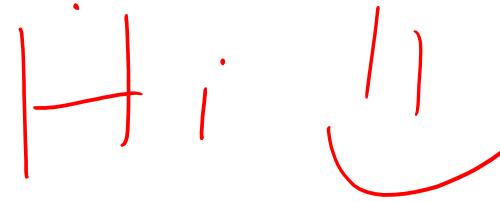


# Buffer Overflows

CSE 351 Autumn 2019



Guest

Instructor:

Andrew Hu

Teaching Assistants:

Andrew Hu

Diya Joy

Maurice Montag

Suraj Jagadeesh

Antonio Castelli

Ivy Yu

Melissa Birchfield

Cosmo Wang

Kaelin Laundry

Millicent Li



# Administrivia

- ❖ Mid-quarter survey due tomorrow (10/31)
  - HW 13 due Nov. 1 (Fri)
  - HW 14 released today due Nov. 4 (Mon)
- ❖ Lab 3 released today, due next Friday (11/8)
  - You will have everything you need by the end of this lecture
- ❖ Midterm grades (out of 100) to be released by Friday
  - 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!
    - Midterm clobber policy can help

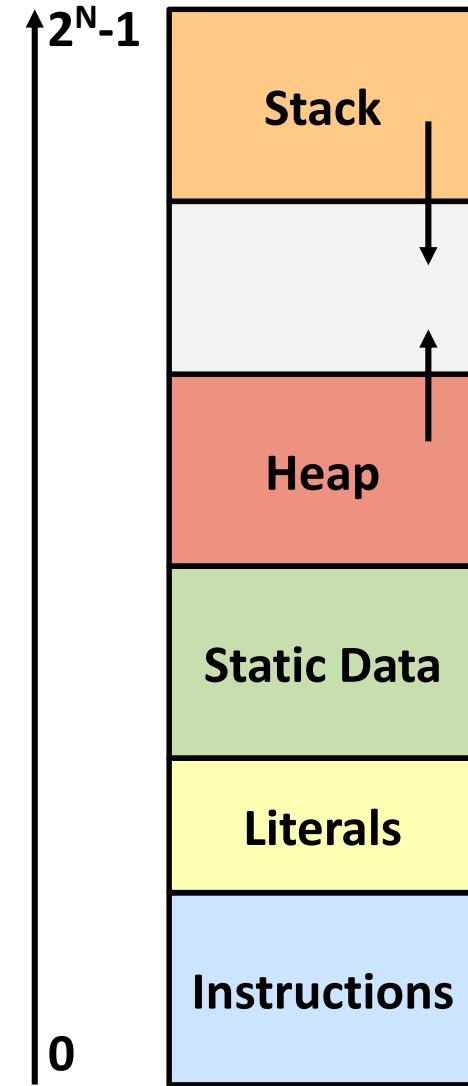
# Buffer Overflows

- ❖ Address space layout (more details!)
- ❖ 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 *like "new", but in C*
  - Dynamically allocated as needed
  - malloc(), calloc(), new, ...
- ❖ Statically allocated Data *from Java*
  - Read/write: global variables (Static Data)
  - Read-only: string literals (Literals)
- ❖ Code/Instructions
  - Executable machine instructions
  - Read-only



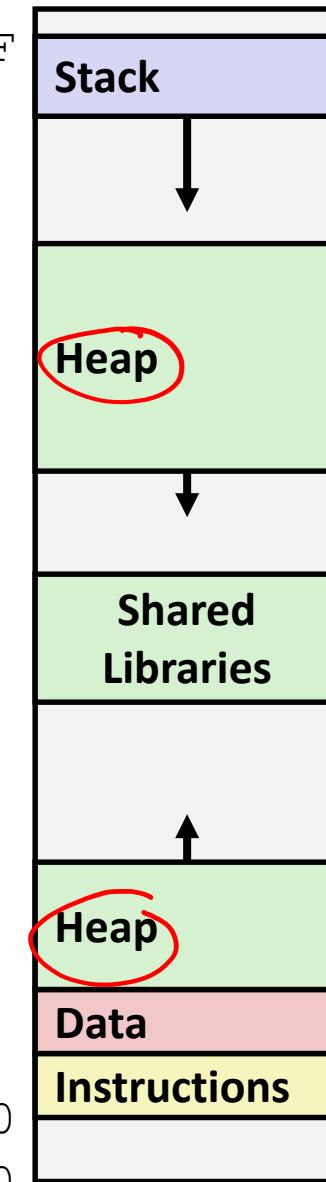
This is extra (non-testable) material

# x86-64 Linux Memory Layout

0x00007FFFFFFFFF

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

Hex Address → 0x400000  
0x000000



*not drawn to scale*

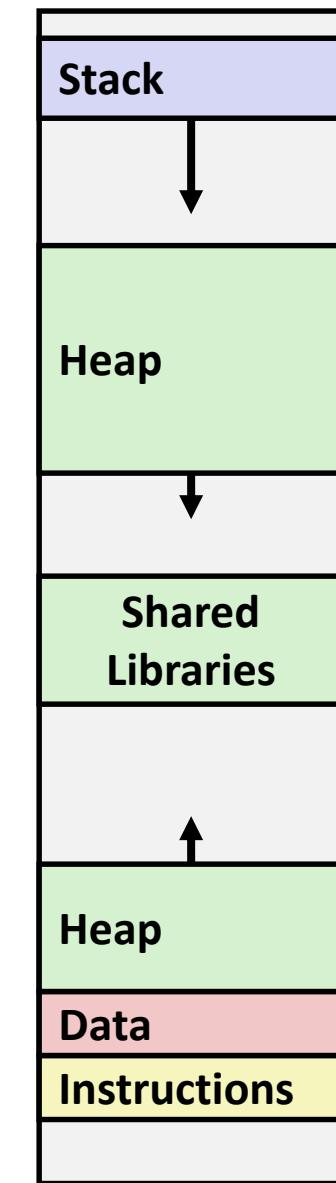
# Memory Allocation Example

```
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?*

*not drawn to scale*

# Memory Allocation Example

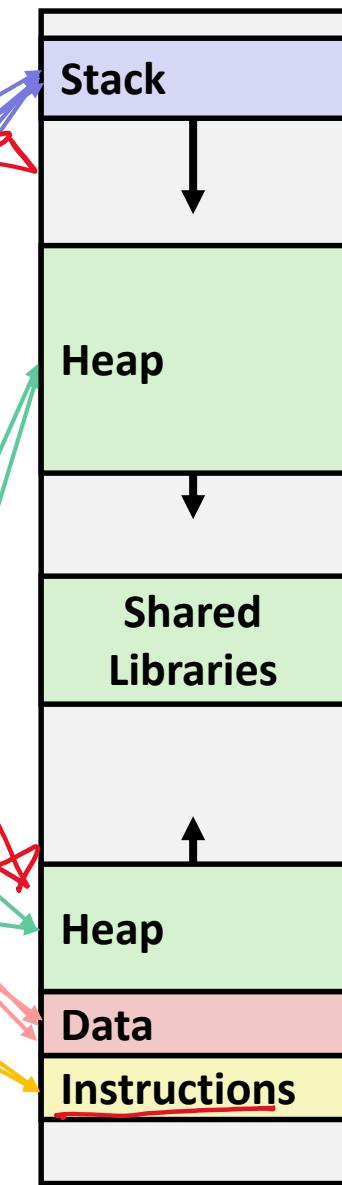
```
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */

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 ... */
}
```

*all go on stack* *V&RS*



*Where does everything go?*

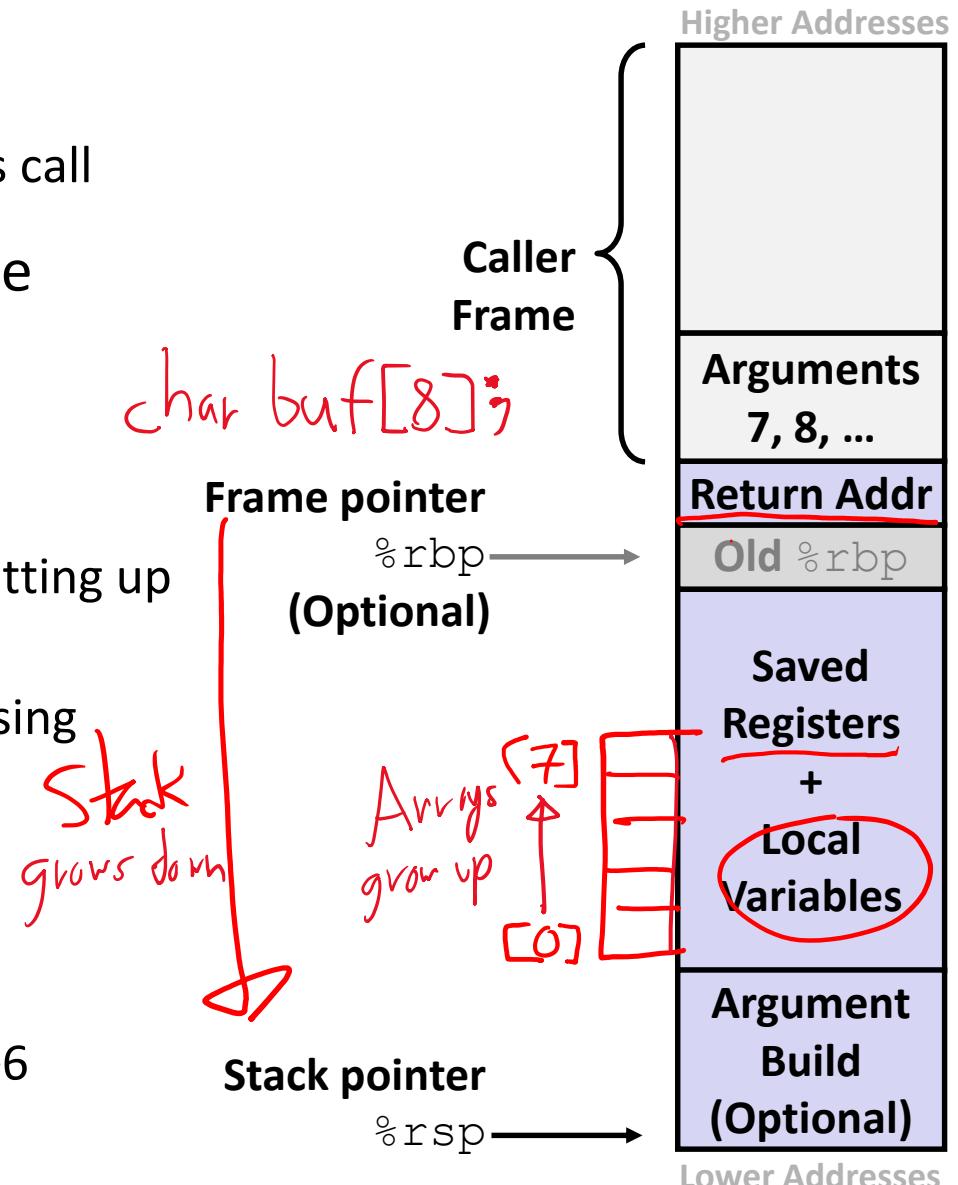
# What Is a Buffer?

- ❖ 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)
  - Caller-saved pushed before setting up arguments for a function call
  - Callee-saved 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

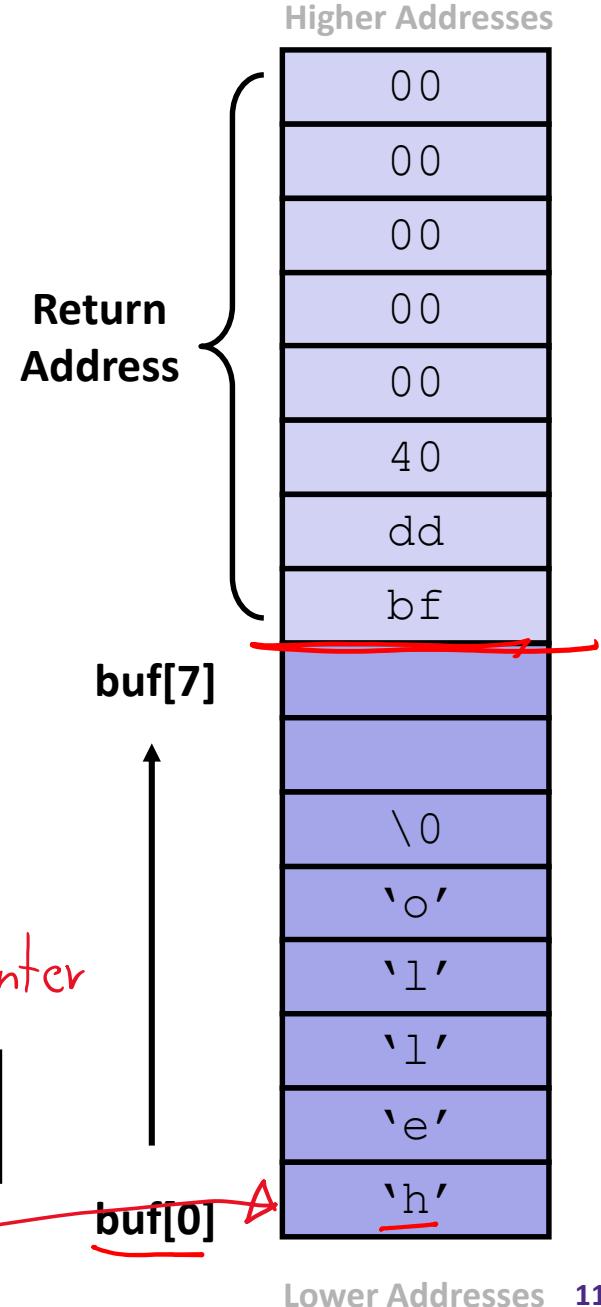
- ❖ C does not check array bounds
  - Many Unix/Linux/C functions don't check argument sizes
  - Allows overflowing (writing past the end) of buffers (arrays)
- ❖ “Buffer Overflow” = Writing past the end of an array
- ❖ Characteristics of the traditional Linux memory layout provide opportunities for malicious programs
  - Stack grows “backwards” in memory
  - Data and instructions both stored in the same memory

# Buffer Overflow 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\n
```

No overflow 😊

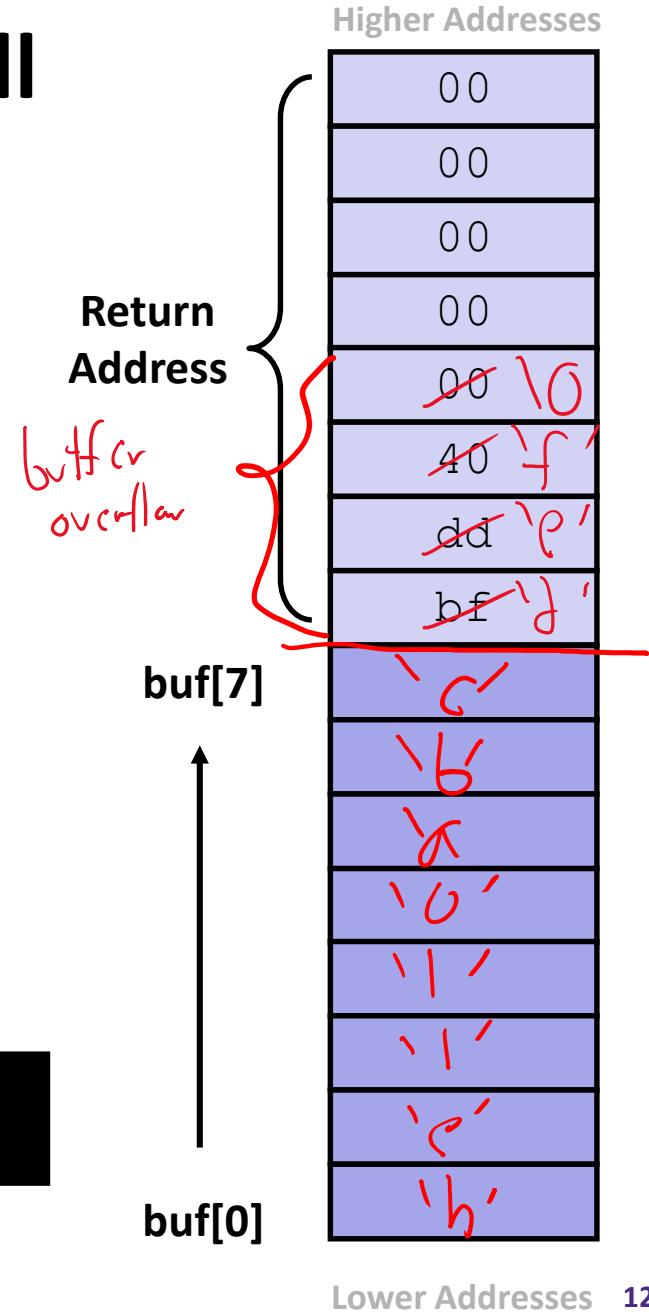


# Buffer Overflow in a Nutshell

- ❖ 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! 😥



# 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
  - 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

- ❖ 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

- ❖ 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;
}
```



Only stop when you  
see '\n'

- 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[8]; /* Way too small! */
    gets(buf); ← read input into buffer
    puts(buf); ← print output from buffer
}
```

Check out the  
code on the website

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

- Compiler warning for  
gets()!

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

```
unix> ./buf-nsp
Enter string: 1234567890123456
Illegal instruction
```

```
unix> ./buf-nsp
Enter string: 12345678901234567
Segmentation Fault
```

# Buffer Overflow Disassembly (buf-nsp)

## echo:

```
0000000000400597 <echo>:  
400597: 48 83 ec 18  
...  
4005aa: 48 8d 7c 24 08  
4005af: e8 d6 fe ff ff  
4005b4: 48 89 7c 24 08  
4005b9: e8 b2 fe ff ff  
4005be: 48 83 c4 18  
4005c2: c3
```

24

**sub** \$0x18,%rsp *Compiler choice*

... calls printf ...

**lea** 0x8(%rsp),%rdi

**callq** 400480 <gets@plt>

**lea** 0x8(%rsp),%rdi

**callq** 4004a0 <puts@plt>

**add** \$0x18,%rsp

**retq**

## call\_echo:

```
00000000004005c3 <call_echo>:  
4005c3: 48 83 ec 08  
4005c7: b8 00 00 00 00  
4005cc: e8 c6 ff ff ff  
4005d1: 48 83 c4 08  
4005d5: c3
```

**sub** \$0x8,%rsp

**mov** \$0x0,%eax

**callq** 400597 <echo>

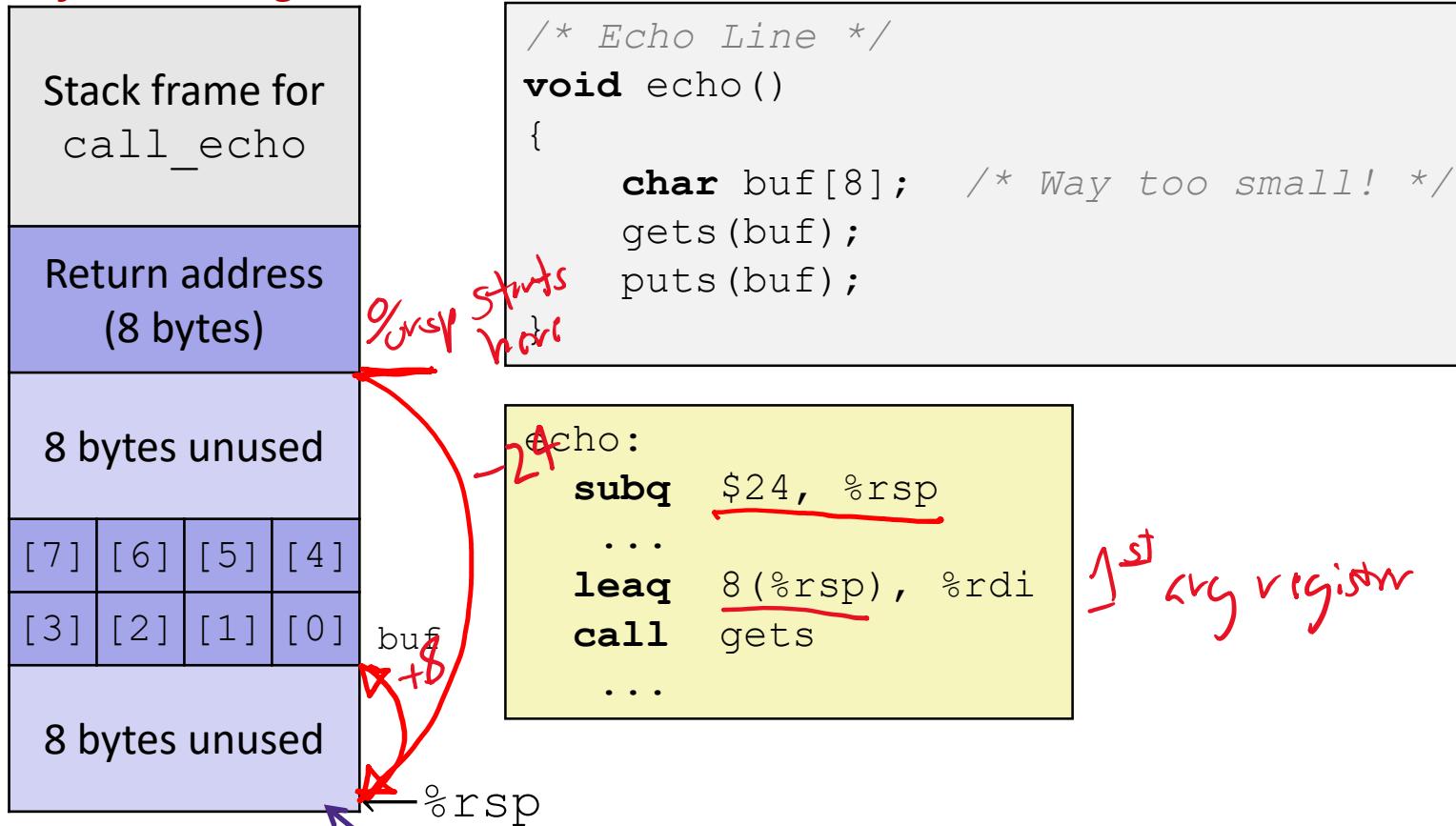
**add** \$0x8,%rsp

**retq**

return address placed on stack

# Buffer Overflow Stack

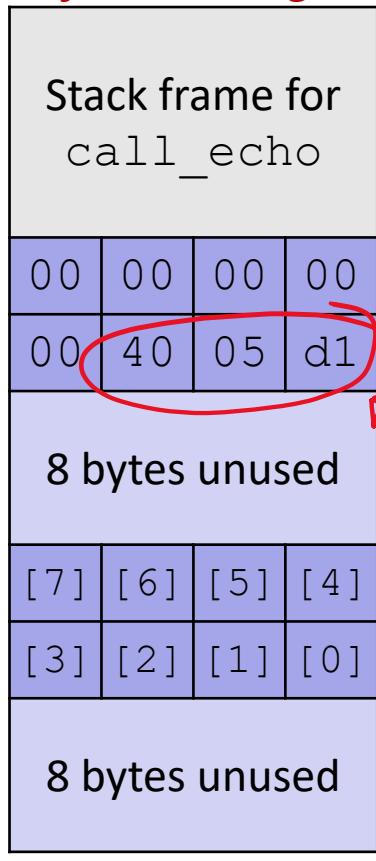
*Before call to gets*



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

# Buffer Overflow Example

**Before call to gets**



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

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

**call\_echo:**

```
. . .
4005cc: callq 400597 <echo>
4005d1: add    $0x8,%rsp
. . .
```

# Buffer Overflow Example #1

## *After call to gets*

Stack frame for call_echo			
00	00	00	00
00	40	05	d1
00	35	34	33
32	31	30	39
38	37	36	35
34	33	32	31
8 bytes unused			

**Note:** Digit "N" is just  $0x3N$  in ASCII!

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

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

**call echo:**

~~call\_e~~

```
...  
4005cc:  callq   400597 <echo>  
4005d1:  add     $0x8,%rsp
```

$$0x\textcolor{red}{\underline{31}} = '1'$$

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

## 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	05	00
<u>36</u>	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);  
}

Victim address  
is now 0x400500

call\_echo:

...  
4005cc: callq 400597 <echo>  
4005d1: add \$0x8,%rsp  
...

unix> ./buf-nsp  
Enter string: 1234567890123456  
Illegal instruction

Overflowed buffer and corrupted return pointer

# Buffer Overflow Example #2 Explained

*After return from echo*

Stack frame for call _echo			
00	00	00	00
00	40	05	00
36	35	34	33
32	31	30	39
38	37	36	35
34	33	32	31
8 bytes unused			

00000000004004f0 <deregister\_tm\_clones>:  
4004f0: push %rbp  
4004f1: mov \$0x601040,%eax  
4004f6: cmp \$0x601040,%rax  
4004fc: mov %rsp,%rbp  
4004ff: je 400518 *2nd byte of this instruction*  
400501: mov \$0x0,%eax  
400506: test %rax,%rax  
400509: je 400518  
40050b: pop %rbp  
40050c: mov \$0x601040,%edi  
400511: jmpq \*%rax  
400513: nopl 0x0(%rax,%rax,1)  
400518: pop %rbp  
400519: retq

“Returns” to a byte that is not the beginning of an instruction,  
so program signals SIGILL, Illegal instruction

# Malicious Use of Buffer Overflow: Code Injection Attacks

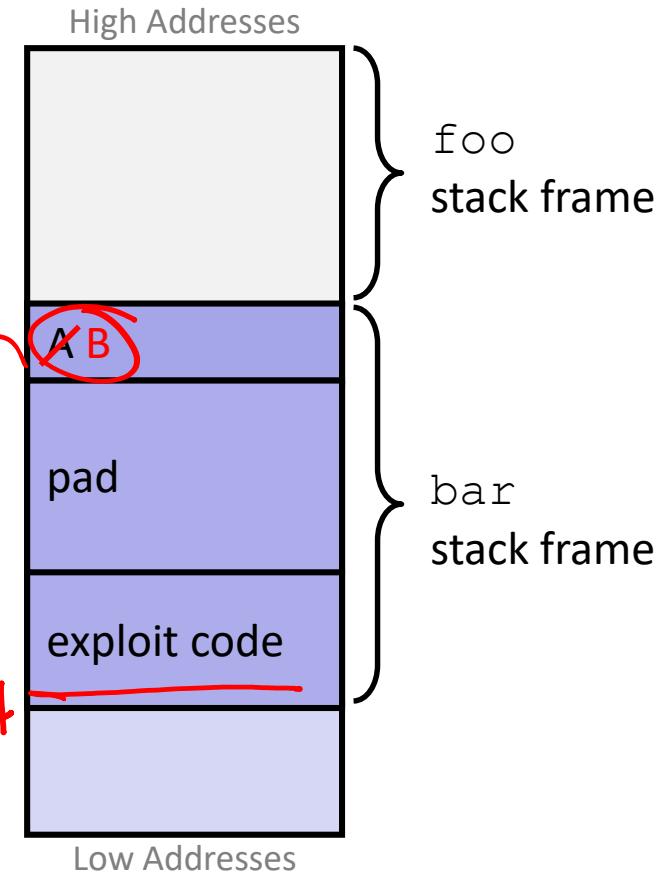
```
void foo() {  
    bar();  
    A: ...  
}
```

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

return address A

data written  
by gets()  
buf starts here → B

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

# Peer Instruction 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)  
 $-64 + 16 = -48$

Previous stack frame			
00	00	00	00
00	40	05	d1
CAF	E	FOOD	
...			
			[0]

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

6 bytes

$$-64 + 16 = -48$$

$$48 + 6 = 54$$

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**

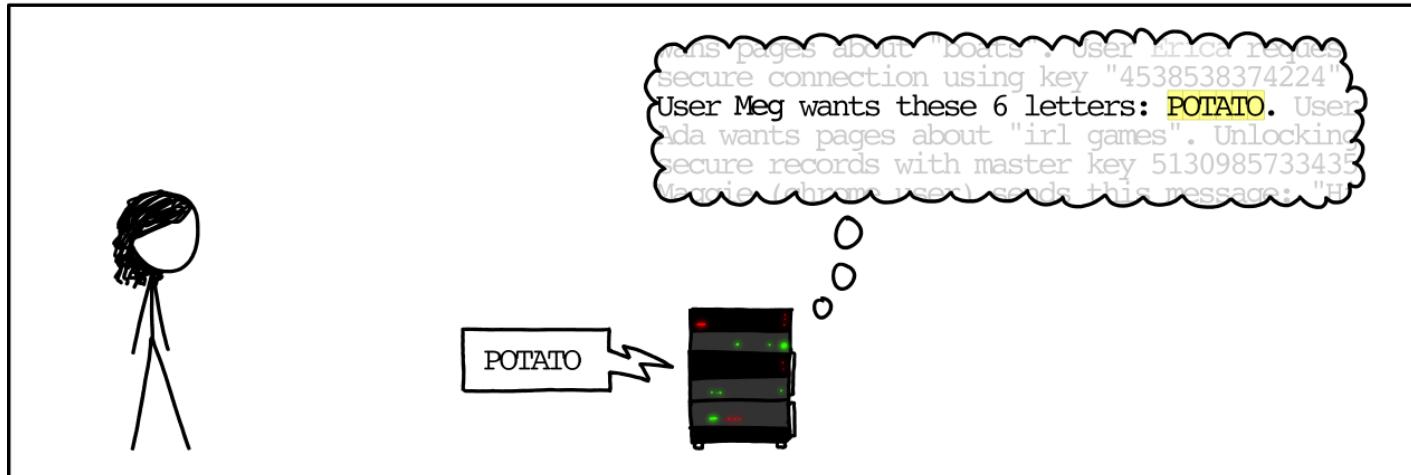
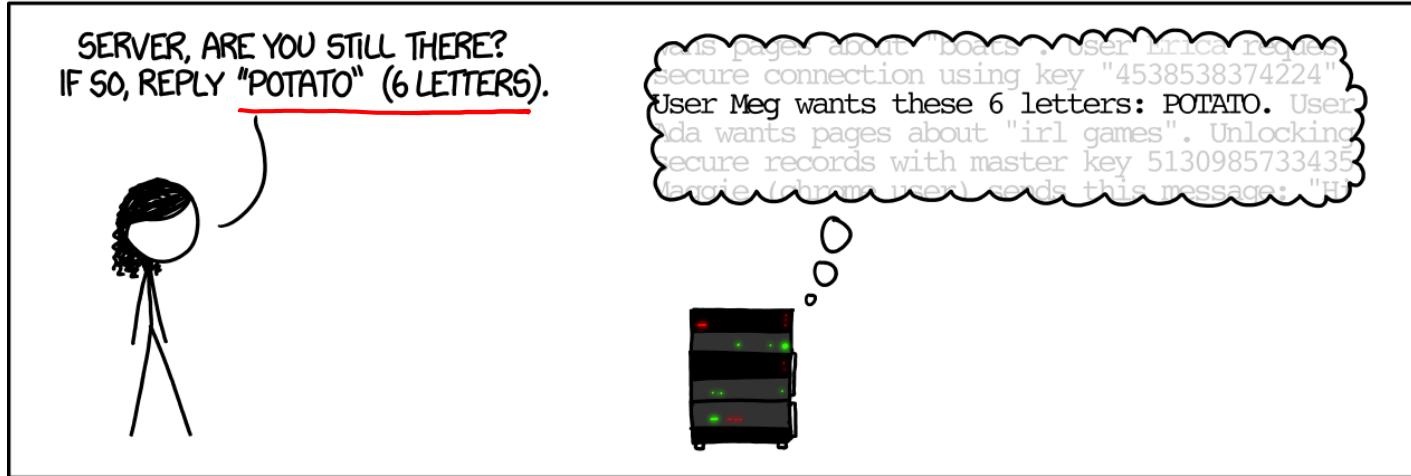
- ❖ Distressingly common in real programs
  - Programmers keep making the same mistakes 😞
  - Recent measures make these attacks much more difficult
- ❖ Examples across the decades
  - Original “Internet worm” (1988)
  - Heartbleed (2014, affected 17% of servers)
    - Similar issue in Cloudbleed (2017)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

# 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

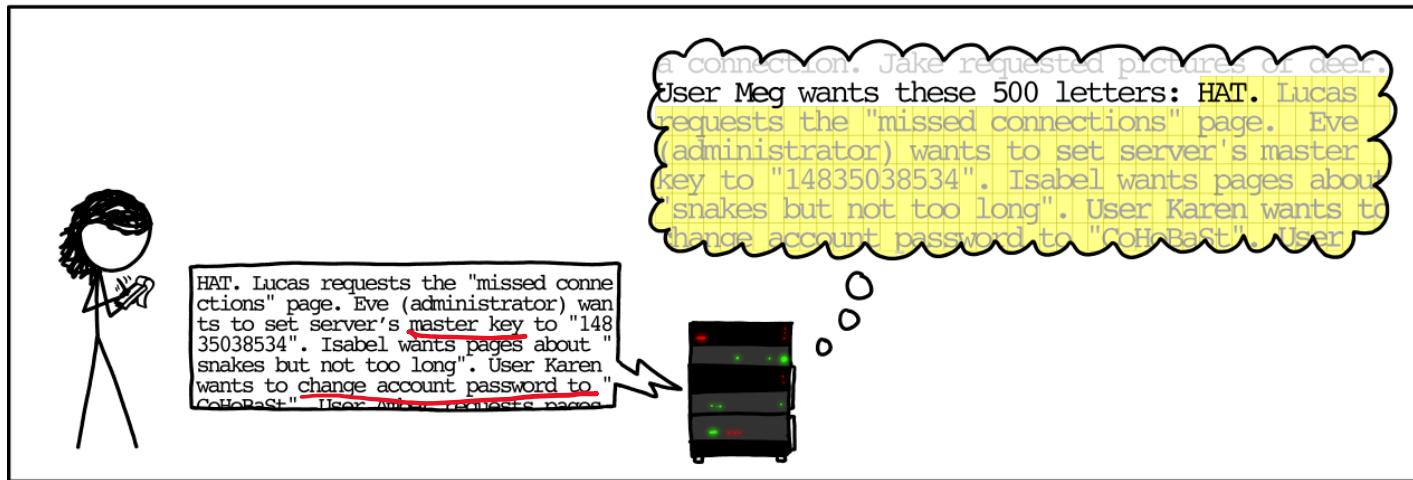
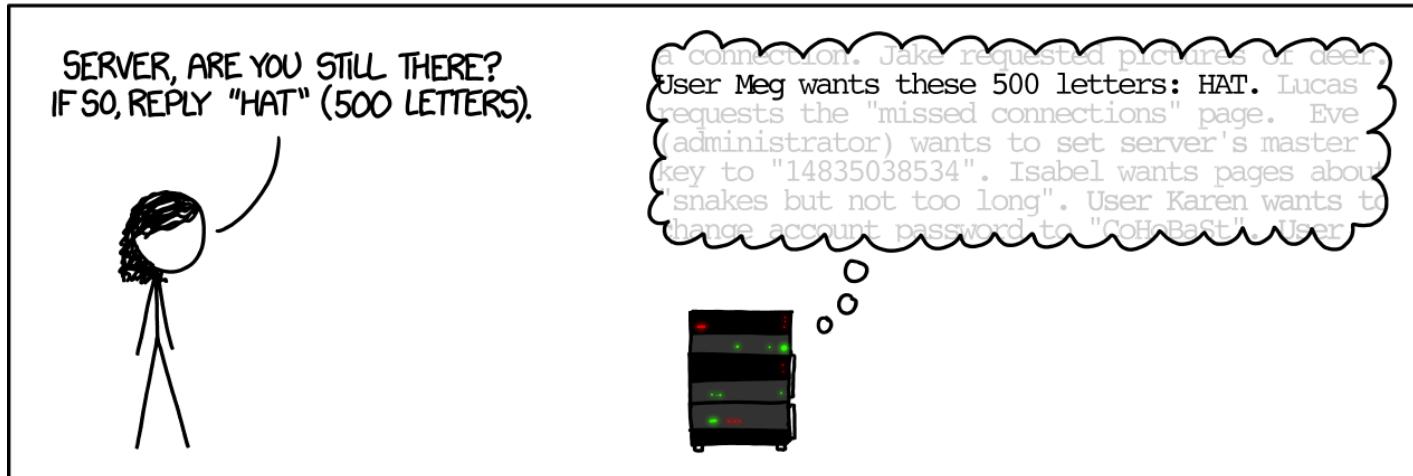
## HOW THE HEARTBLEED BUG WORKS:



# Example: Heartbleed

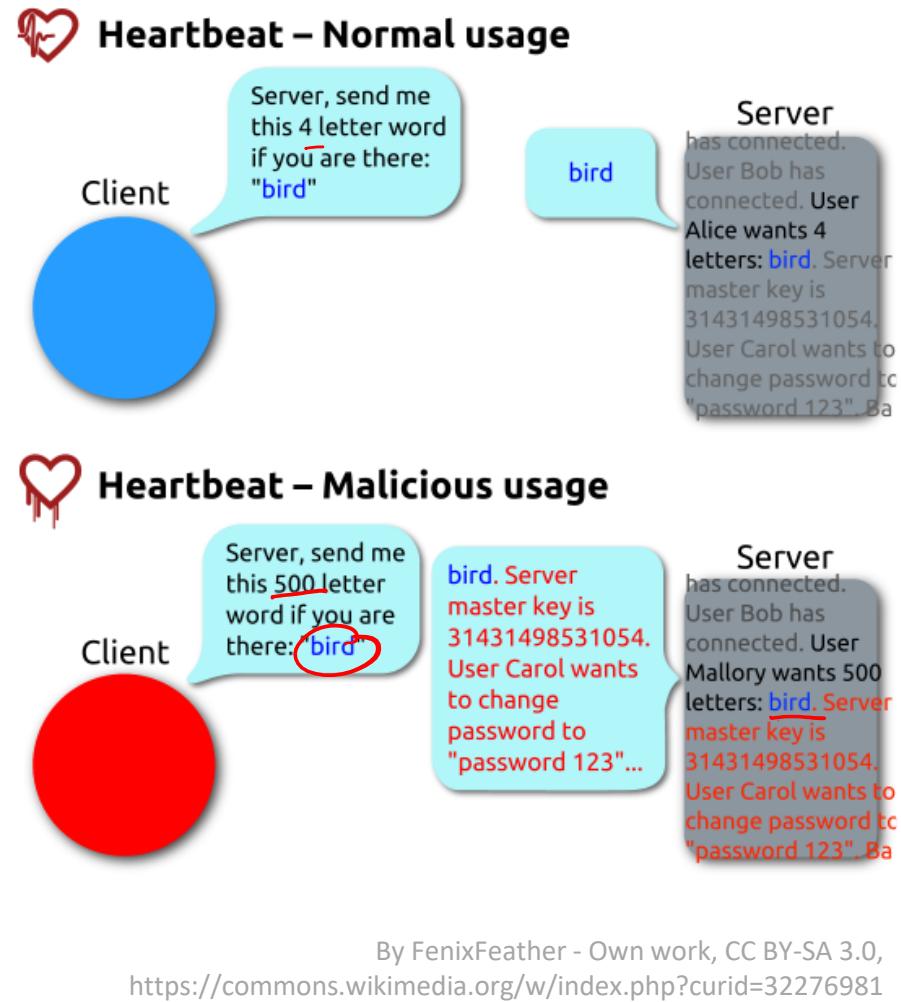


# Example: Heartbleed



# 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, ...



# Hacking Cars

- ❖ UW CSE research from 2010 demonstrated wirelessly hacking a car using buffer overflow
- ❖ Overwrote the onboard control system's code
  - Disable brakes
    - Cause the car to crash
  - Unlock doors
    - Stop the car
  - Turn engine on/off
    - Stop the car



# Hacking DNA Sequencing Tech

- ❖ 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/>



## Computer Security and

Paul G. Allen School of Computer Science

There has been rapid improvement in the cost and speed of DNA sequencing over the past decade, the cost to sequence a human genome has dropped from \$10,000 in 2001 to under \$1,000 in 2012. This was made possible by faster, massively parallel processing techniques that allow for the simultaneous sequencing of hundreds of millions of DNA strands simultaneously. This has led to many applications ranging from personalized medicine, ancestry, and forensic science to environmental monitoring and agriculture.

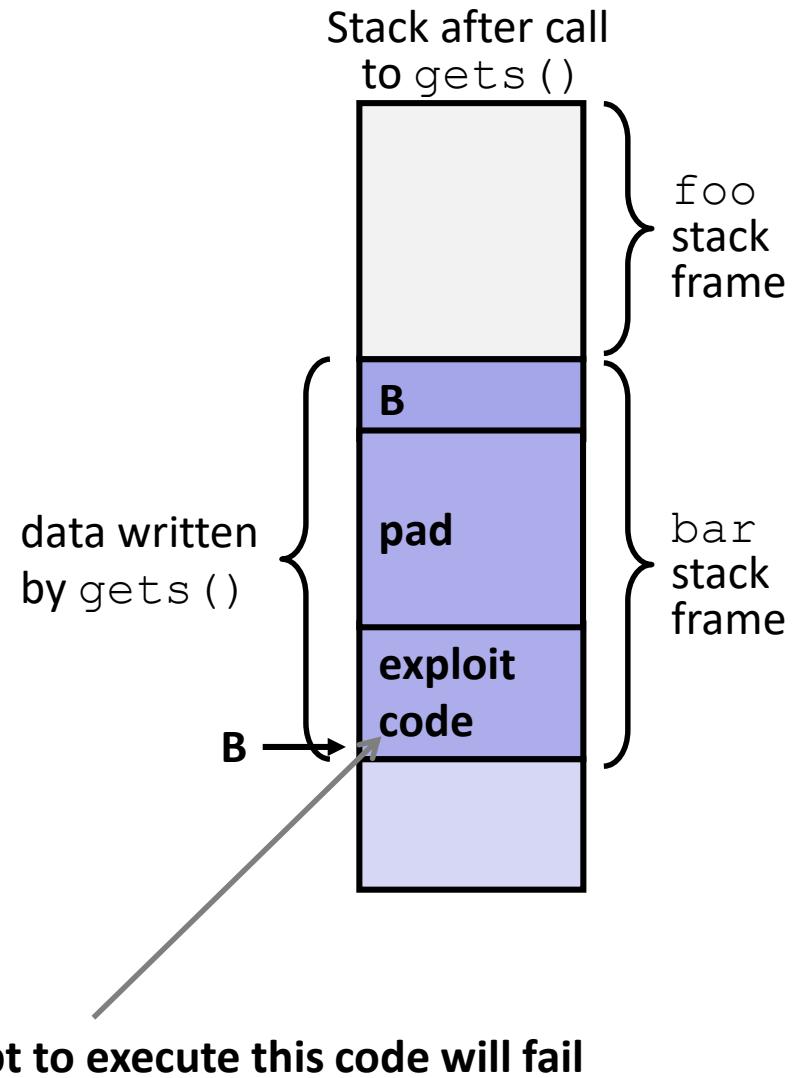
Figure 1: Our synthesized DNA exploit

# Dealing with buffer overflow attacks

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

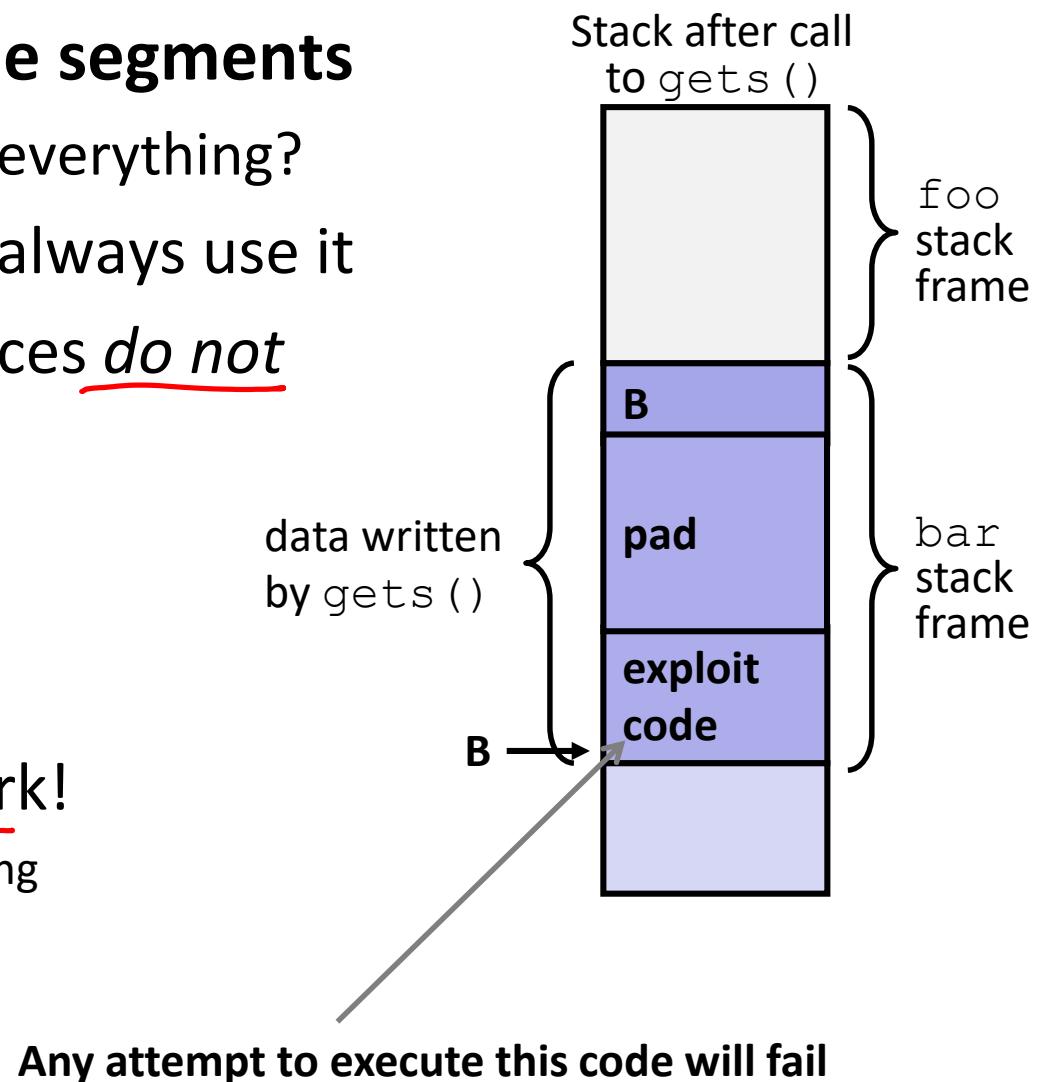
# 1) System-Level Protections

- ❖ **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



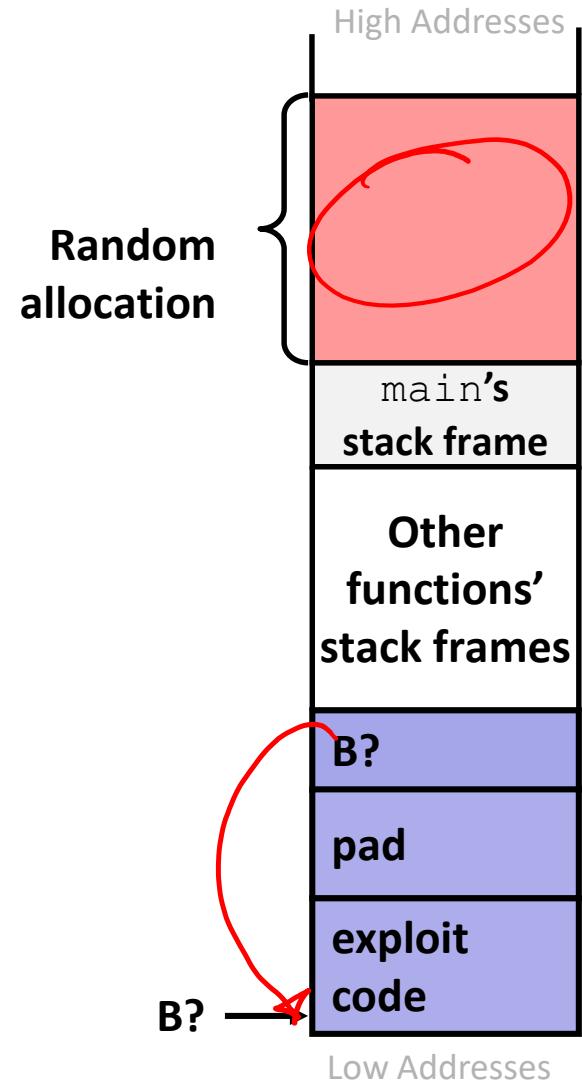
# 1) System-Level Protections

- ❖ **Non-executable code segments**
  - Wait, doesn't this fix everything?
  - ❖ Works well, but can't always use it
  - ❖ Many embedded devices do not have this protection
    - 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: Code from Slide 6 executed 5 times; address of variable `local` =
  - ~~0x7ffd19d3f8ac~~
  - ~~0x7ffe8a462c2c~~
  - 0x7ffe927c905c
  - 0x7ffefd5c27dc
  - 0x7ffffa0175afc
  - Stack repositioned each time program executes



## 2) Avoid Overflow Vulnerabilities in Code

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

A red circle highlights the number 8 in the line `fgets(buf, 8, stdin);`. A red arrow points from this circle to the text "character read limit" written in red.

- ❖ Use library routines that limit string lengths
  - `fgets` instead of `gets` (2<sup>nd</sup> 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

## 2) 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`
  - Rust language was designed with security in mind
    - Panics on index out of bounds, plus more protections

# 3) Stack Canaries

- ❖ Basic Idea: place 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 ***
```

# Protected Buffer Disassembly (buf)

This is extra  
(non-testable)  
material

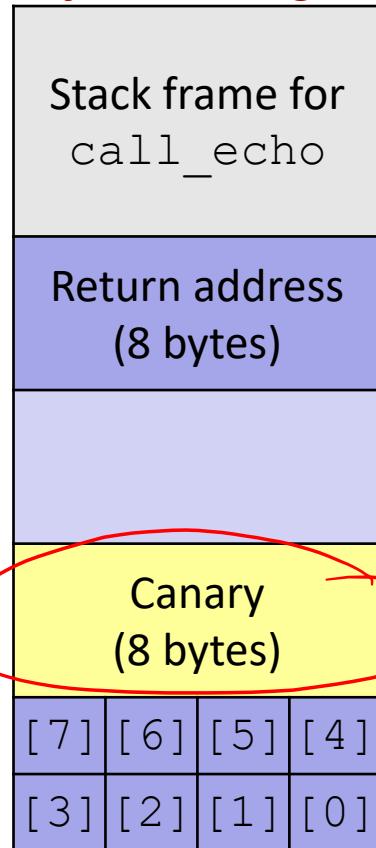
echo:

```
400607: sub    $0x18,%rsp
40060b: mov    %fs:0x28,%rax # read canary value
400614: mov    %rax,0x8(%rsp) # store canary on Stack
400619: xor    %eax,%eax    # erase canary from register
...
...    ... call printf ...
400625: mov    %rsp,%rdi
400628: callq  400510 <gets@plt>
40062d: mov    %rsp,%rdi
400630: callq  4004d0 <puts@plt>
400635: mov    0x8(%rsp),%rax # read current canary on Stack
40063a: xor    %fs:0x28,%rax # compare against original value
400643: jne    40064a <echo+0x43> # if unchanged, then return
400645: add    $0x18,%rsp
400649: retq
40064a: callq  4004f0 <__stack_chk_fail@plt> # stack smashing
                                                 detected
```

try: diff buf-nsp.s buf.s

# Setting Up Canary

*Before call to gets*



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

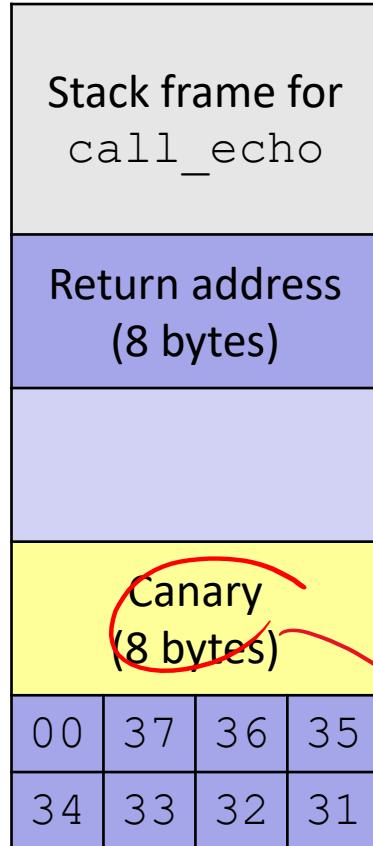
echo:

```
    . . .
    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[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
```

buf ← %rsp

**Input: 1234567**

compare

after vs. before

if different, crash

This is extra  
(non-testable)  
material

# Summary of Prevention Measures

## 1) Employ system-level protections

- Code on the Stack is not executable
- Randomized Stack offsets

## 2) Avoid overflow vulnerabilities

- Use library routines that limit string lengths
- Use a language that makes them impossible

## 3) Have compiler use “stack canaries”

# Think this is cool?

- ❖ You'll love Lab 3 😊
  - Released today, due next Friday (11/8)
  - Check out the buffer overflow simulator!
- ❖ 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=hB6eY73sLV>

# Extra Notes about %rbp

This is extra  
(non-testable)  
material

- ❖  $\%rbp$  is used to store the frame pointer
  - Name comes from “base pointer”
- ❖ You can refer to a variable on the stack as  $\%rbp + \text{offset}$
- ❖ The base of the frame will never change, so each variable can be uniquely referred to with its offset
- ❖ The top of the stack ( $\%rsp$ ) may change, so referring to a variable as  $\%rsp - \text{offset}$  is less reliable
  - For example, if you need save a variable for a function call, pushing it onto the stack changes  $\%rsp$