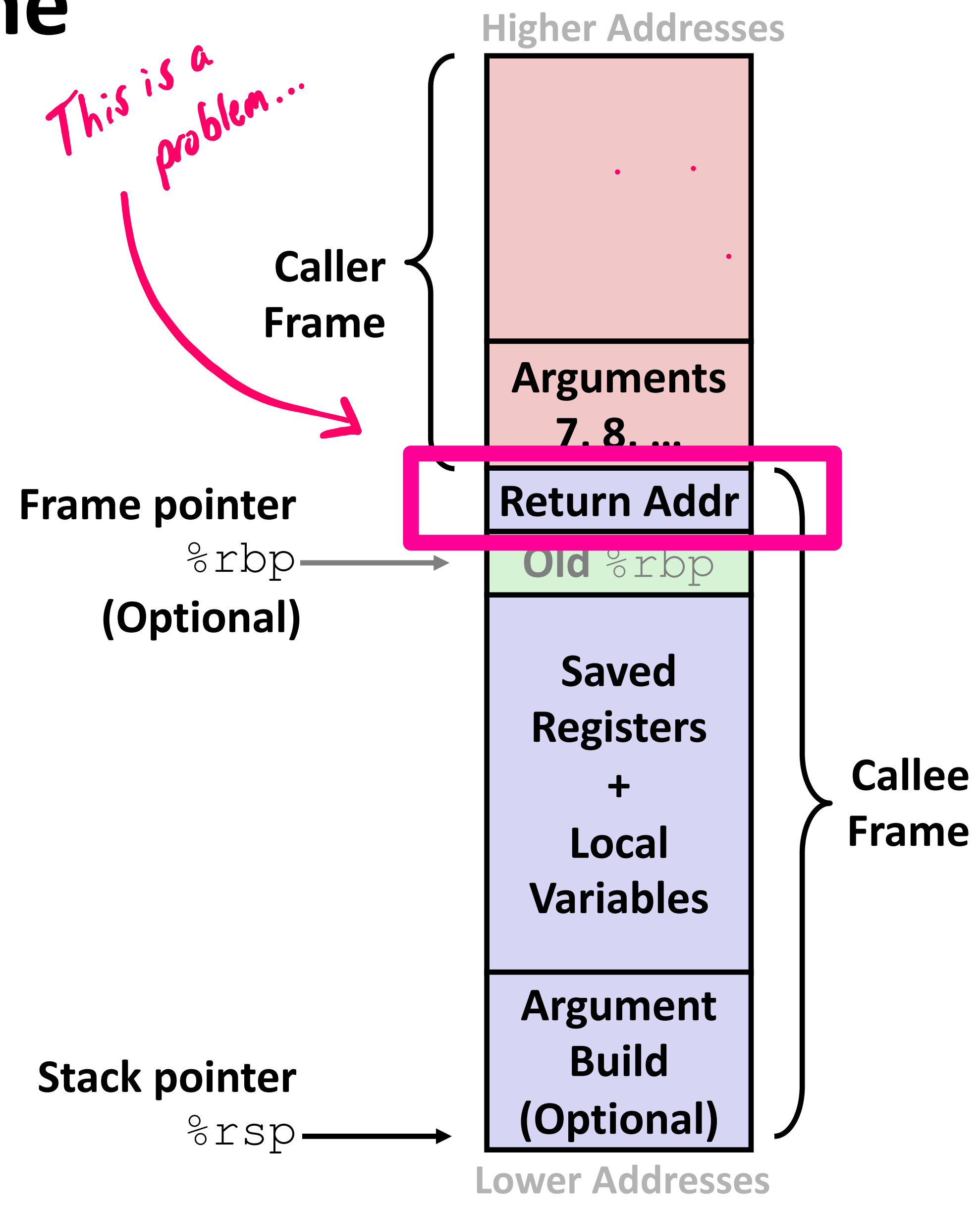
- A **buffer** is an array used to temporarily store data
- You've probably seen "video buffering…"
  - The video is being written into a buffer before being played



- Buffers can also be used to 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)
  - Caller-saved registers pushed before setting up arguments for a function call
  - Callee-saved registers pushed before using long-term registers
  - Local variables, if can't be kept in registers
  - "Argument build" area—Need to call a function with >6 arguments? Put them here!



# Buffer Overflow in a Nutshell

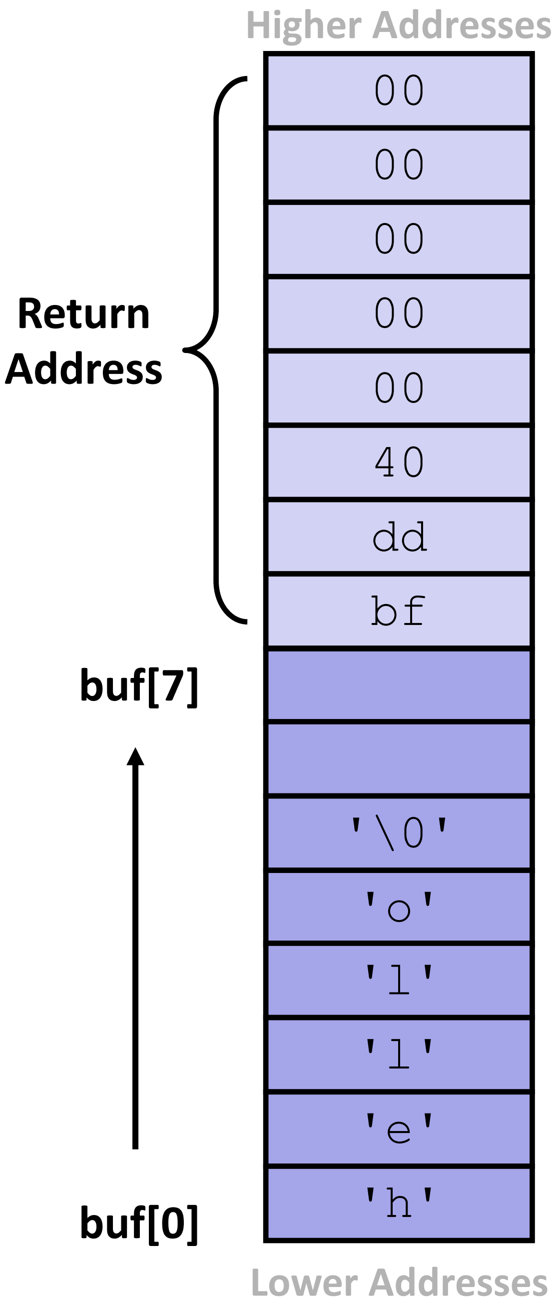
- C does not check array bounds (as we so well know by now...)
  - Many Unix/Linux/C functions don't check argument sizes
  - Allows overflowing—or, writing past the end—of buffers/arrays

- "Buffer Overflow" = Writing past the end of an array, intentionally or unintentionally...

- **Key Observation**: Characteristics of the traditional Linux memory layout [provide opportunities] for malicious actions
  - Stack grows "backwards" in memory
  - Data and instructions both stored in the same memory!

# Buffer Overflow in a Nutshell

- 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!



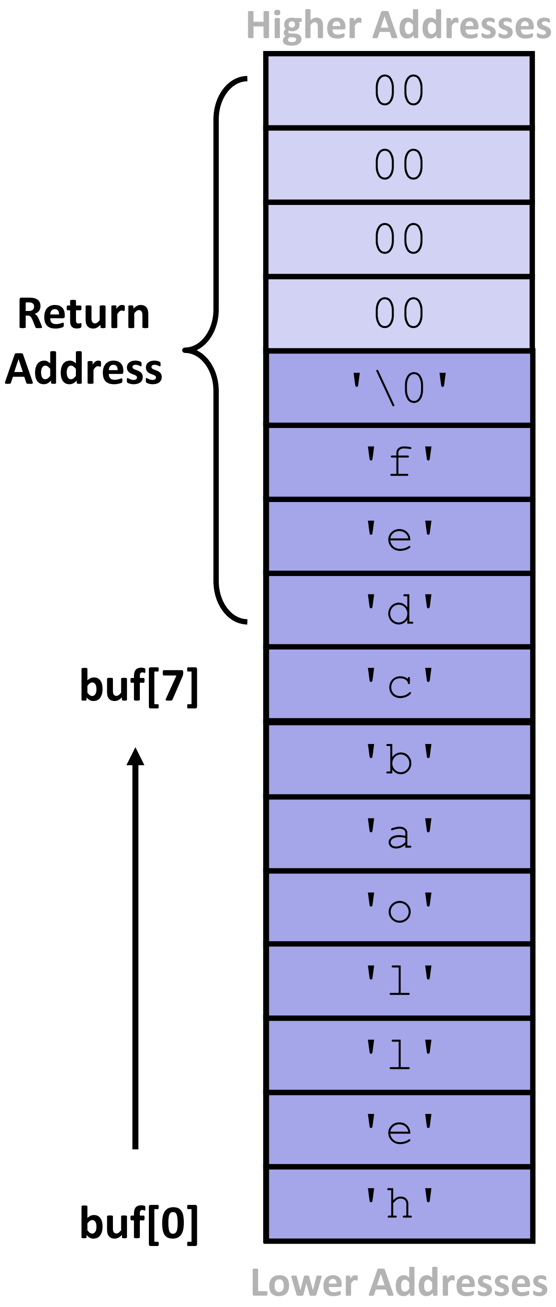

```
Enter input: hello
```

No overflow ☺

# Buffer Overflow in a Nutshell

- 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
```

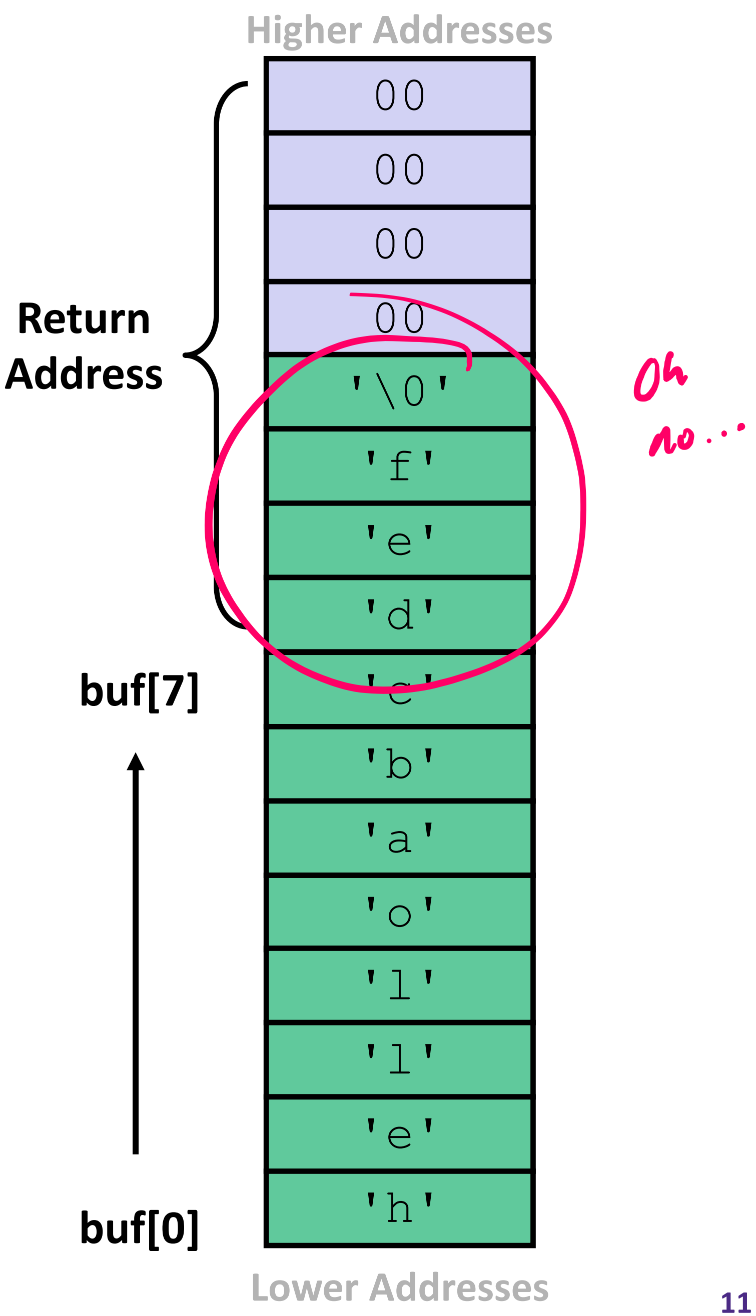


# Buffer Overflow in a Nutshell

- 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! ☹**



# Buffer Overflow in a Nutshell

- 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
  - Specifically, 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!
    - e.g. Heartbleed, cloudbleed, etc.
    - #1 overall cause is social engineering/user ignorance 😬

So let's see how systems let us do this…

# String Library Code

- Actual source code implementation of Unix function `gets()`:

```
/* Get string from stdin
one character at a time */
char* gets(char* dest) {
    int c = getchar(); // read 1 byte
    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

- Actual source code implementation of Unix function `gets()`:

```
/* Get string from stdin
one character at a time */
char* gets(char* dest) {
    int c = getchar(); // read 1 byte
    char* p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

Similar problems with other Unix functions:

- `strcpy`: Copies string of arbitrary length to a dst
- `scanf`, `fscanf`, `sscanf`, when given `%s` specifier

🚨 No way to specify **limit** on number of characters to read! 🚨

The `man` page for `gets(3)` now says "BUGS: Never use `gets()`."

# An Example: Vulnerable Buffer Code

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

```
/* Echo Line */
void echo() {
    char buf[8];  /* Way too small! */
    gets(buf);    /* Read input into buf */
    puts(buf);    /* Print output from buf */
}
```

gets

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

puts

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

# Buffer Overflow Disassembly (buf-nsp)

## 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 to place on stack

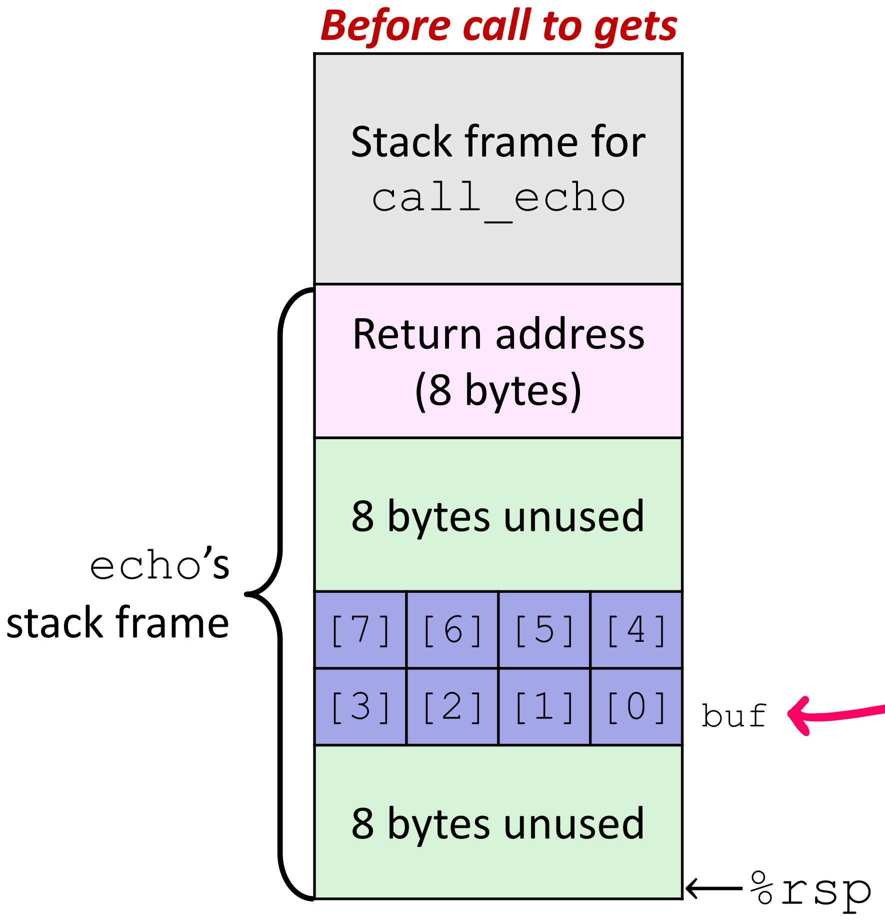
## echo:

hex!

```
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
```

- Allocate 24 bytes in stack (compiler's choice)
- Calculate address location to be passed to `gets`
- Calculate address location to be passed to `puts`
- Clean up stack & return

# Buffer Overflow Stack




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

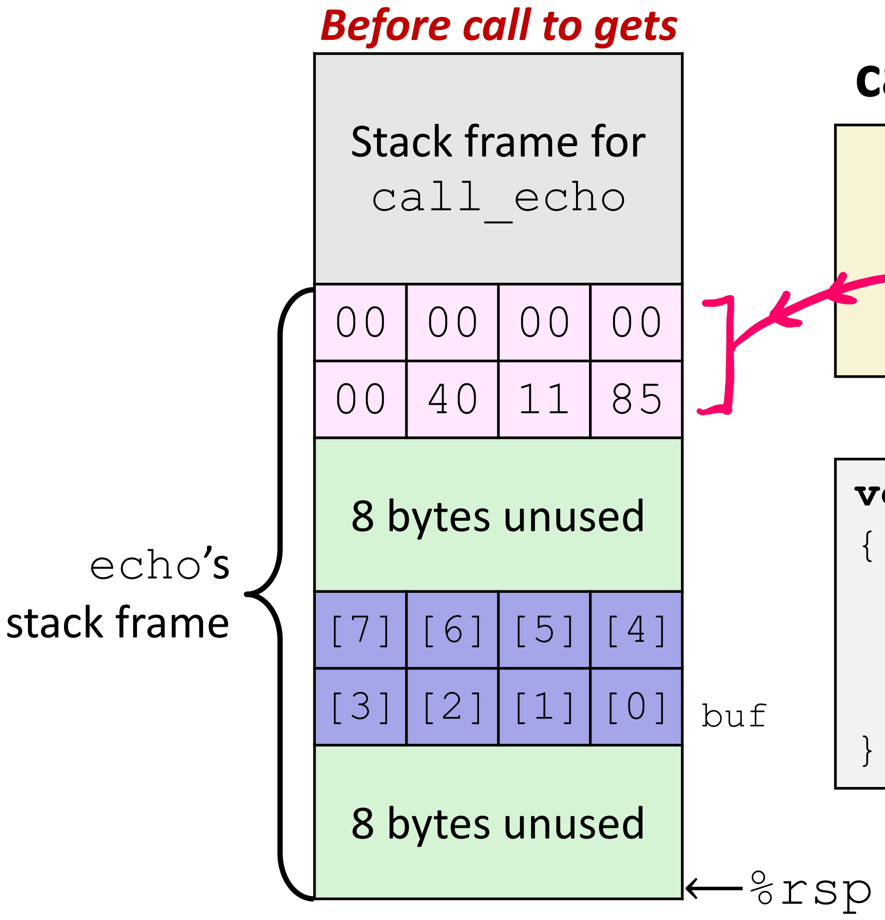
```
echo:
  subq   $24, %rsp
   ...
  leaq   8(%rsp), %rdi
  mov    $0x0,%eax
  call   gets
   ...
```

in base 10

// start buf at 8 bytes from where %rsp points.

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

# Buffer Overflow Setup




## call_echo:

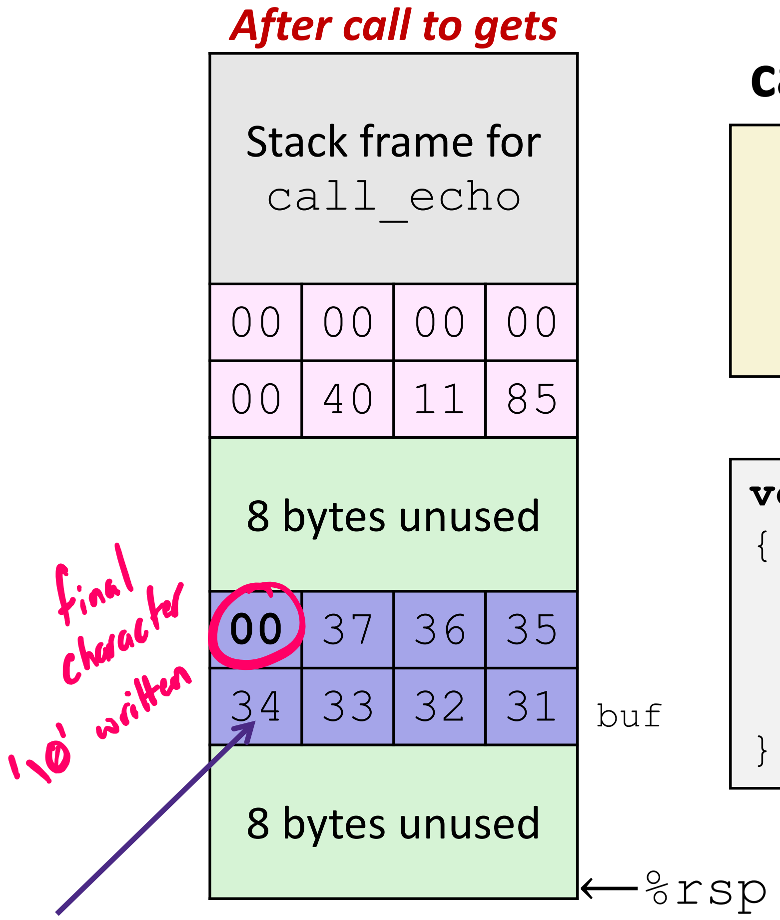
```
. . .
401180:  callq  401146 <echo>
401185:  add    $0x8,%rsp
. . .
```

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

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

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

# Buffer Overflow Example #1: 1234567




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

**call_echo:**

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

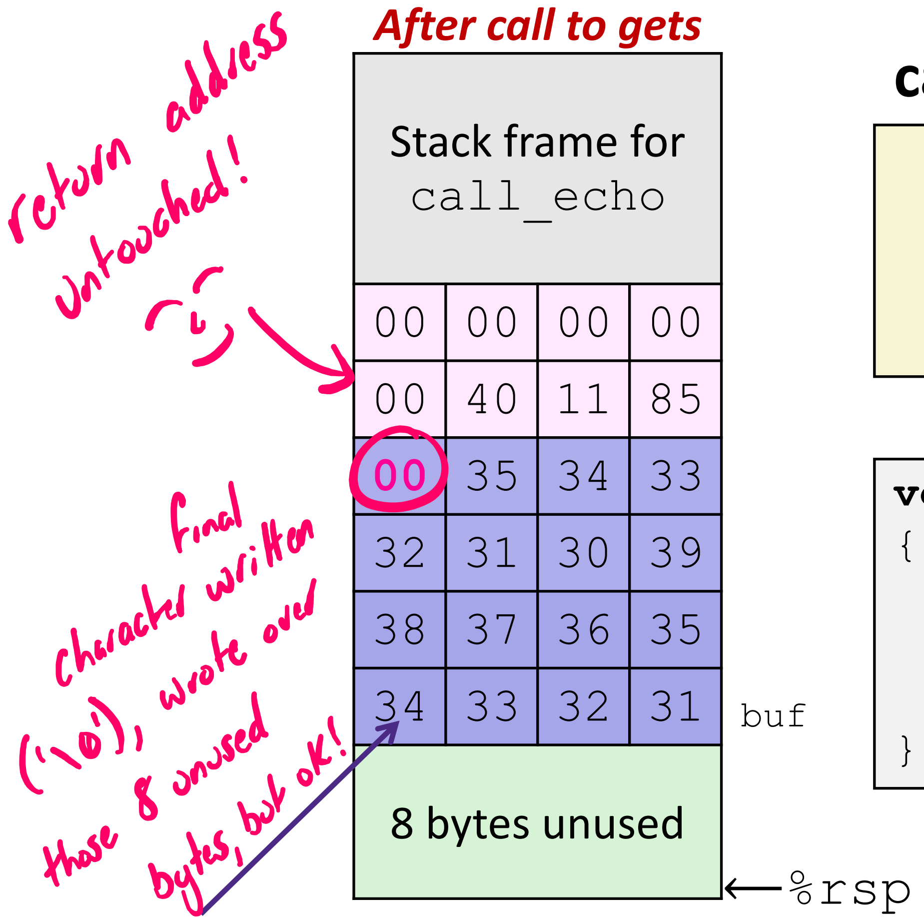
```
void echo()
{
    char buf[8];
    gets(buf);
    . . .
}
```

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

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

**All good! No overflow. Phew!**

# Buffer Overflow Example #2: 123456789012345




**call_echo:**

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

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

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

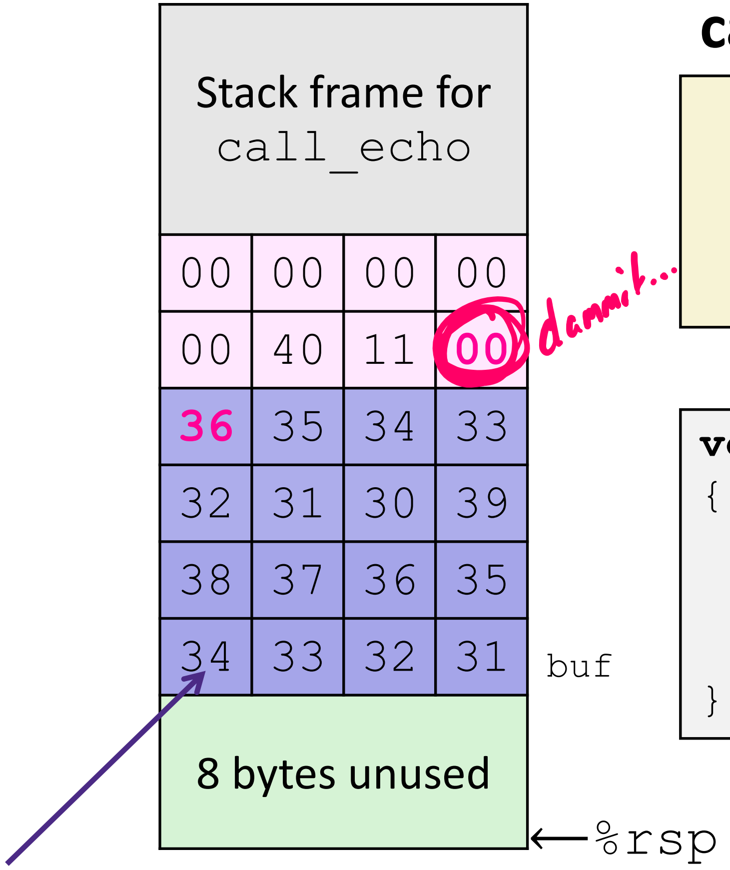
```
unix> ./buf-nsp
Enter string: 123456789012345
123456789012345
```

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

**Overflowed buffer, but did <u>not</u> corrupt state.**

# Buffer Overflow Example #3: 1234567890123456

*After call to gets*




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

## call_echo:

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

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

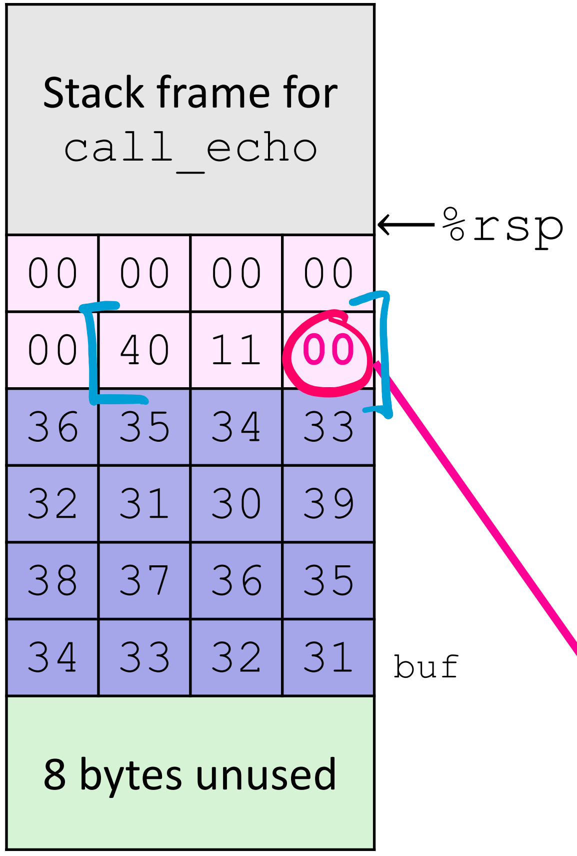
```
echo:
  subq  $24, %rsp
   ...
  leaq  8(%rsp), %rdi
  mov   $0x0,%eax
  call  gets
   ...
```

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

**Overflowed buffer and corrupted return pointer!**

# Buffer Overflow Example #3 Explained

***After return from echo***




```
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`

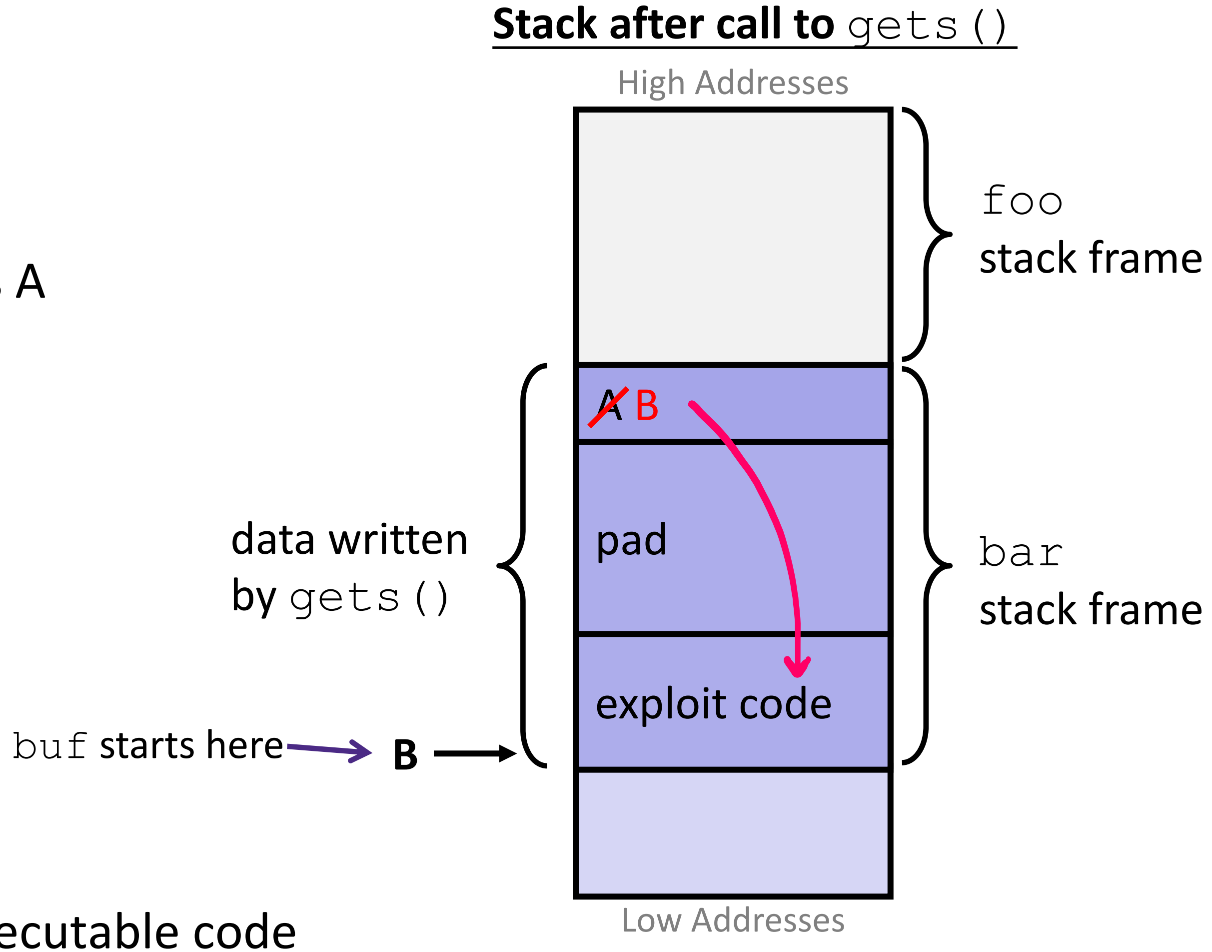
# Attack Time

# Malicious Use of Buffer Overflow: Code Injection Attacks

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

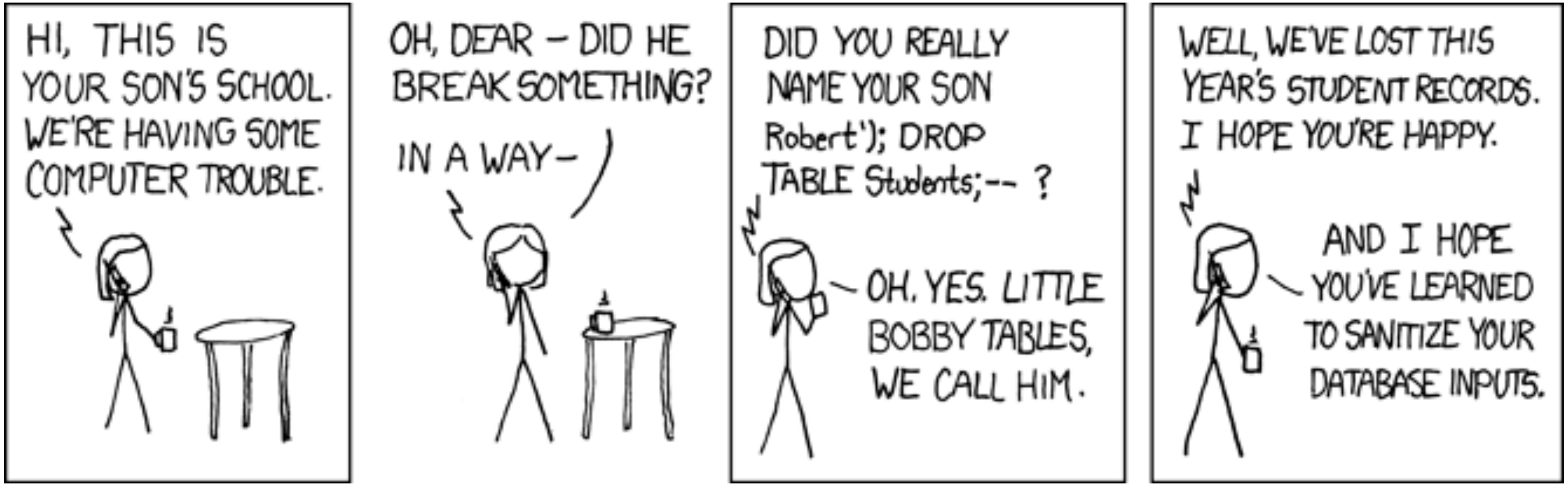
return address A

```
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**

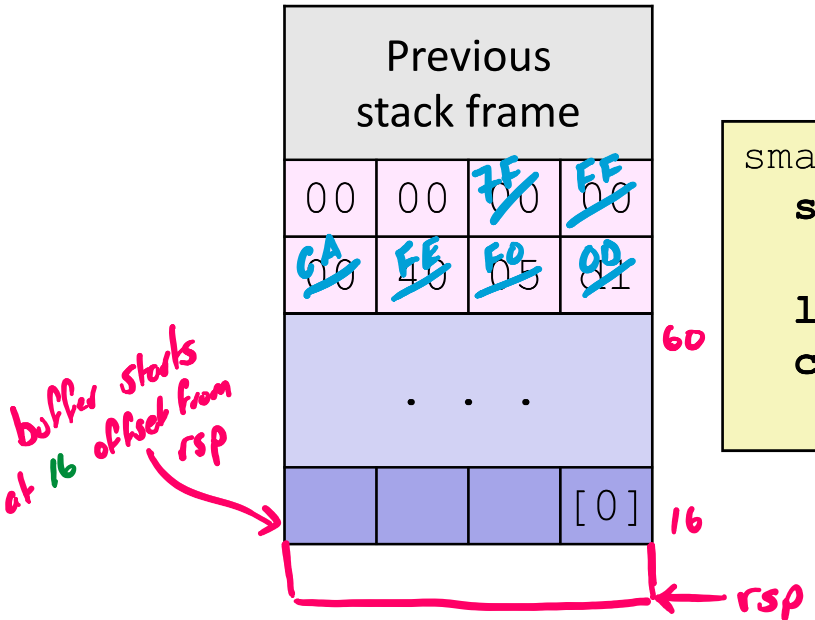
# Don't Execute Inputs, y'all.




https://xkcd.com/327

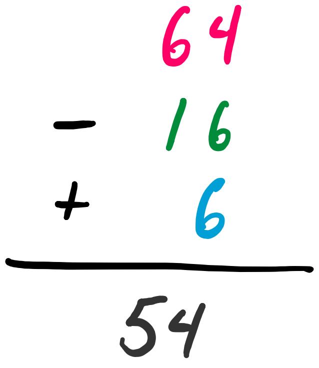
# Practice Question

- `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: (`0x00 00 7f ff ca fe f0 0d`)




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

A. 27
B. 30
C. 51
D. 54
E. We're lost...



# 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 (yikes)

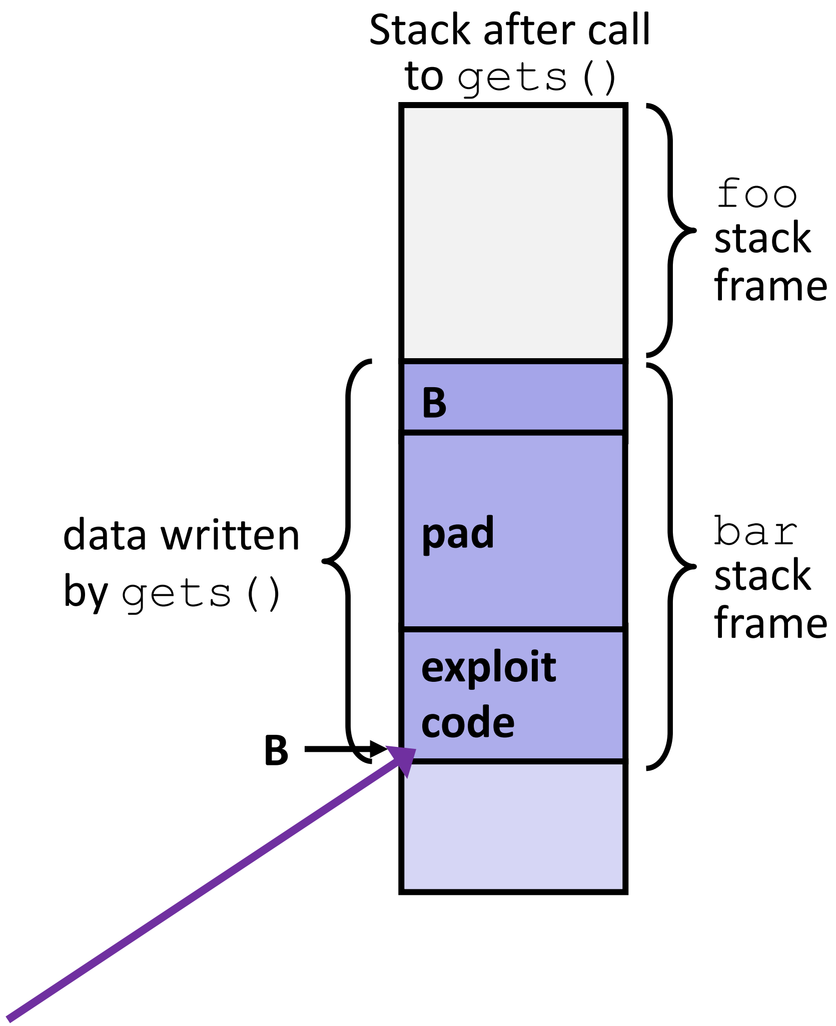
# Dealing with buffer overflow attacks

1) Employ system-level protections
2) Have compiler use “stack canaries”
3) Avoid overflow vulnerabilities in the first place…

# 1) 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




**Any attempt to execute this code will fail**

# 1) System-Level Protections

**Non-executable code segments:** Wait, doesn't this fix everything?

- Works well, but can't always use it
- Many embedded devices <u>**do not**</u> have this protection
  - *e.g.*, cars, smart homes, pacemakers
- Some exploits still work!
  - Return-oriented programming
  - Return to libc attack
  - JIT-spray attack

# 1) System-Level Protections

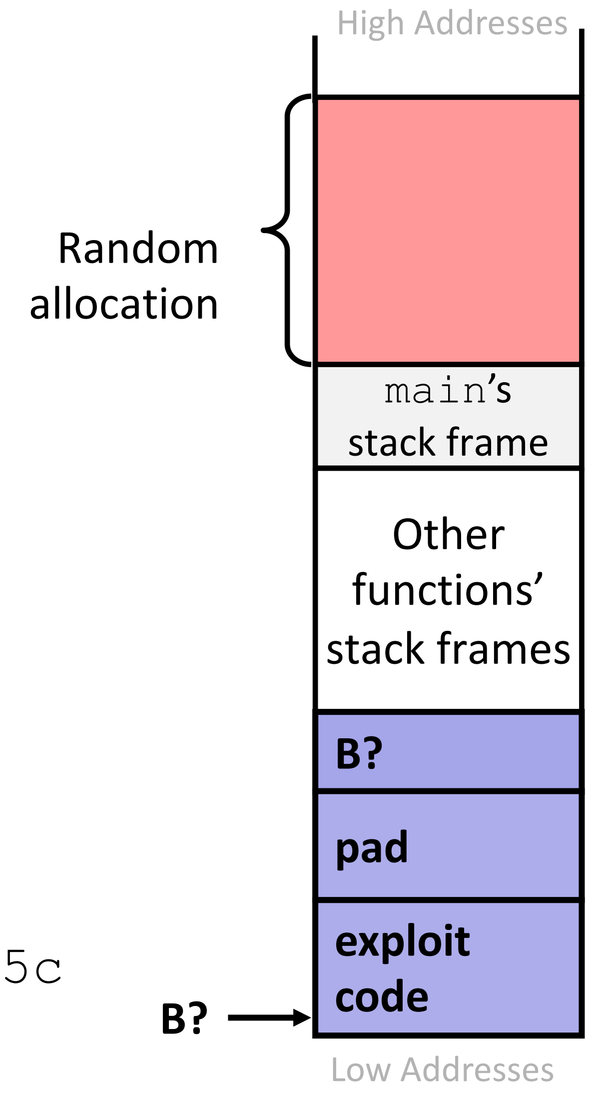
## Randomized stack offsets

- 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: Address of variable `local` for when Slide 5 code executed 3 times:

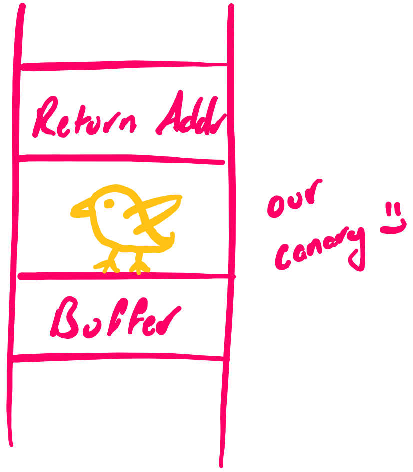
  - `0x7ffd19d3f8ac, 0x7ffe8a462c2c, 0x7ffe927c905c`

- **Stack repositioned each time program executes**
  - Not infallible, sadly: re-run attack til it works, use lots of nops, etc.



# 2) Stack Canaries

- Basic Idea: place a special value ("canary") on stack just beyond buffer
  - **Secret** value that is randomized before `main`
  - Placed between 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 ***
```

The overflow example code in RD15 had a canary in place!

# Protected Buffer Disassembly (buf)

This is extra (non-testable) material

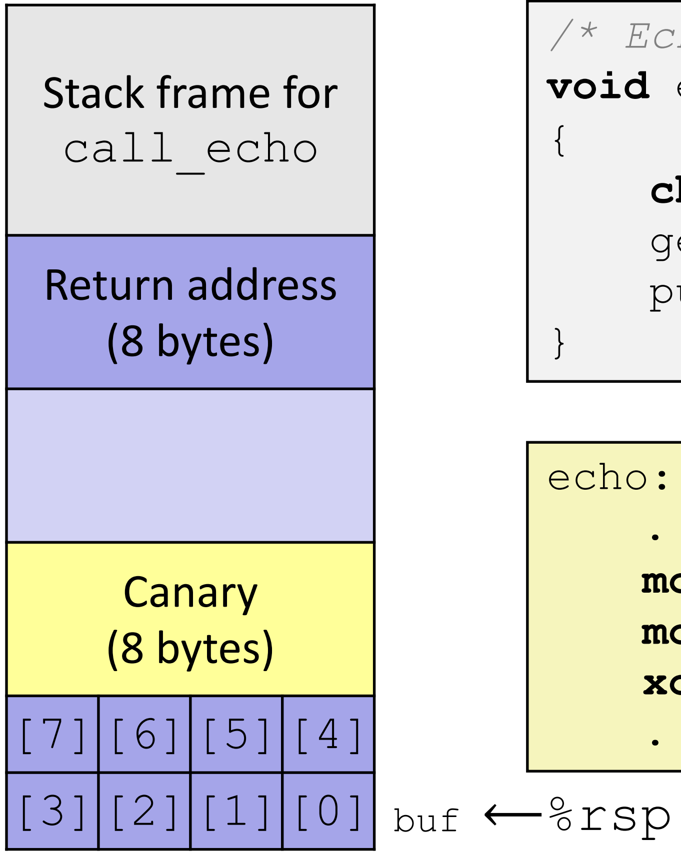
**echo:**

```
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);
}
```

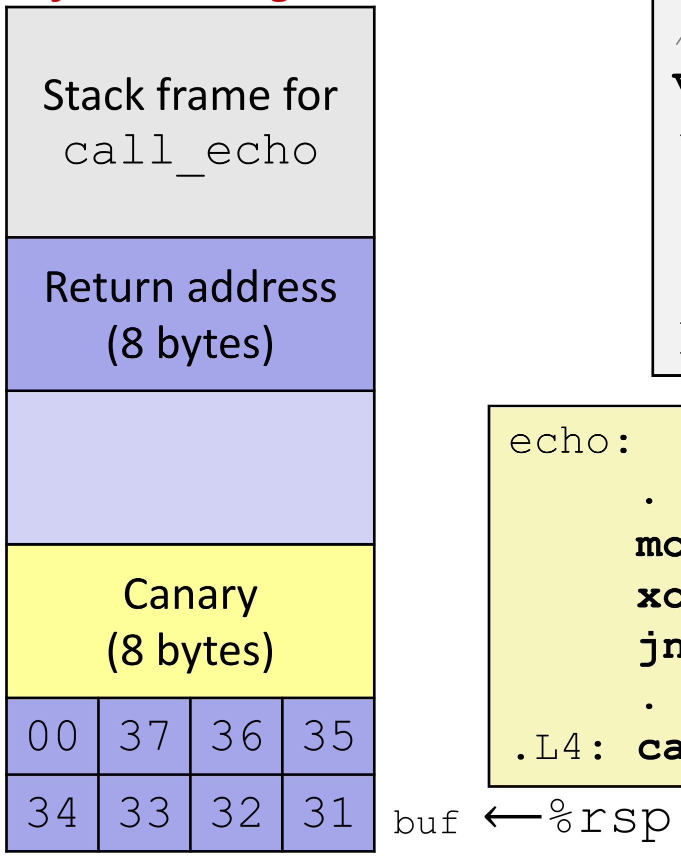
**Segment register**
***(don't worry about it)***

```
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

```
/* 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

- Alternatively, don't use C—use a language that does array index bounds check
  - Buffer overflow is impossible in Java
    - `ArrayIndexOutOfBoundsException`
- What if I need a "low-level" systems language?
  - Rust was designed with this in mind; Joe Biden is definitely a Rustacean 🦀
  - Golang has protection against this attack as well
- But sometimes you still need to manually manipulate memory…
  - Programming microprocessors or embedded systems – "poke" memory to perform I/O

# 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

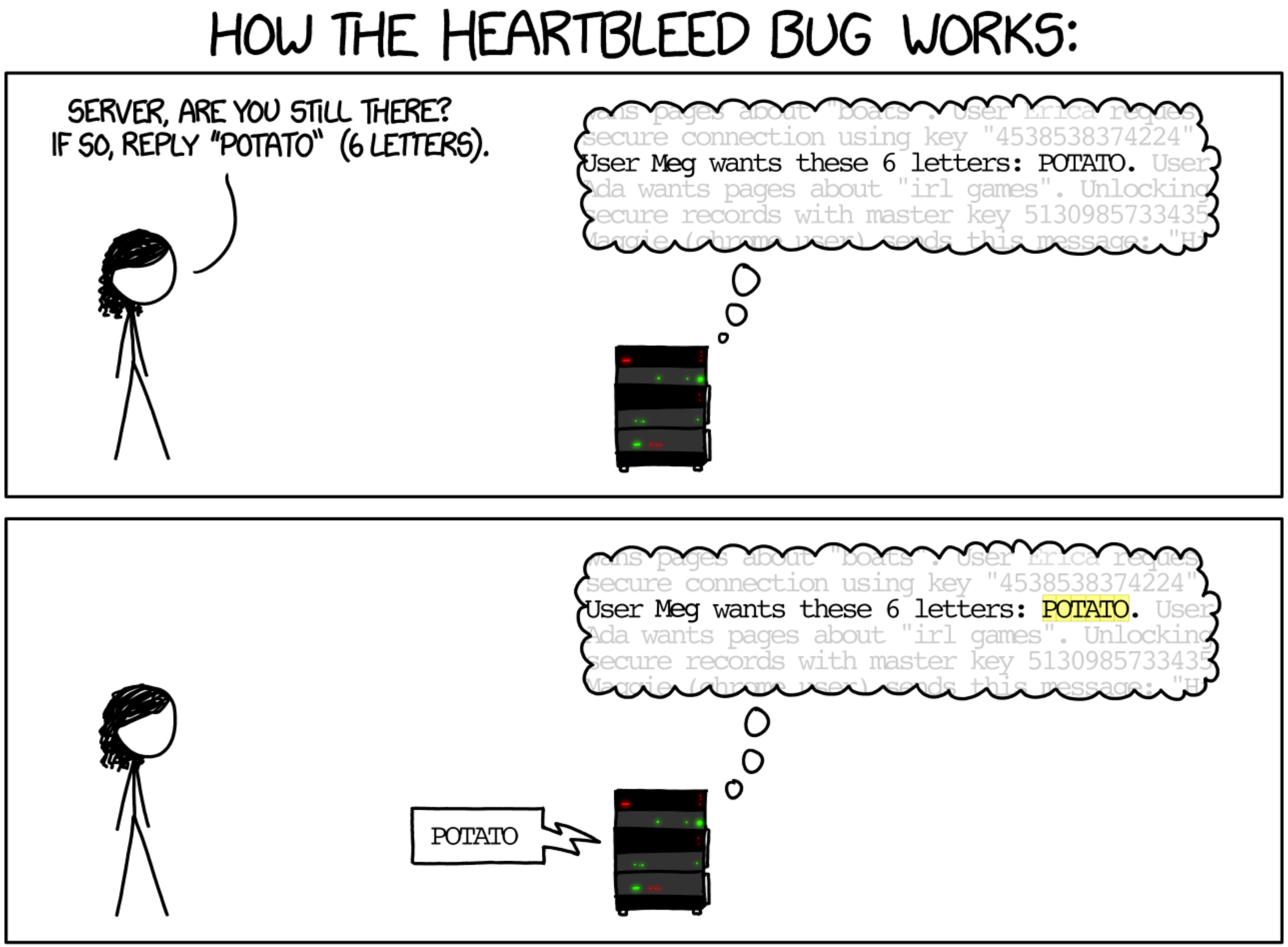
# Think this is cool?

- 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

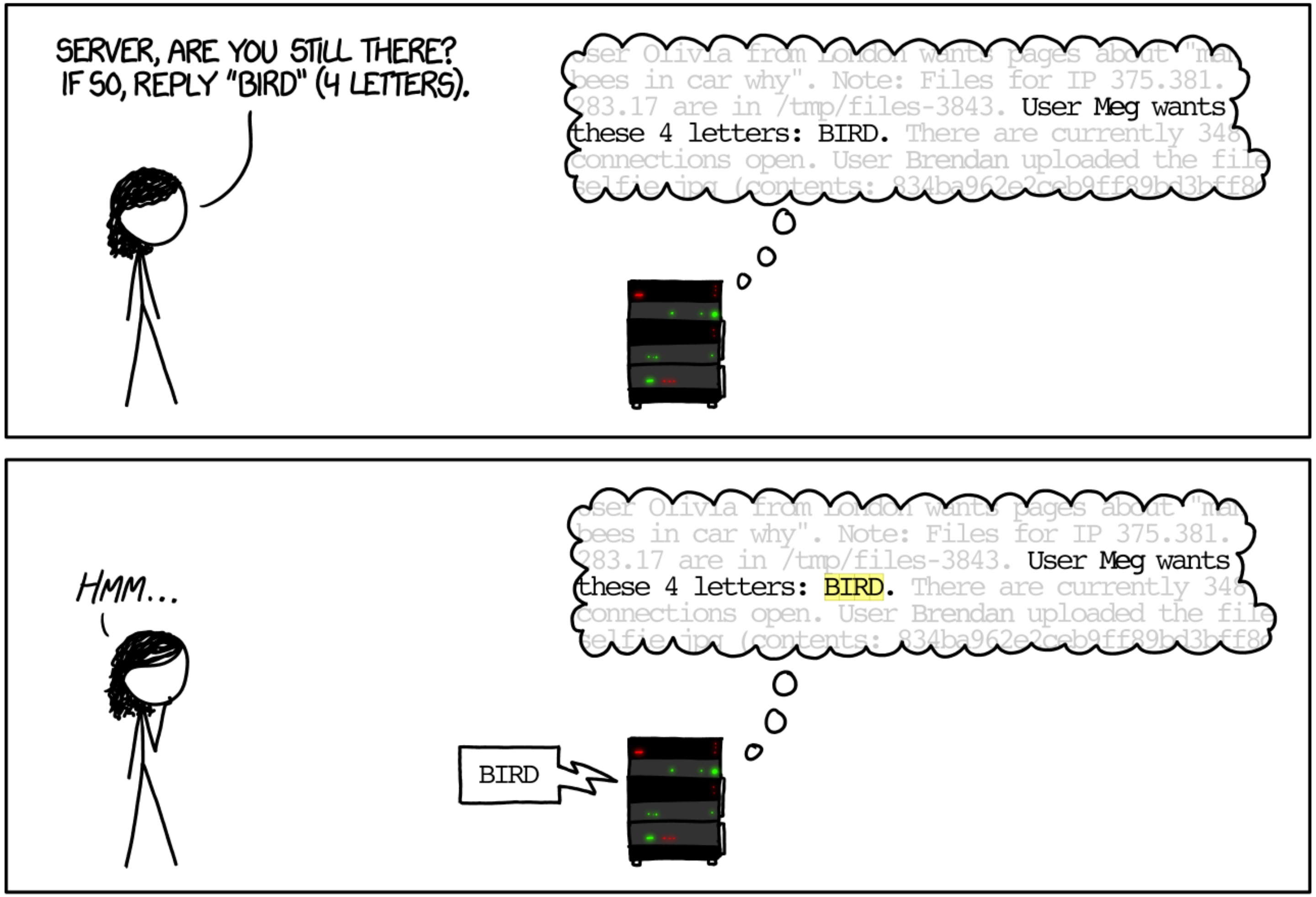
# Example: the original Internet worm (1988)

- 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…

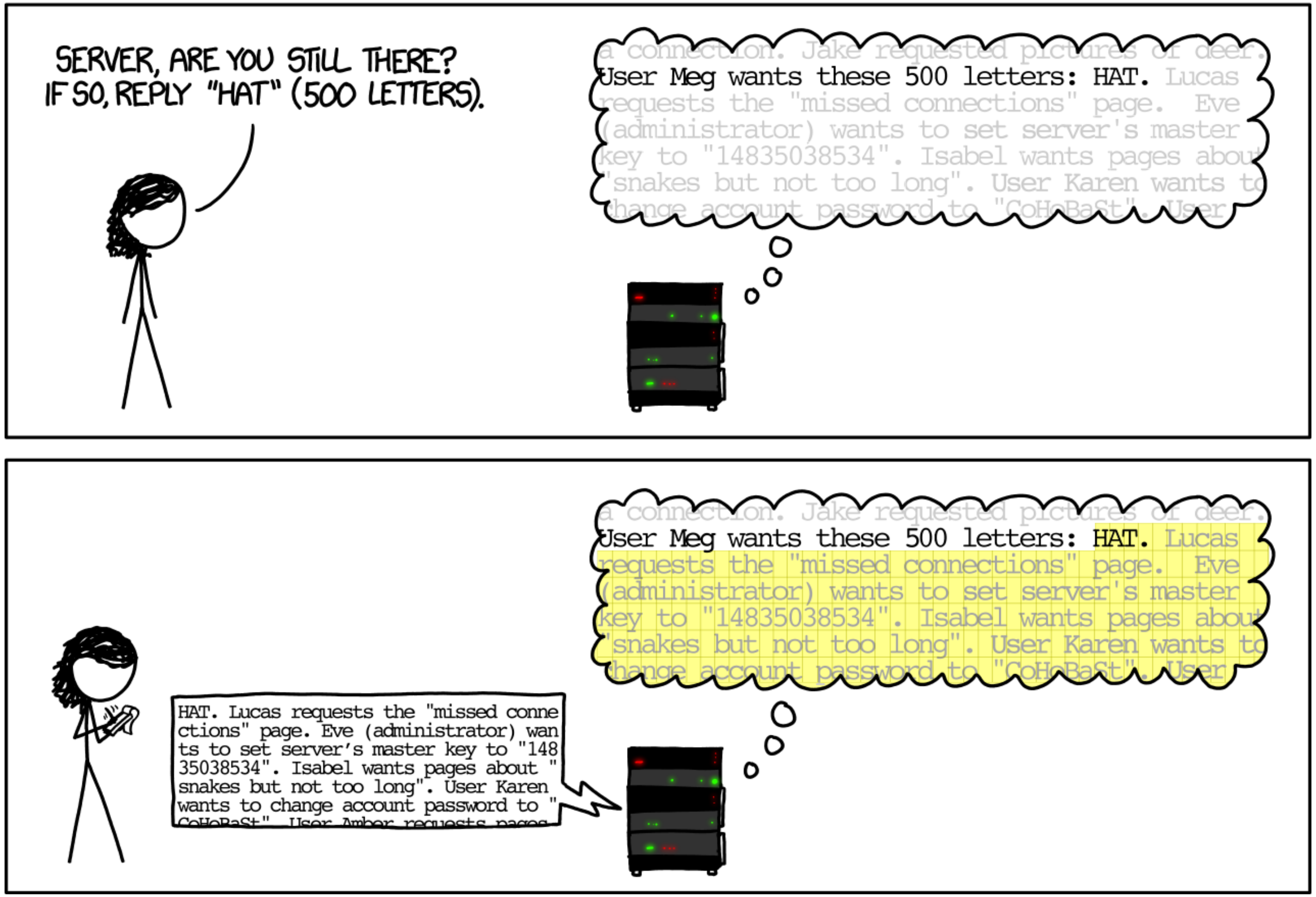
# Example: Heartbleed (2014)



# Example: Heartbleed (2014)

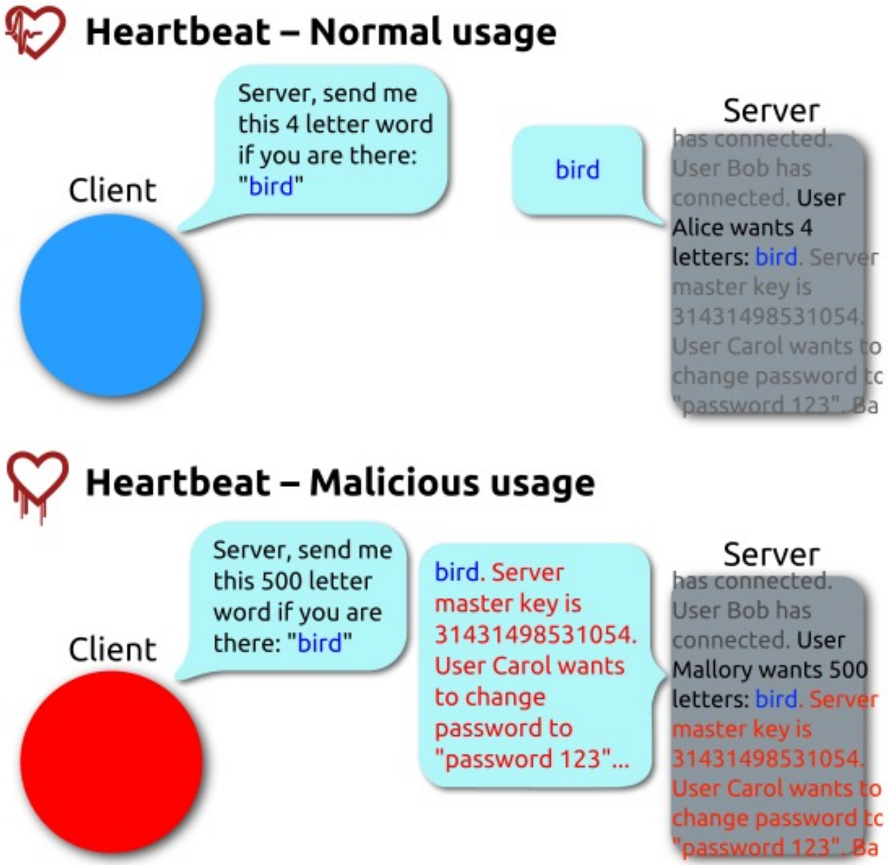


# Example: Heartbleed (2014)



# 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, ...



# Hacking Cars (2010)

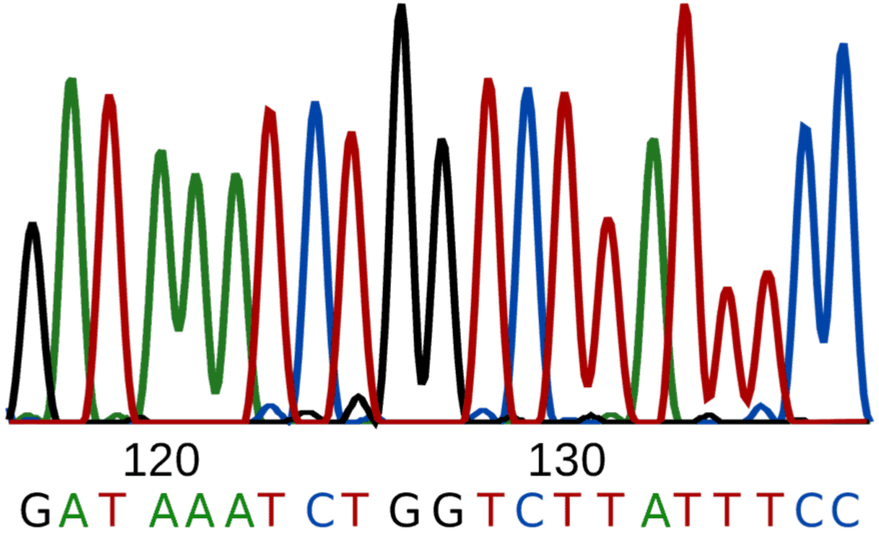
- UW CSE research demonstrated wirelessly hacking a car using buffer overflow
  - http://www.autosec.org/pubs/cars-oakland2010.pdf
- Overwrote the onboard control system's code
  - Disable brakes, unlock doors, turn engine on/off

# Hacking DNA Sequencing Tech (2017)

## Computer Security and Privacy in DNA Sequencing

Paul G. Allen School of Computer Science & Engineering, University of Washington

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



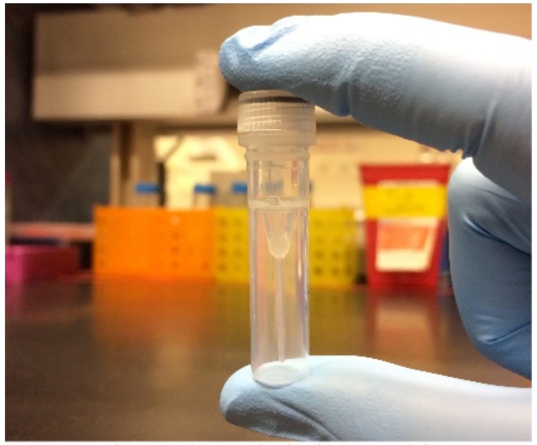



Figure 1: Our synthesized DNA exploit