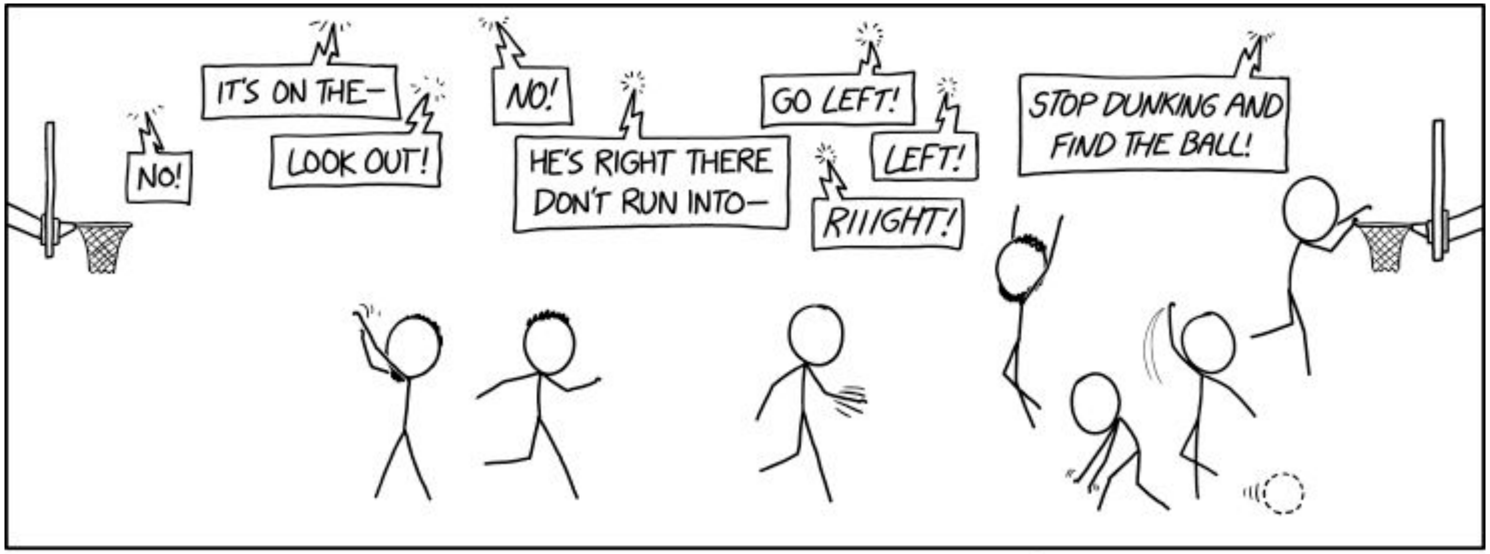
Buffer Overflows

CSE 351 Summer 2021

Instructor:
Mara Kirdani-Ryan

Teaching Assistants:
Kashish Aggarwal
Nick Durand
Colton Jobes
Tim Mandzyuk



NO ONE LIKED MY NEW SPORTS SYSTEM, IN WHICH EACH PLAYER IS IN A SEPARATE ARENA SHARING A SINGLE VIRTUAL BALL THAT THEY CAN'T SEE WHILE ONLINE VIEWERS YELL INSTRUCTIONS, BUT IT WAS FUN TO WATCH WHILE IT LASTED.

http://xkcd.com/2291/

Gentle, Loving Reminders

- hw12 due Friday (7/23)
- hw13 due Monday (7/26)
- *Lab 2 due tonight!* (7/21)
  - Extra Credit portion – make sure you also submit to the Lab 2 Extra Credit assignment on Gradescope
- Lab 3 released!
  - Today's lecture on buffer overflow.
  - You get to write some buffer overflow exploits!

Learning Objectives

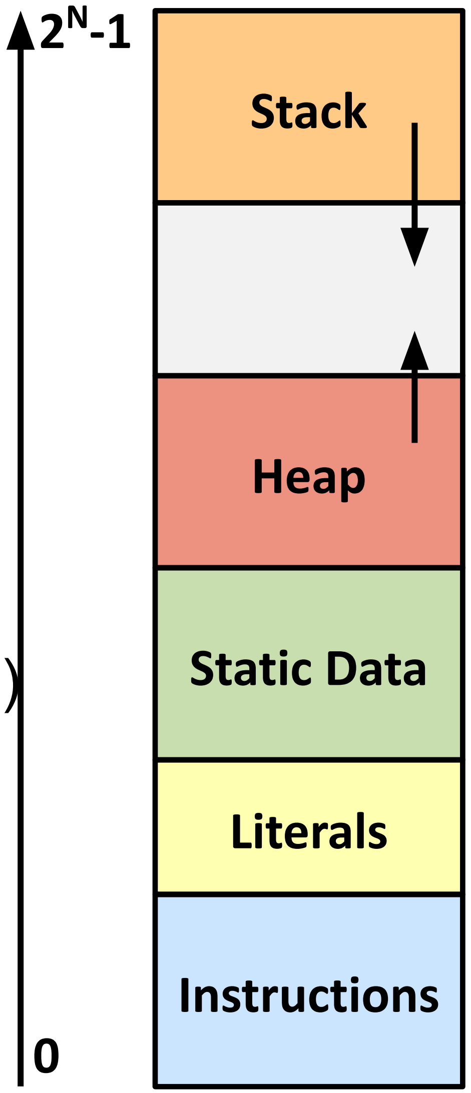
Understanding this lecture means you can:

- Explain what buffers are, and why they can overflow
- Perform buffer overflow exploits on vulnerable code (Lab 3)
- Explain a few mitigations strategies for buffer overflows
- Critically read technologies through the lens of agency, support, & access
- Explain race (and other modes of oppression) as technologies

not drawn to scale

Review: General Memory Layout

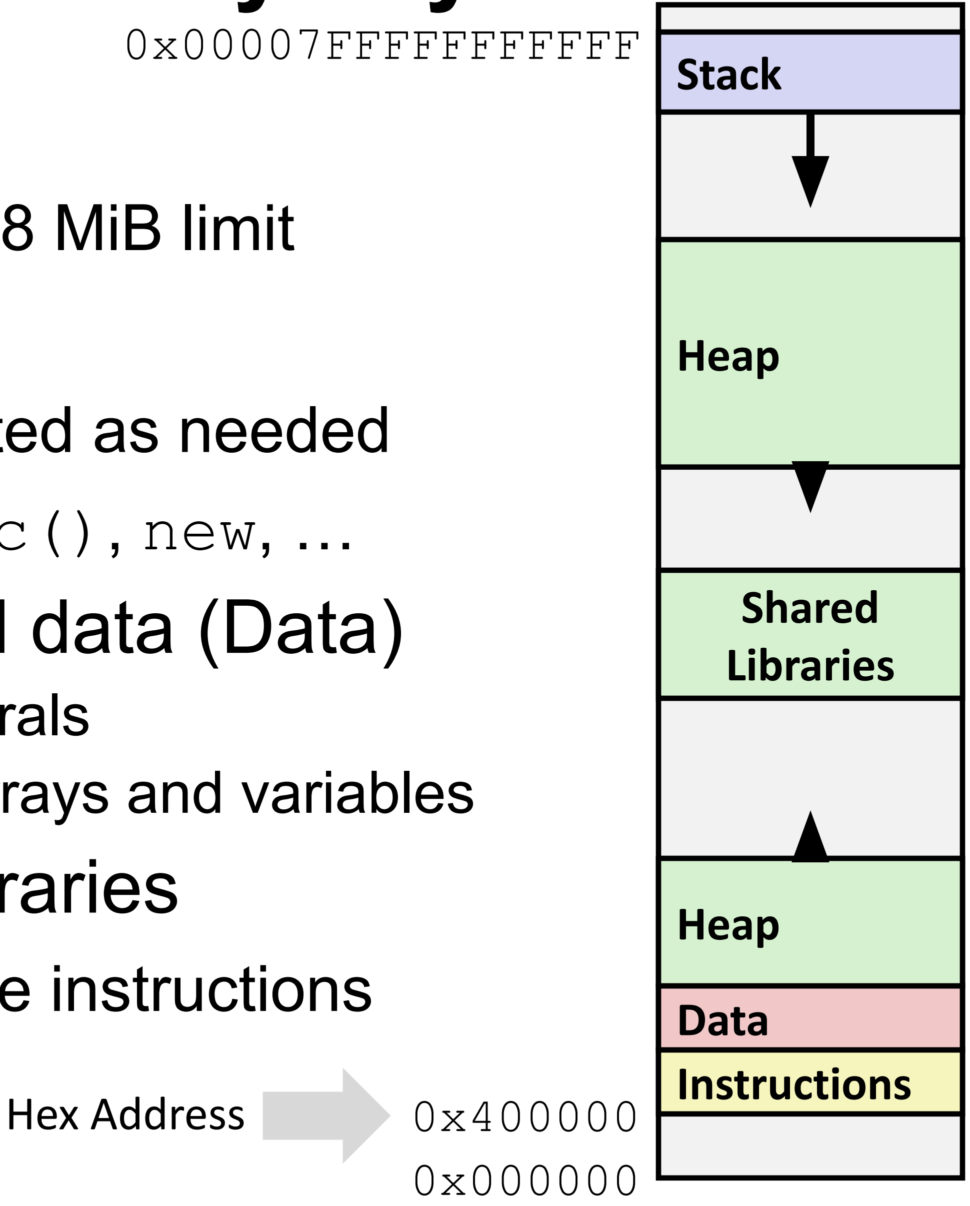
- Stack
  - Local variables (procedure context)
- Heap
  - Dynamically allocated as needed
  - `malloc(), calloc(), new, …`
- Statically allocated Data
  - Read/write: global variables (Static Data)
  - Read-only: string literals (Literals)
- Code/Instructions
  - Executable machine instructions
  -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



This is extra (non-testable) material

x86-64 Linux Memory Layout

- 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



not drawn to scale

Memory Allocation Example

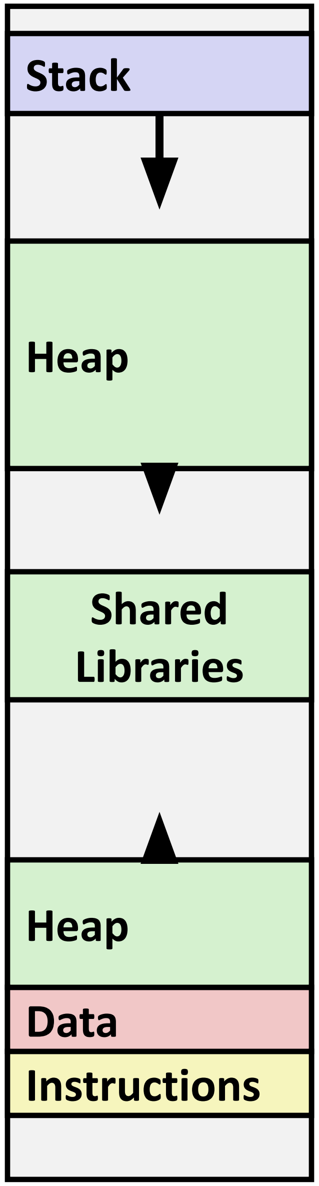
```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

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

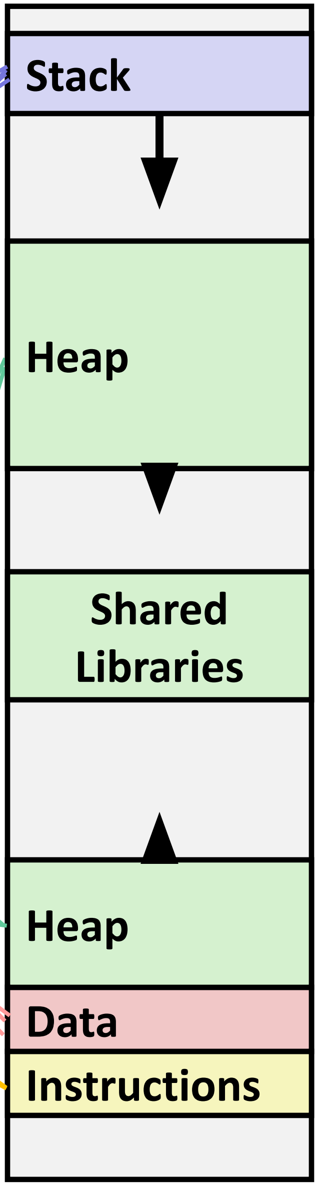
Memory Allocation Example

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



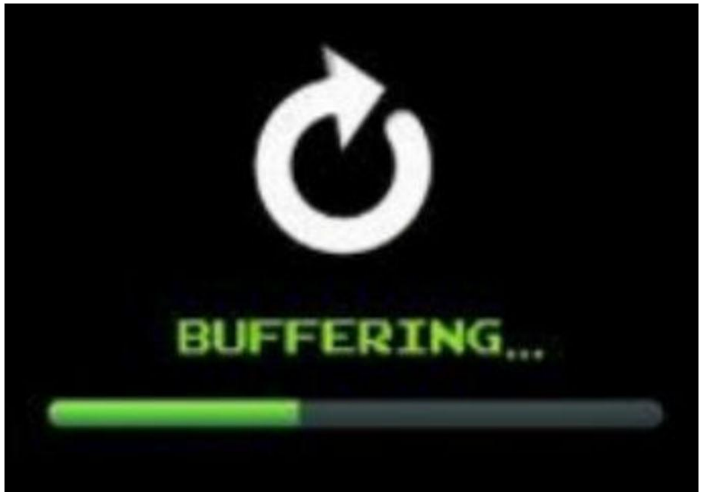
Where does everything go?

What Is a Buffer?

- A buffer is an array used to temporarily store data

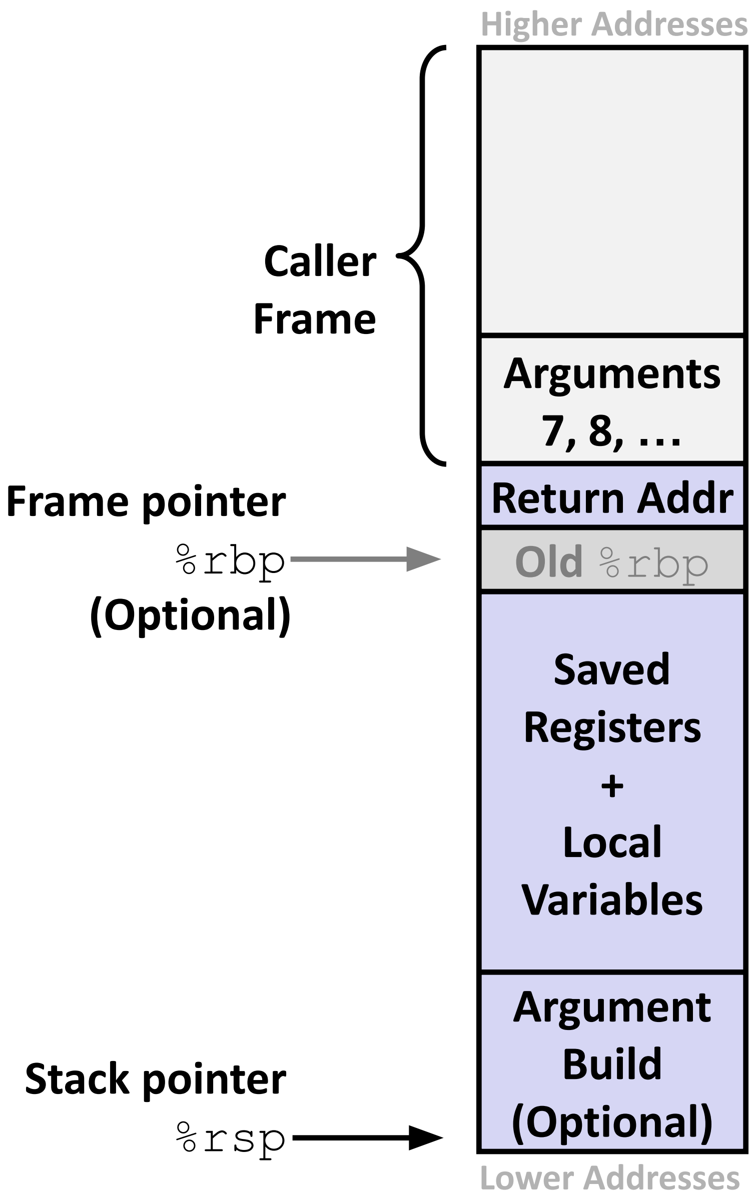
- You've probably seen "video buffering…"
  - The video is being written into a buffer before being played

- 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 arg sizes
  - Allows overflowing (writing past the end) of buffers (arrays)

- “Buffer Overflow” = Writing past end of an array

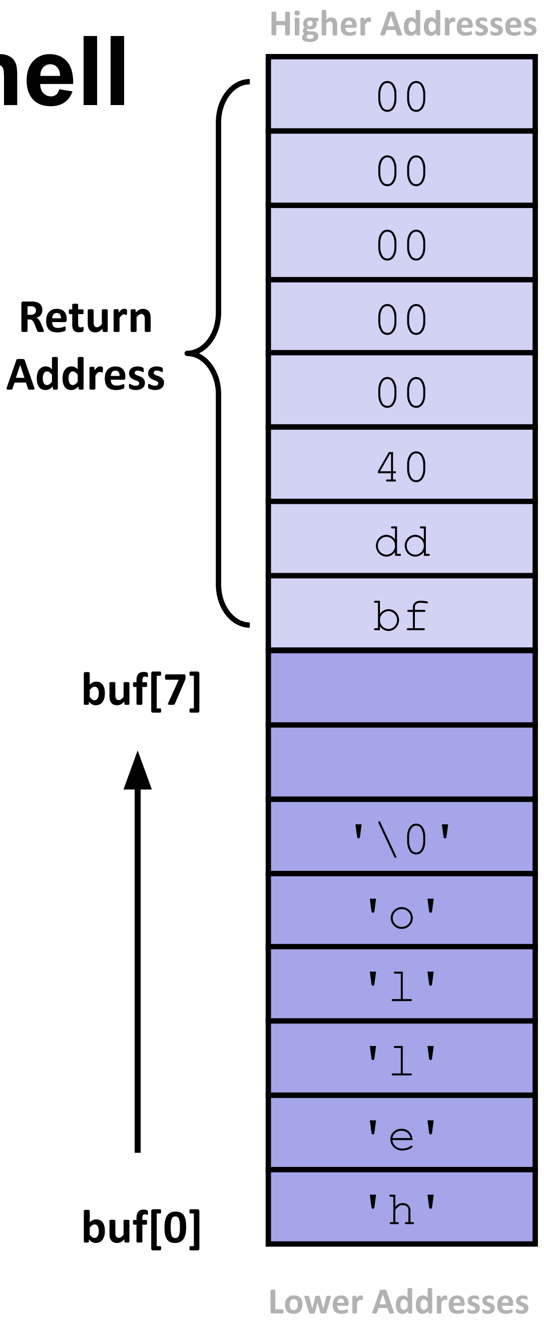
- Linux memory layout provides opportunities for malicious programs
  - Stack grows “backwards” in memory (downwards)
  - Data and instructions both stored in the same memory

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
```

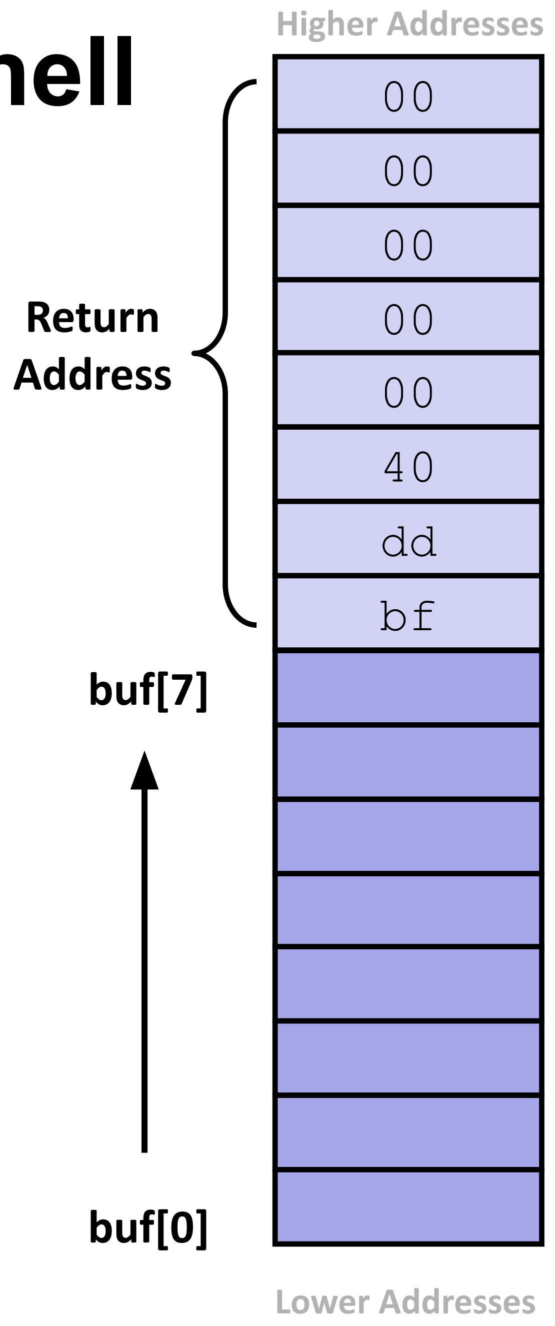
No overflow ☺️



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
```

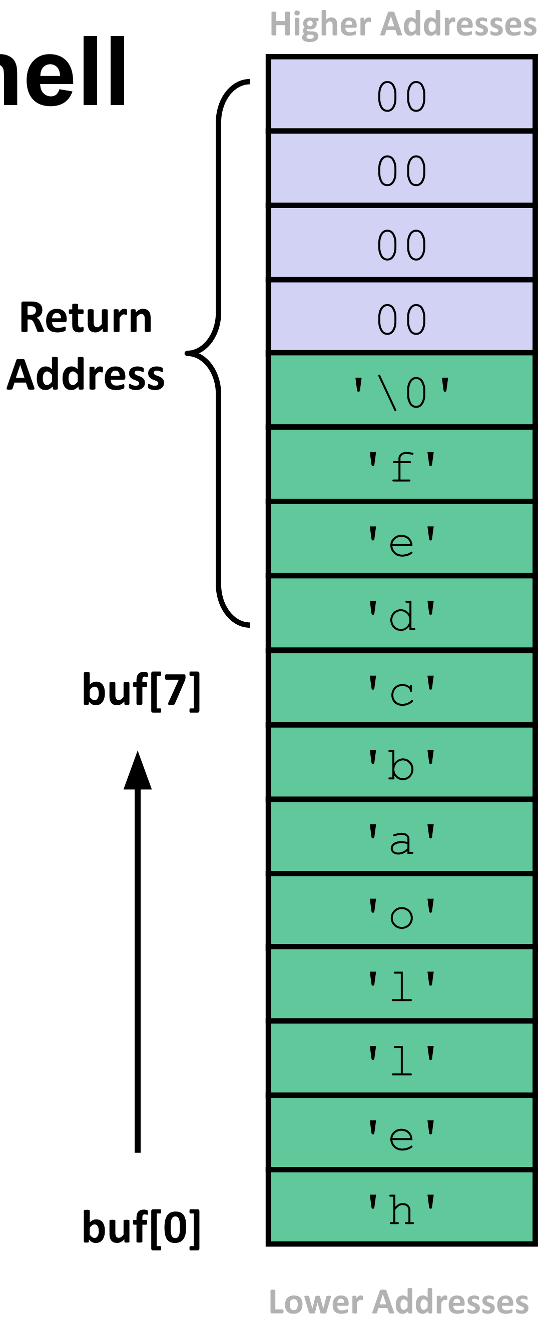


Buffer Overflow in a Nutshell

- 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! ☹️



How are you feeling so far?

Buffer Overflow in a Nutshell

- Buffer overflows on the stack can overwrite "interesting" data
  - Attackers just choose the right inputs
- Simplest form (a.k.a. called "stack smashing")
  - Unchecked length on string input into bounded array causes overwriting of stack data
  - 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

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;
}
```

  - 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);
}
```

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

```
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 Assembly (buf-nsp)

echo:

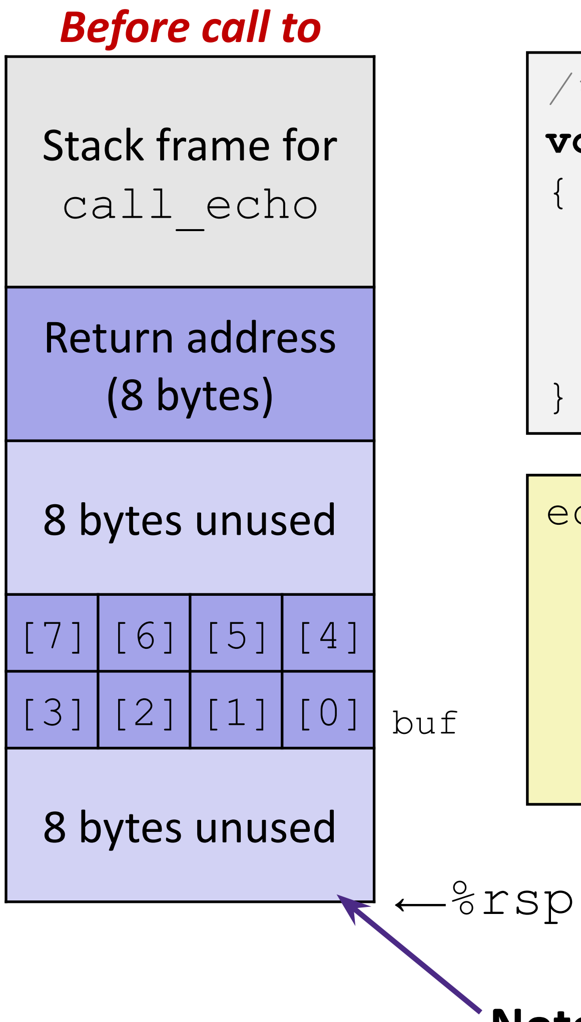
```
0000000000400597 <echo>:
  400597:   48 83 ec 18             sub    $0x18,%rsp
    ...                              ... calls printf ...
  4005aa:   48 8d 7c 24 08          lea    0x8(%rsp),%rdi
  4005af:   e8 d6 fe ff ff          callq  400480 <gets@plt>
  4005b4:   48 89 7c 24 08          lea    0x8(%rsp),%rdi
  4005b9:   e8 b2 fe ff ff          callq  4004a0 <puts@plt>
  4005be:   48 83 c4 18             add    $0x18,%rsp
  4005c2:   c3                      retq
```

call_echo:

```
00000000004005c3 <call_echo>:
  4005c3:   48 83 ec 08             sub    $0x8,%rsp
  4005c7:   b8 00 00 00 00          mov    $0x0,%eax
  4005cc:   e8 c6 ff ff ff          callq  400597 <echo>
  4005d1:   48 83 c4 08             add    $0x8,%rsp
  4005d5:   c3                      retq
```

return address

Buffer Overflow Stack



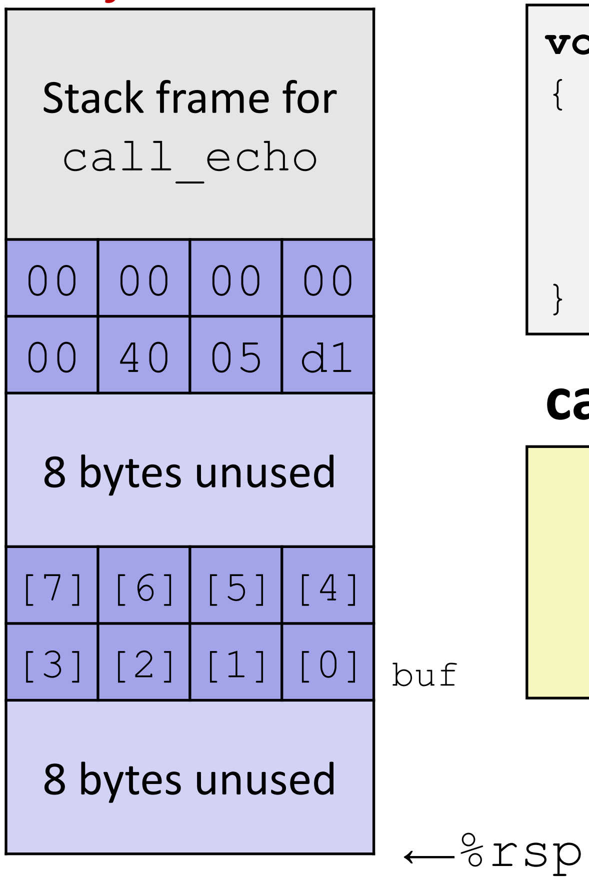
```
/* Echo Line */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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

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

Buffer Overflow Example

Before call to



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

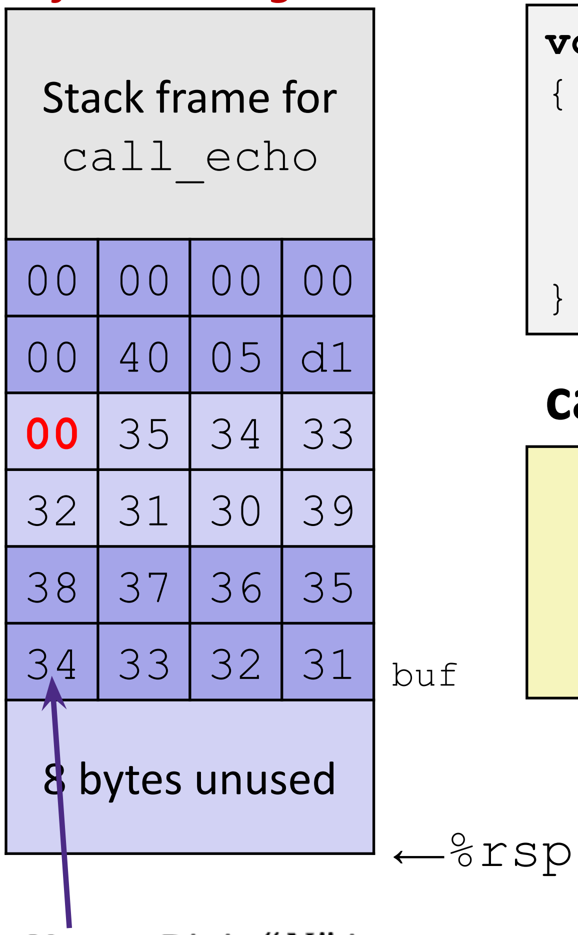
```
echo:
  subq   $24, %rsp
   ...
  leaq   8(%rsp), %rdi
  call   gets
   ...
```

call_echo:

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

Buffer Overflow Ex #1

After call to gets



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

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

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

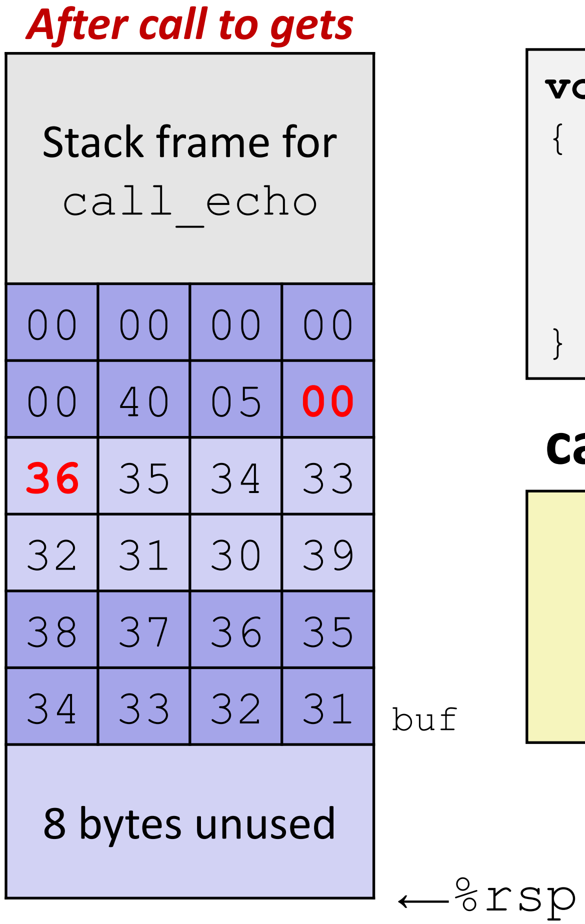
call_echo:

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

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

Overflowed buffer, but did not corrupt state

Buffer Overflow Ex #2



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

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

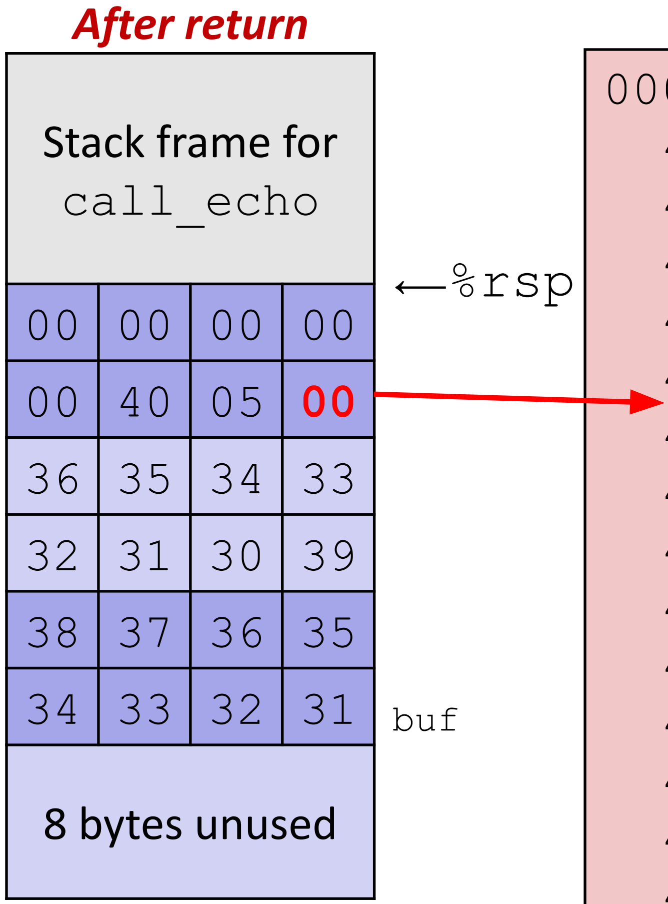
call_echo:

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

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

Overflowed buffer and corrupted return pointer

Buffer Overflow Ex #2 Explained



```
00000000004004f0 <deregister_tm_clones>:
  4004f0:  push   %rbp
  4004f1:  mov    $0x601040,%eax
  4004f6:  cmp    $0x601040,%rax
  4004fc:  mov    %rsp,%rbp
  4004ff:  je     400518
  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
```

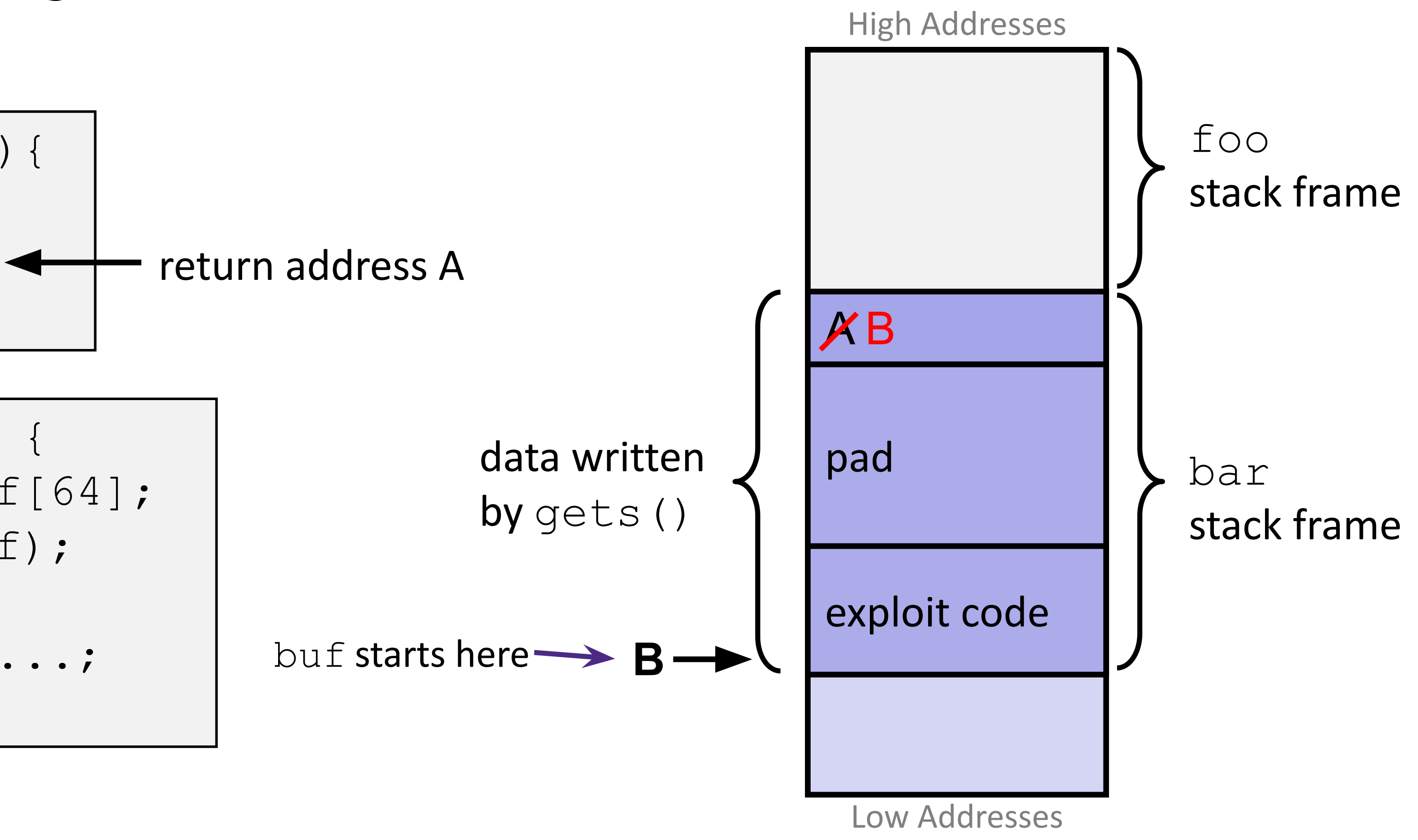
"Returns" to a byte that is not the beginning of an instruction, so program signals `SIGILL, Illegal instruction`

How do you feel about overwriting return addresses?

Buffer Overflow: Code Injection Attacks

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

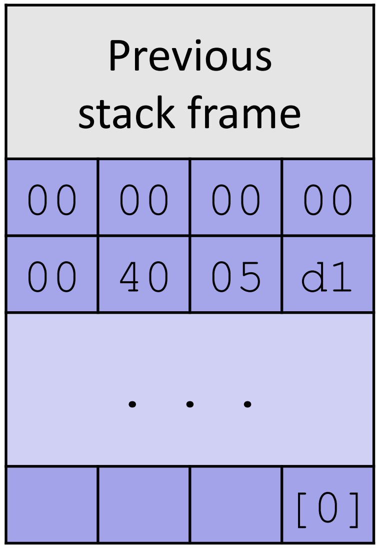
```
int bar() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



- 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

Checking in!

- `vulnerable` 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 0x00 00 7f ff CA FE F0 0D?
  - (This is a stack address)



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

🐶 27
🐱 30
🐑 51
🦄 54
🥶 Help!

Exploits Based on Buffer Overflows

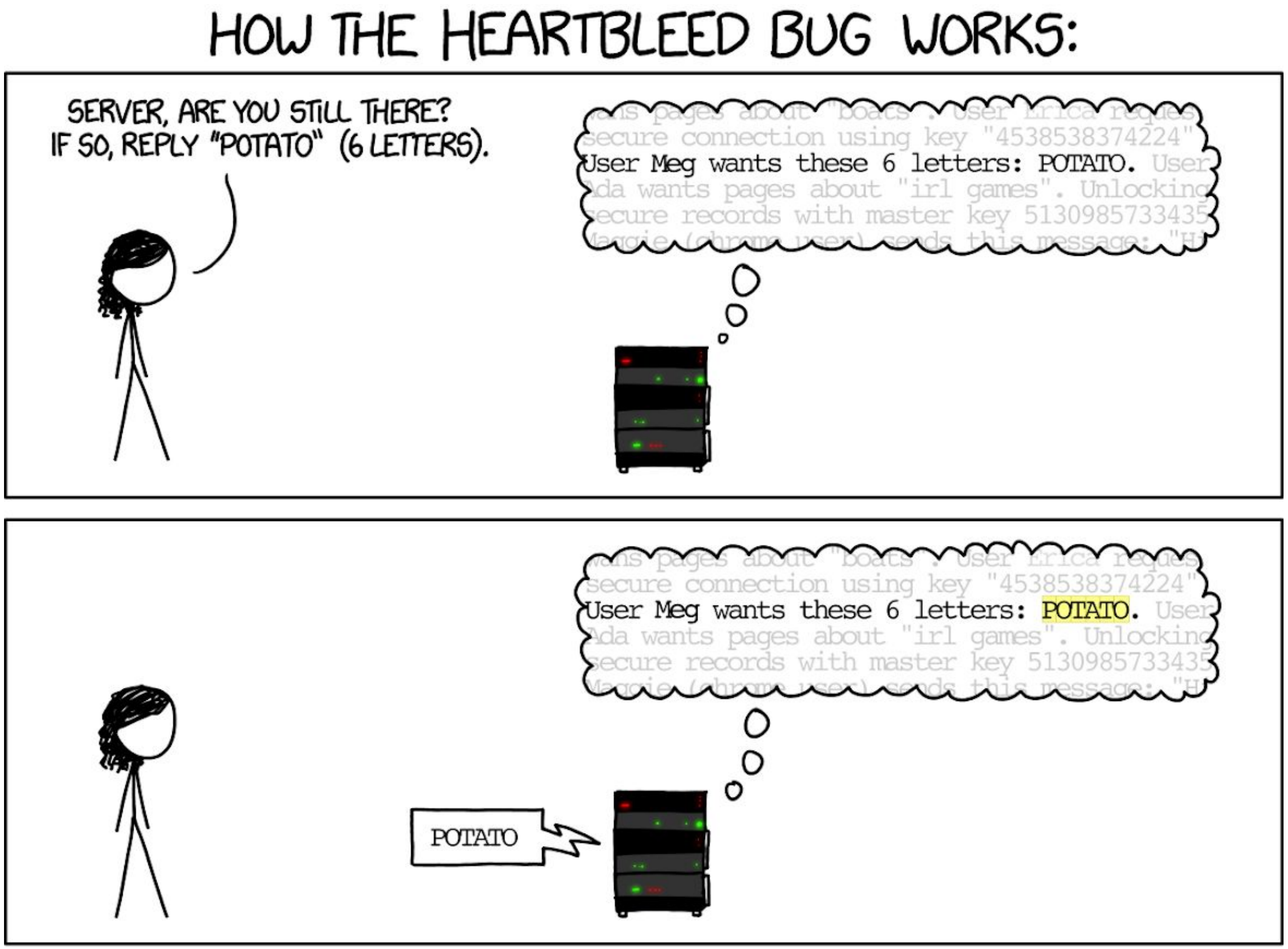
Buffer overflow bugs can allow attackers to execute arbitrary code on victim machines

- Distressingly common in real programs
  - Programmers keep making the same mistakes ☹
  - Recent measures make these attacks 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

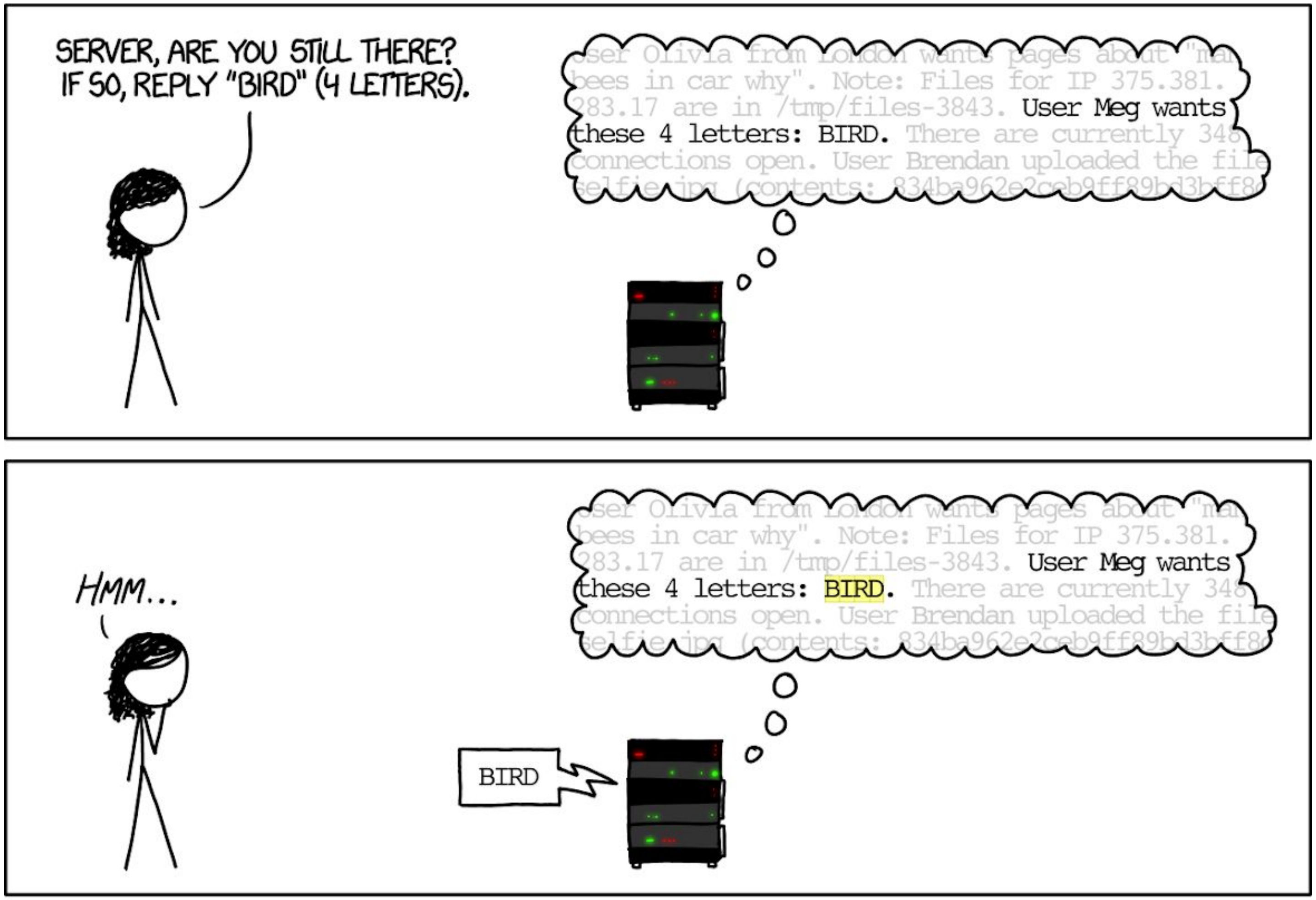
Ex: the original Internet worm (1988)

- 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 Internet)
    - see June 1989 article in *Comm. of the ACM*
  - The author (Robert Morris*) was prosecuted…

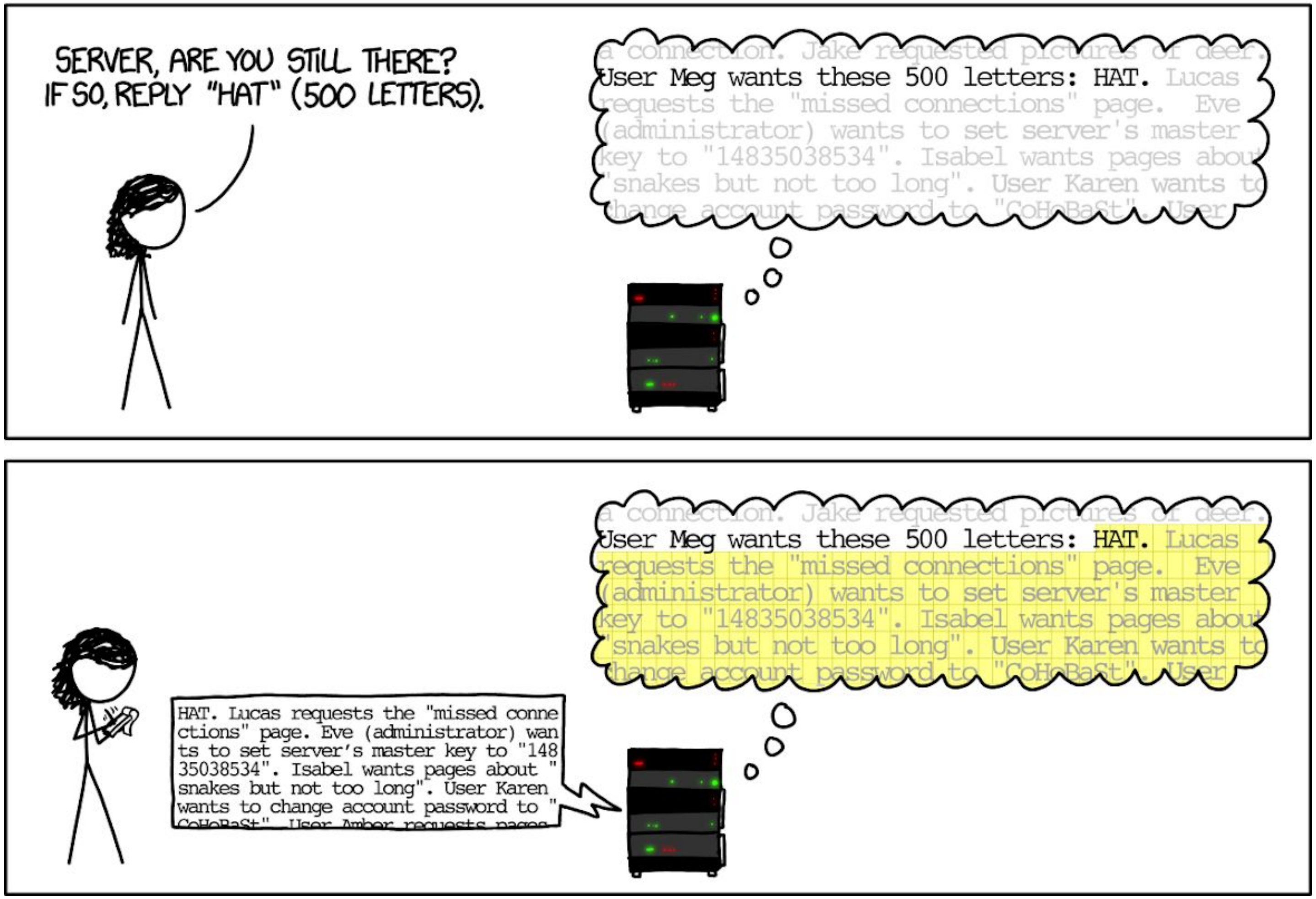
Example: Heartbleed


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
wants pages about "boats". User Erica requests secure connection using key "4538538374224". User Meg wants these 6 letters: POTATO. User Ada wants pages about "irl games". Unlocking secure records with master key 5130985733435. Maggie (chrome user) sends this message: "H
POTATO

Example: Heartbleed

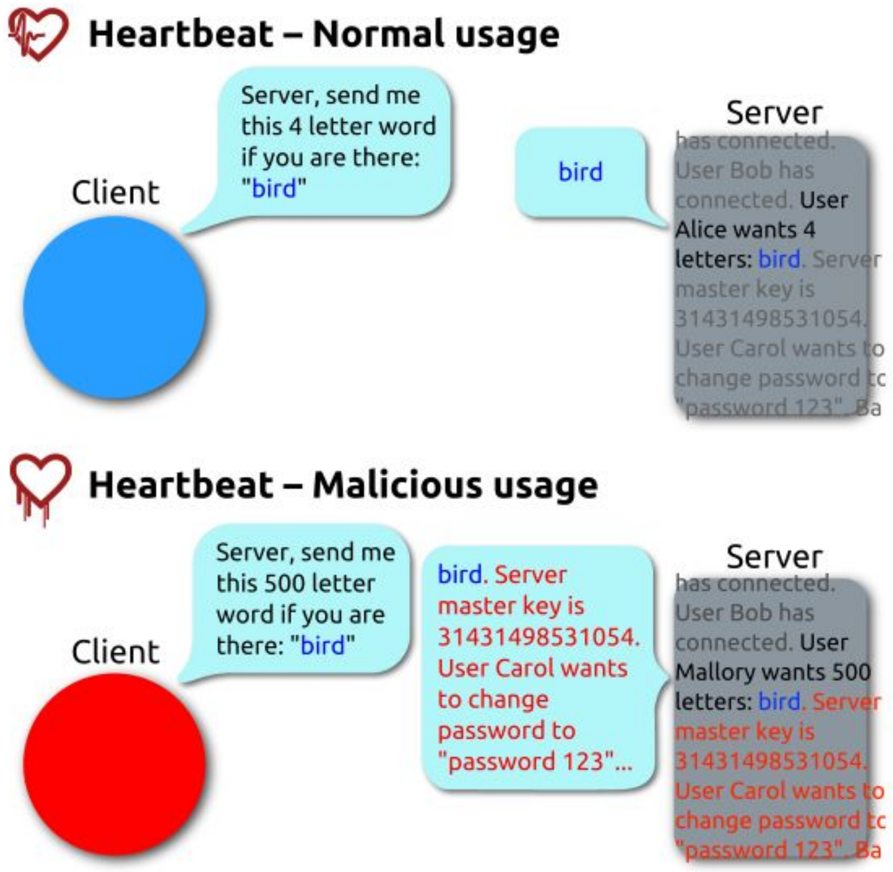

SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
User Olivia from London wants pages about "man bees in car why". Note: Files for IP 375.381.283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 348 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e2ceb9ff89bd3bff8
HMM...
User Olivia from London wants pages about "man bees in car why". Note: Files for IP 375.381.283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 348 connections open. User Brendan uploaded the file selfie.jpg (contents: 834ba962e2ceb9ff89bd3bff8
BIRD

Example: Heartbleed


SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
a connection. Jake requested pictures of deer. User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User Amber requests pages

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, ...


Heartbeat – Normal usage
Client
Server, send me this 4 letter word if you are there: "bird"
bird
Server
has connected. User Bob has connected. User Alice wants 4 letters: bird. Server master key is 31431498531054. User Carol wants to change password to "password 123". Ba
Heartbeat – Malicious usage
Client
Server, send me this 500 letter word if you are there: "bird"
bird. Server master key is 31431498531054. User Carol wants to change password to "password 123"...
Server
has connected. User Bob has connected. User Mallory wants 500 letters: bird. Server master key is 31431498531054. User Carol wants to change password to "password 123". Ba

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

Hacking Cars

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

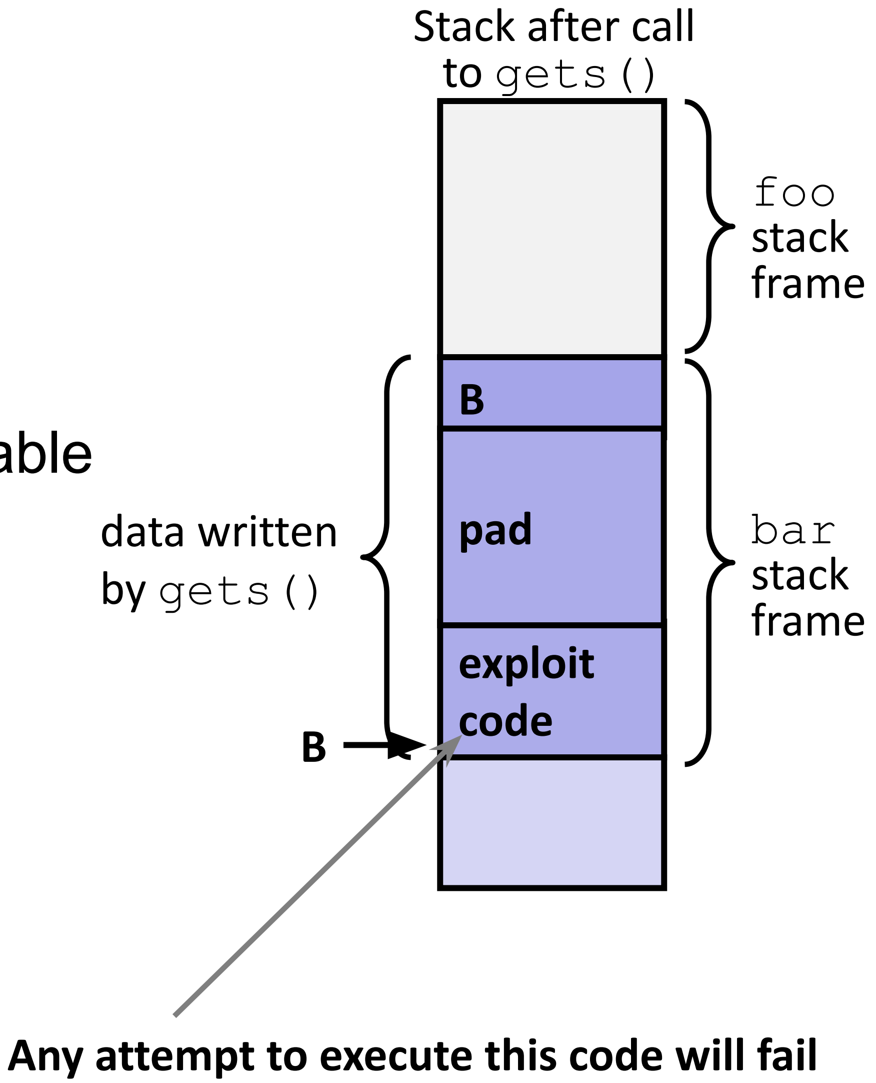
How do we feel about exploiting buffers?

Dealing with buffer overflow attacks

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

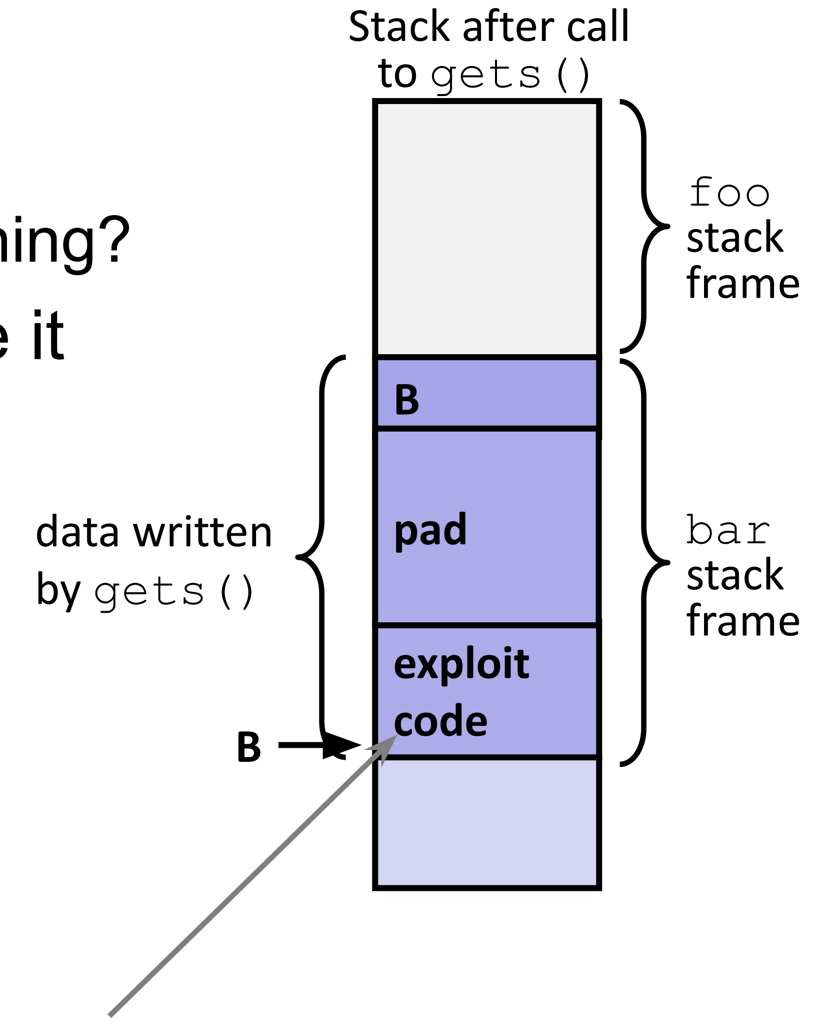
1) System-Level Protections

- **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

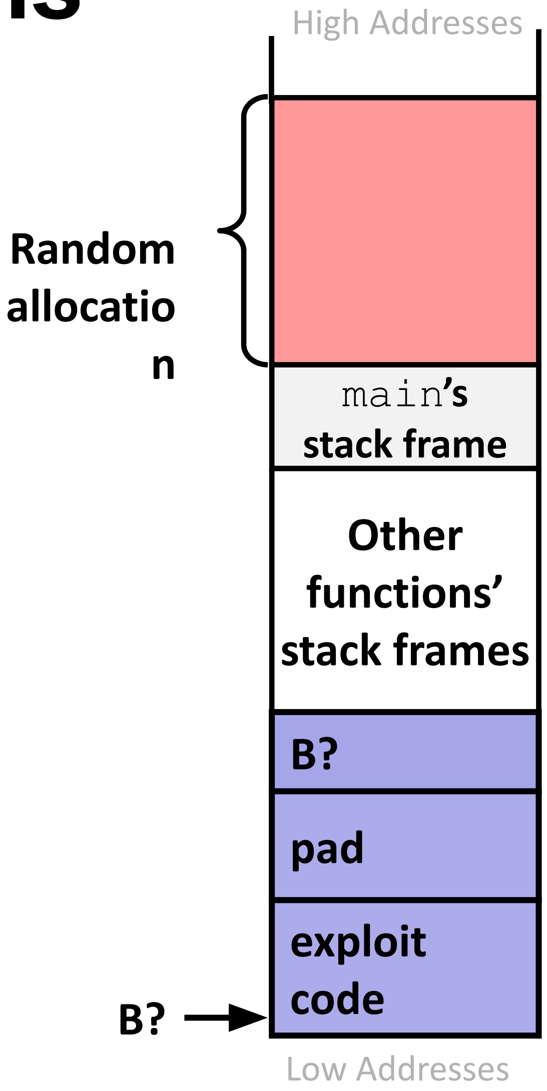
- **Non-executable code segments**
  - Wait, doesn't this fix everything?
- Works, but can't always use it
- Many embedded devices *do not* have this protection
  - Cars
  - Smart homes
  - Pacemakers
- Some exploits still work!
  - Return-oriented programming
  - Return to libc attack



Any attempt to execute this code will fail

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`
    - `0x7fffa0175afc`
  - Stack repositioned when program executes



2) Avoid Vulnerabilities in Code

```
/* 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 `%ns` where `n` is a suitable integer

2) 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 (buf)

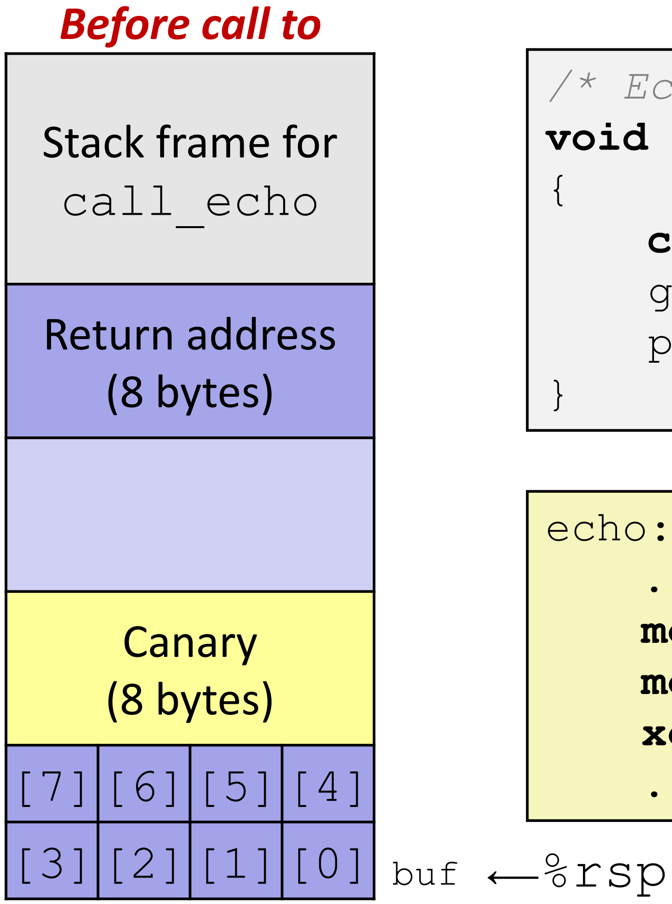
This is extra (non-testable) material

echo:

```
400607:   sub    $0x18,%rsp
40060b:   mov    %fs:0x28,%rax
400614:   mov    %rax,0x8(%rsp)
400619:   xor    %eax,%eax
 ...      ... 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
40063a:   xor    %fs:0x28,%rax
400643:   jne    40064a <echo+0x43>
400645:   add    $0x18,%rsp
400649:   retq
40064a:   callq  4004f0 <__stack_chk_fail@plt>
```

Setting Up Canary

This is extra (non-testable) material



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

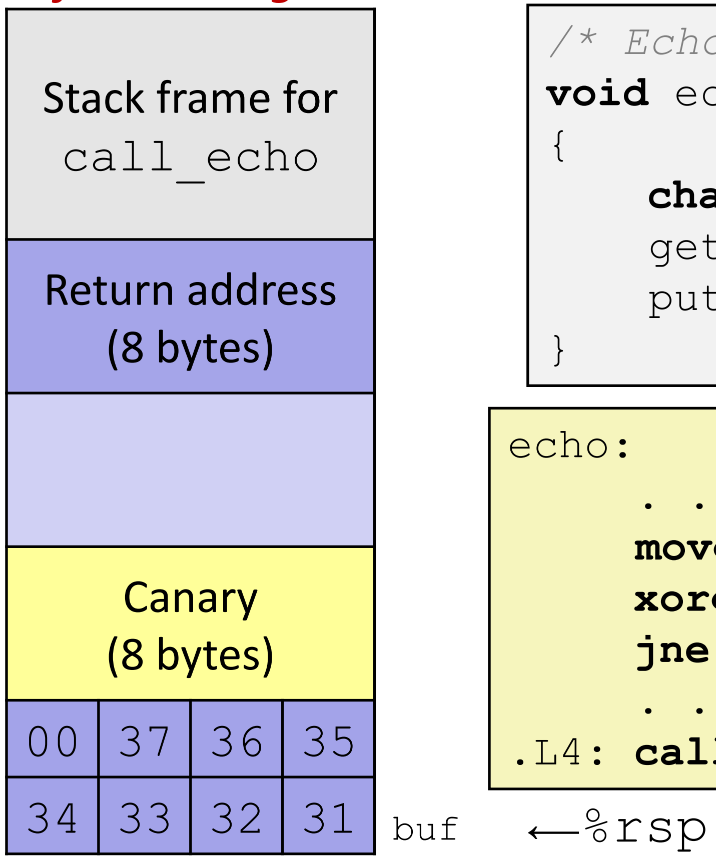
Segment register *(don't worry about it)*

```
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

- 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
- **C isn’t accessible! Though, neither is programming, but C is particularly bad.**

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 😉
  - 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

How do we feel about mitigating buffers?

Now, fun stuff!

(though buffer overflows are interesting)

Last Time

Defining Usability

- If I write C and forget to bounds-check arrays, whose fault is it?
    - Mine? "I should have known better"
    - K&R's? "They should have known better"
- Blame tends to be an individualistic focus
    - Sometimes helpful, i.e. malicious criminal cases
    - Sometimes less helpful, i.e. racism
- **Def: Use, without causing harm, independent of physical or cognitive capabilities**
    - Inaccessibility is a structural issue, not a personal one

Accessibility & CS

- CS, programming, in general, is inaccessible
  - Lots of cognitive requirements
- Many consumer technologies aren’t accessible
  - Few designed with accessibility in mine
- CS tech isn’t accessible either!
  - Tendency to over-emphasize individual → ideological foundations of CS
  - Many structures aren’t usable, “just don’t use them” isn’t always an option
  - Also, switches emphasis back to individual, away from structural inequity

Agency, Support, Accessibility

Agency, Support, Accessibility

- **Agency:** Capacity for exploration and self-expression (*fulfilling wants*)
- **Support:** Ability to receive assistance, receive care (*fulfilling needs)*
- **Accessibility**: Usability, based on needed support and desired agency
  - What agency can be expressed? What support is given? Who can use this technology?

- *A lens to view technologies*
- Who's designing tech? What assumptions are being made in that design?

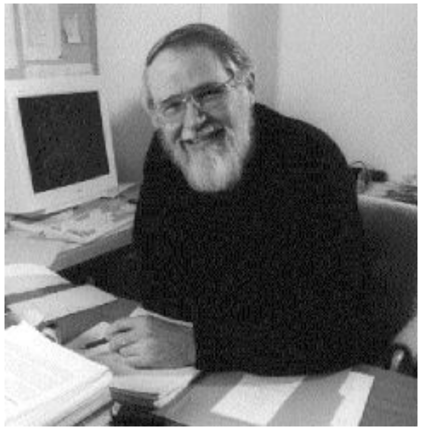
Agency, Support, Access: Assembly

- **Agency:** Basically, anything that can be computed, limited by ISA definitions
- **Support:** None! Everything from the ground up, the only support is a processor manual

- **Accessibility:** Building everything from the ground up requires a lot of assembly, and a high potential for bugs. Processor manuals are a lot to read and understand (Intel's is ~8000 pages!).
  - *Low support (basically none), high agency*

C language (1978)

- Created in 1972, “standardized” in 1978
  - Goal of writing Unix (precursor to Linux/OSX)
- Explicit Goals:
  - Make a *faster* programming language than B
  - Make a simple, minimalistic language; easy to learn
- Non-Goal: make C accessible for learners



Agency, Support, Access: C

- **Agency:** Matches assembly, anything that can be computed, within ISA limitations
  - Portability means meeting requirements of many ISAs
- **Support:** Basic programming constructs, but nothing more. You're assumed to not need support beyond loops/branches/functions
- **Access:** There's library functions, but most things need to be built. The "ground" is higher than assembly, but not by much. No processor manuals, but lots of things can go wrong.
  - *Low support, high agency*

C: Arrays and Buffers

- Arrays are an opportunity to express agency!
  - Technically available in assembly
  - C gives a high-agency interface, mimicking the assembly level
- There's no support with arrays!
  - Compared with assembly, a more powerful tool, without more support
  - *Lack of support means more opportunities for harm*
  - *There's more to keep track of! Higher potential for vulnerabilities, like buffer overflows*

At the time

- Computing was still specialized
  - Programming wasn't considered a "general task"
- C improves accessibility over assembly, and agency over B
  - Generally, programming languages aim to improve on agency/support of prior language
  - C isn't an exception

Java

- No signed/unsigned distinction!
- Out of bounds exceptions
- No buffer overflows!
- “Write once, run anywhere”

- Aims to improve upon C’s lack of support/OOP, while maintaining agency expected from programming languages

Agency, Support, Access: Java

- **Agency:** You're further from the hardware, but still turing complete. Generally slower than C
- **Support:** Exceptions, array bounds checks, much more support than C, portability built in, but lots of security vulnerabilities

- **Access:** Lots of libraries, pre-built code, more support than C. Further up from C/Assembly in the house of computing.
  - *Compared with C: Higher support, lower agency*

We’ve been talking about ideologies all quarter, what about those?

“We shape our tools, and thereafter, our tools shape us”

1967

“Reification”, if you want a single word. To make the abstract concrete.

Computing is a tool, but a tool built by a distinctly non-neutral society!
We’ve always had values!

Technology:

Knowledge, encoded into artifacts, tools

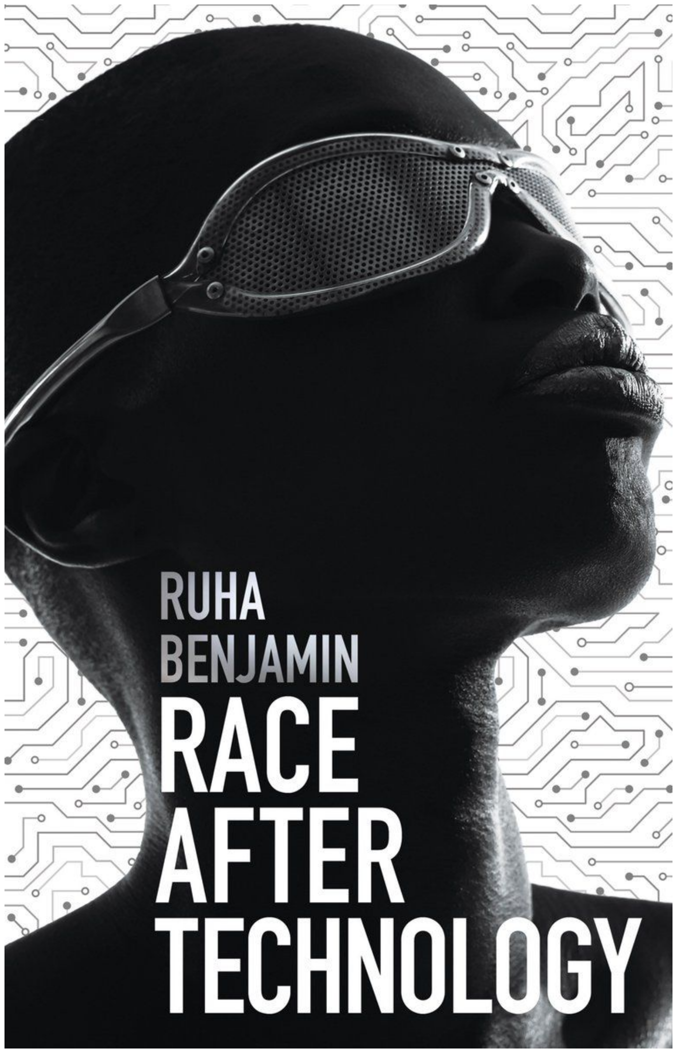
"Human toolmaking is not limited to the stone instruments of our early ancestors or to the sleek gadgets produced by the modern tech industry. Human cultures also create symbolic devices that structure society.

"Human toolmaking is not limited to the stone instruments of our early ancestors or to the sleek gadgets produced by the modern tech industry. Human cultures also create symbolic devices that structure society. **Race, to be sure, is one of our most powerful tools – developed over hundreds of years, varying across time and place, codified in law and refined through custom, and, tragically, still considered by many people to reflect immutable differences between groups."**

Ruha Benjamin

Return to Ideology

- Not every “technology” is “technical”!
- **What does it mean to “use” social or symbolic technologies?**



ASA: Race

- **Agency:** Varying, based on position/power
- **Support:** Varying, based on position/power
- **Access:** Extremely accessible, access to “race as technology” continues to be guaranteed by law
  - Both as oppressed and oppressors!

- Support for whom? Agency for whom?

Assembly, C, Java:

Support for whom?
Agency for whom?
Access for whom?

This extends to other tech...

Racist Technologies

- We can say a technology is *racist* when agency, support, and access provided by that technology are contingent upon one's racial status

- See: every facial recognition system, motion detection that assumes whiteness, software-aided photography, cameras, prison sentencing algorithms, policing in general, healthcare, especially around neurodiversity and cognition, education
  - *If a system has outcomes that differ by race, we should call that system racist*

Our path this quarter

- Historical context outside computing to surface ideologies within computing
- Critically reading *texts* for ideologies
- Critically reading *tech* for ideologies

- Comparing ideological priorities

Broad Ideologies

- Neoliberalism
  - Agency’s really important! Access/Support are “implementation details”
- Communism/Marxism
  - **“From each according to his ability, to each according to his needs” - Marx**
  - Support’s really important! Access/Agency are “implementation details”

Just one framework!

There are many others, but I like this one.

Critical Reading

- **For texts:**
  - What's being assumed? What's being prioritized? What's emphasized? What's given space?
- **For tech:**
  - What's prioritized, between agency, support, and access?
  - What assumptions are made around agency, support, access?
  - ASA for who? For everyone? Or, only for those with privilege? Only for those *seen as legitimate*?
    - Do you need a "Computer Scientist" card?

Who’s welcome in the house of computing?

Who’s welcome?

- What’s given to newcomers?
- When we hear about agency, for whom?
- When we hear about support, for whom?
- When we hear about access, for whom?

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+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-offset` is less reliable
  - For example, if you need save a variable for a function call, pushing it onto the stack changes `%rsp`

Hacking DNA Sequencing Tech

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

Computer Security and

Paul G. Allen School of Computer Scienc

There has been rapid improvement in the cost an
decade, the cost to sequence a human genome ha
was made possible by faster, massively parallel pr
hundreds of millions of DNA strands simultaneou
ranging from personalized medicine, ancestry, an

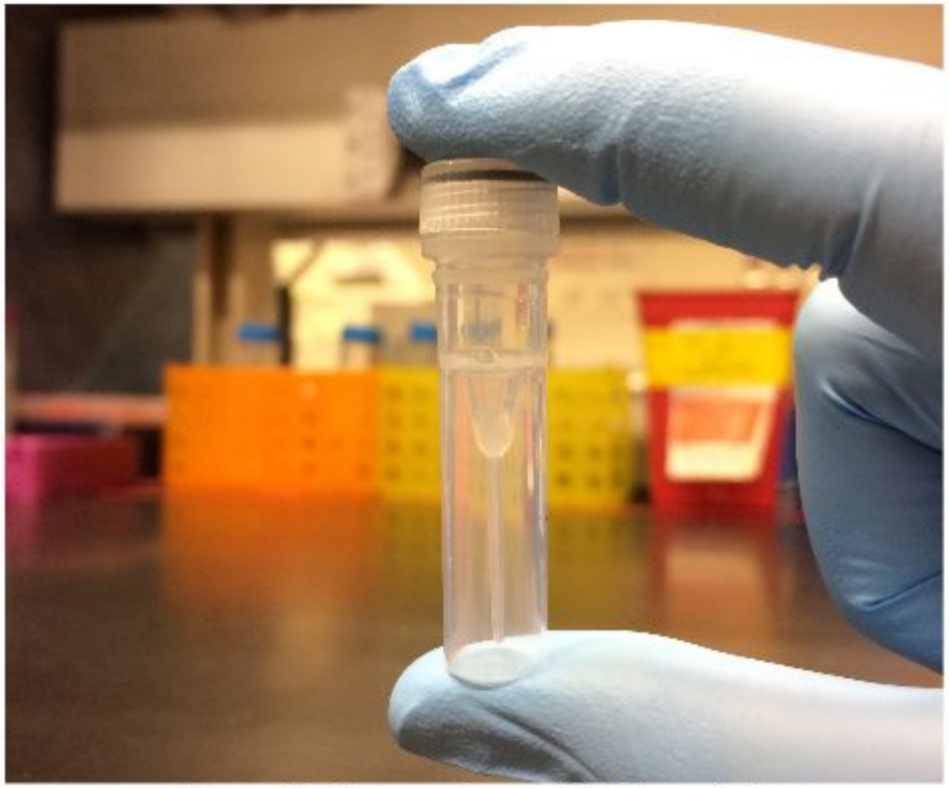
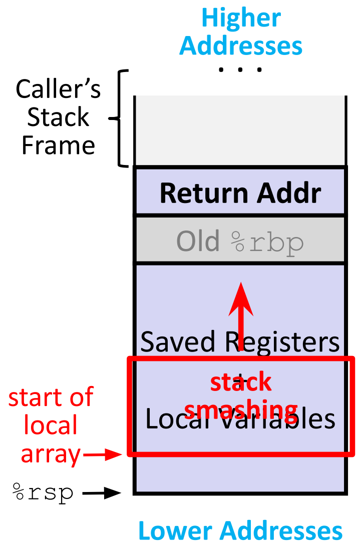


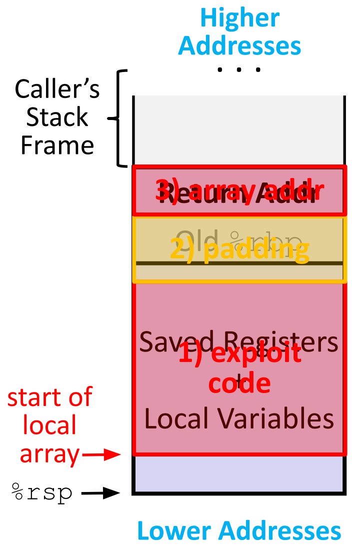
Figure 1: Our synthesized DNA exploit

Where Is It?

Variable/Label	Section of Memory
`big_array`	
`global`	
`huge_array`	
`local`	
`main`	
`p1`	
`*p1`	
`useless`	

Notes Diagrams


Higher Addresses
. . .
Caller's Stack Frame
Return Addr
Old %rbp
Saved Registers
stack smashing
Local Variables
start of local array
%rsp
Lower Addresses


Higher Addresses
. . .
Caller's Stack Frame
Return Addr
3) array addr
Old %rbp
2) padding
Saved Registers
1) exploit code
Local Variables
start of local array
%rsp
Lower Addresses