

Buffer Overflows

CSE 351 Autumn 2017

Instructor:

Justin Hsia

Teaching Assistants:

Lucas Wotton

Michael Zhang

Parker DeWilde

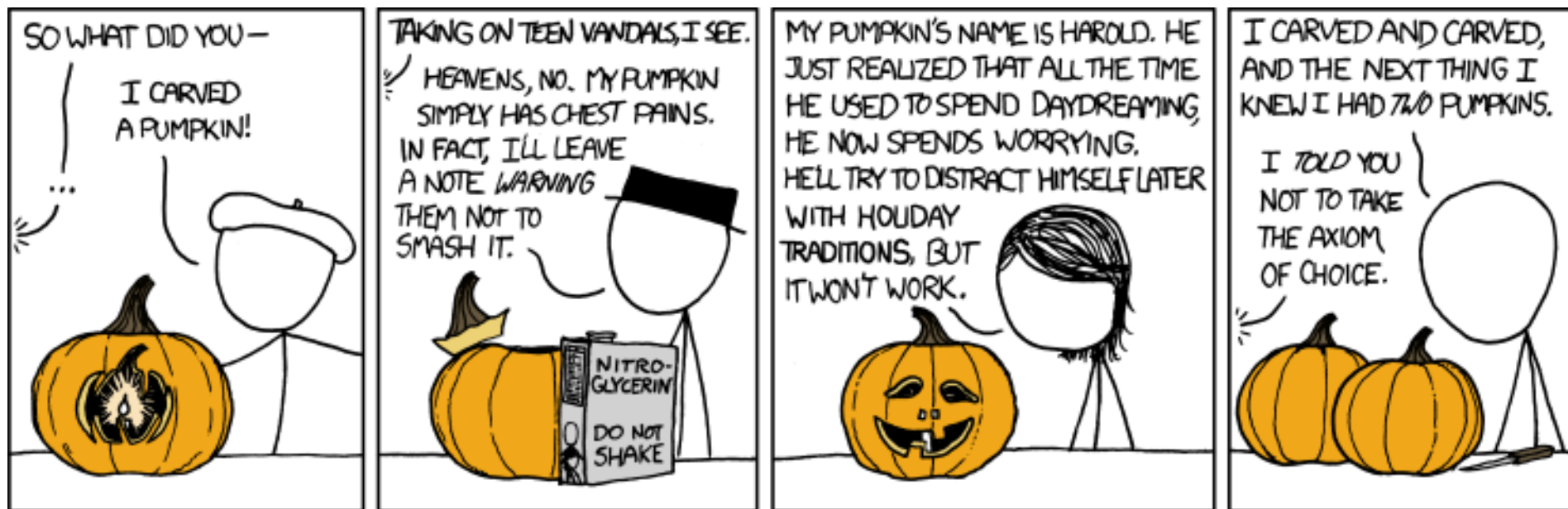
Ryan Wong

Sam Gehman

Sam Wolfson

Savanna Yee

Vinny Palaniappan



Administrivia

- ❖ Mid-quarter survey due tomorrow (11/2)
- ❖ Homework 3 due Friday (11/3)
- ❖ Lab 3 released today, due next Thursday (11/9)

- ❖ Midterm grades (out of 50) to be released by Saturday
 - Solutions posted on website
 - Rubric and grades will be found on Gradescope
 - Regrade requests will be open for a short time after grade release

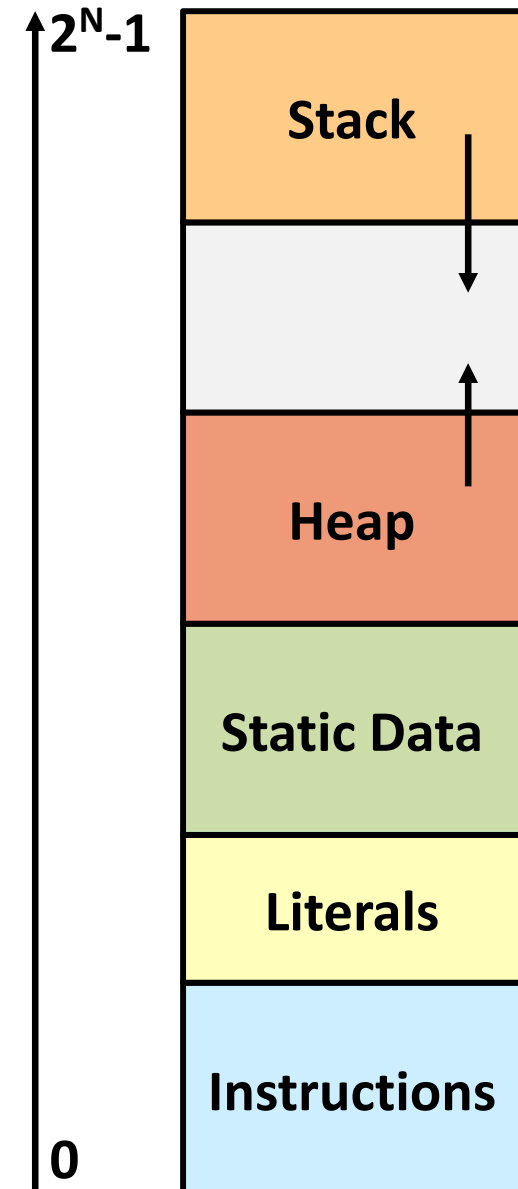
Buffer Overflows

- ❖ 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
 - 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

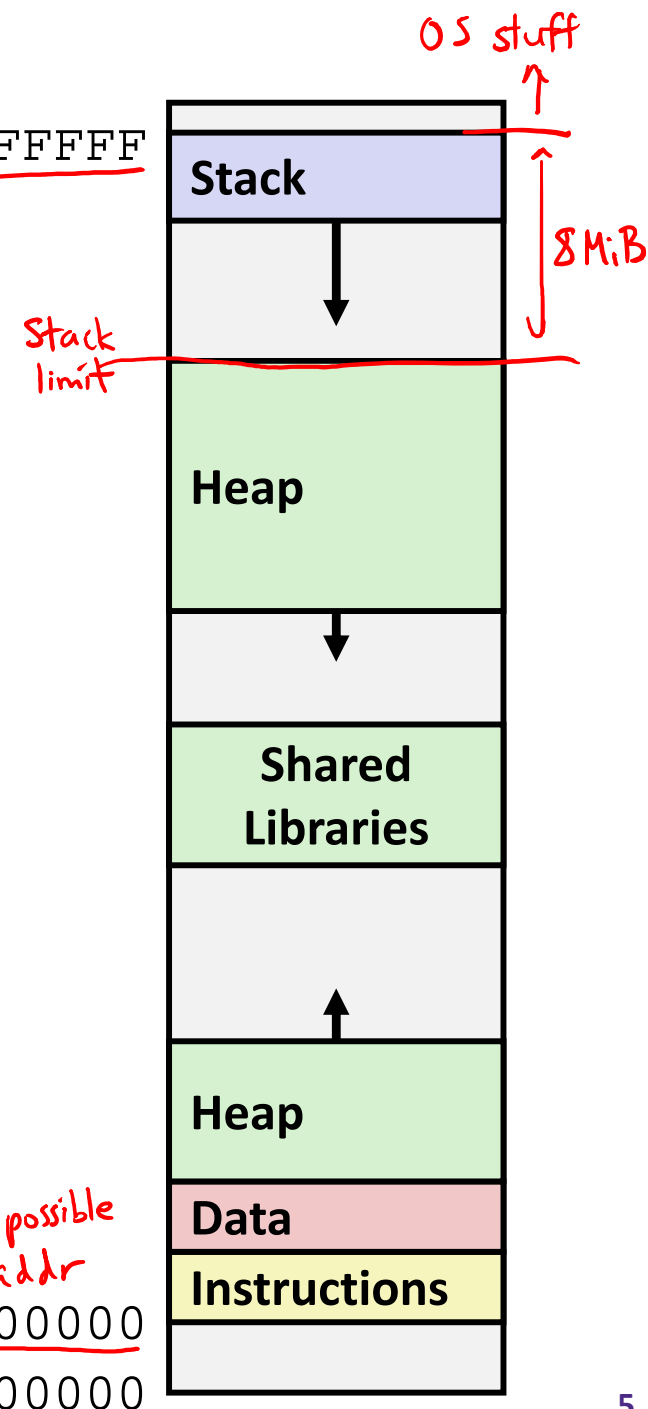


x86-64 Linux Memory Layout

0x00007FFFFFFFFFFF
 48-bits

- ❖ 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



Hex Address



lowest possible
instr addr
0x400000
 0x000000

not drawn to scale

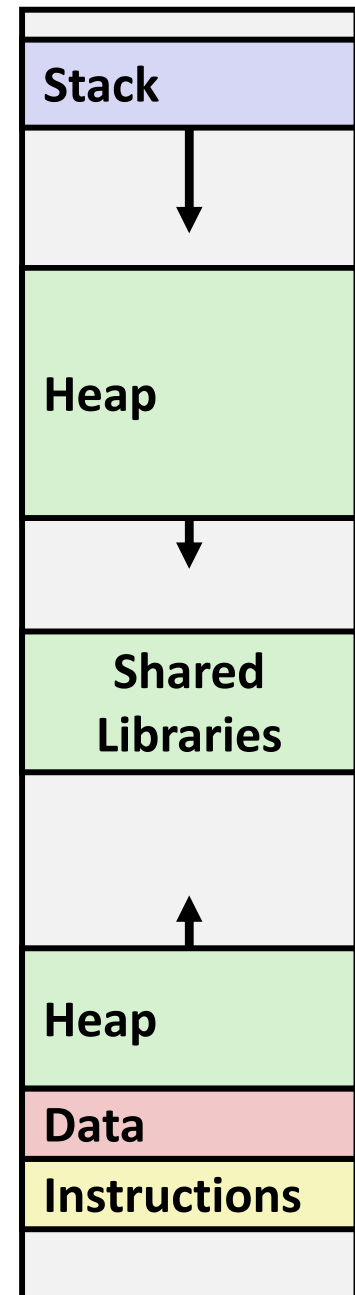
Memory Allocation Example

```

char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */
                                } global (Data)

int global = 0;

int useless() { return 0; }
                                ↗ labels in code
int main()
{
    void *p1, *p2, *p3, *p4; } local (stack)
    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 ... */
}
                                ↖ dynamically-allocated memory
                                (Heap)
    
```



Where does everything go?
p1 → stack address
**p1 → heap address*

not drawn to scale

Memory Allocation Example

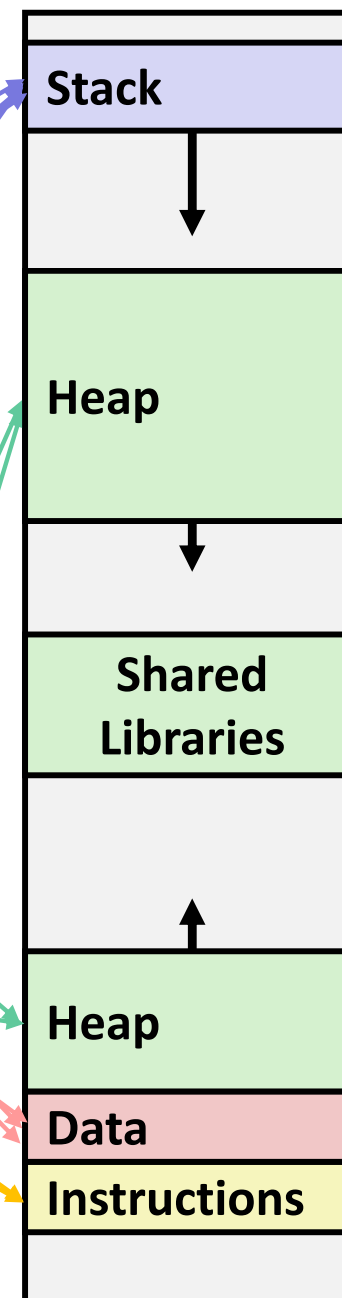
```

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

int global = 0;

int useless() { return 0; }

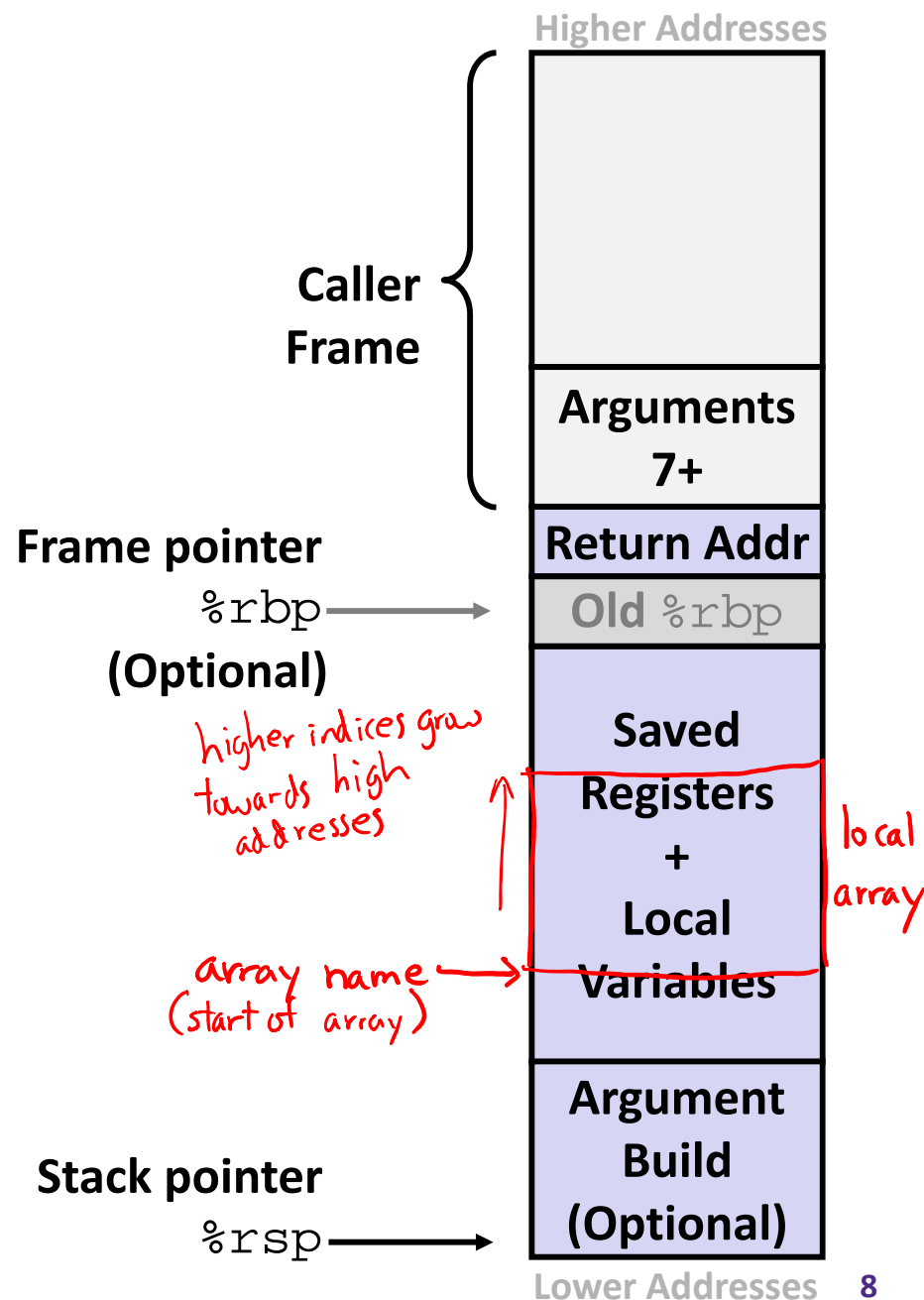
int main()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8); /* 256 B */
    p3 = malloc(1L << 32); /* 4 GB */
    p4 = malloc(1L << 8); /* 256 B */
    /* Some print statements ... */
}
    
```



Where does everything go?

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)
 - Saved register context (when reusing registers)
 - Local variables (if can't be kept in registers)
 - "Argument build" area (If callee needs to call another function -parameters for function about to call, if needed)



Buffer Overflow in a Nutshell

- ❖ 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
- ❖ 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 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 is (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;
}
```

end of file (points to EOF in while condition)

newline (points to '\n' in while condition)

reads character from input stream (points to getchar())

pointer to start
of an array (*don't know
size!*)

same as:

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

- What could go wrong in this code?

String Library Code

❖ 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
↖ 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);  
    puts(buf);  
}
```

input buffer (with arrow pointing to buf[8])
read input into buffer (with arrow pointing to gets(buf))
print output from buffer (with arrow pointing to puts(buf))

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

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

```
unix> ./buf-nsp  
Enter string: 123456789012345678901234  
Segmentation Fault
```

Buffer Overflow Disassembly (buf-nsf)

echo:

00000000004005c6 <echo>:	
4005c6: 48 83 ec 18	sub \$0x18,%rsp
...	... calls printf ...
4005d9: 48 89 e7	mov %rsp,%rdi
4005dc: e8 dd fe ff ff	callq 4004c0 <gets@plt>
4005e1: 48 89 e7	mov %rsp,%rdi
4005e4: e8 95 fe ff ff	callq 400480 <puts@plt>
4005e9: 48 83 c4 18	add \$0x18,%rsp
4005ed: c3	retq

Handwritten annotations:
 - Red arrow pointing to \$0x18 with "24"
 - Red circle around "sub \$0x18,%rsp" with "Compiler choice" pointing to it

call_echo:

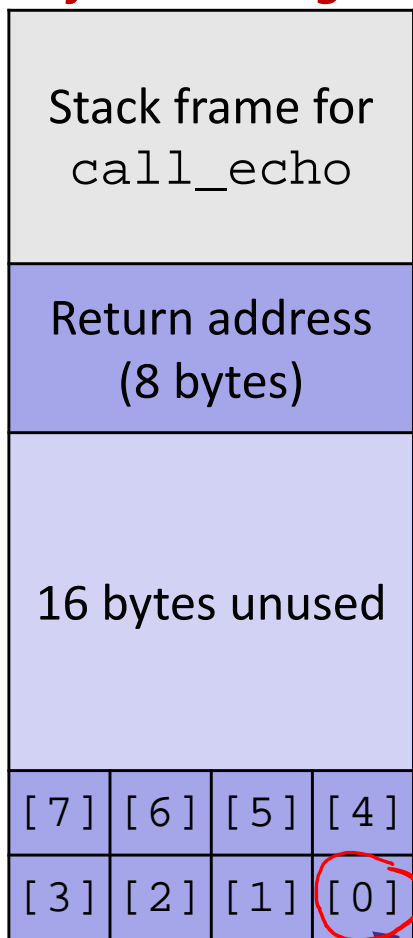
00000000004005ee <call_echo>:	
4005ee: 48 83 ec 08	sub \$0x8,%rsp
4005f2: b8 00 00 00 00	mov \$0x0,%eax
4005f7: e8 ca ff ff ff	callq 4005c6 <echo>
4005fc: 48 83 c4 08	add \$0x8,%rsp
400600: c3	retq

Handwritten annotations:
 - Red box around 4005fc
 - Red underline under callq 4005c6 <echo>
 - Purple arrow pointing from 4005fc to the text "return address placed on stack"

return address placed on stack

Buffer Overflow Stack

Before call to gets



```

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

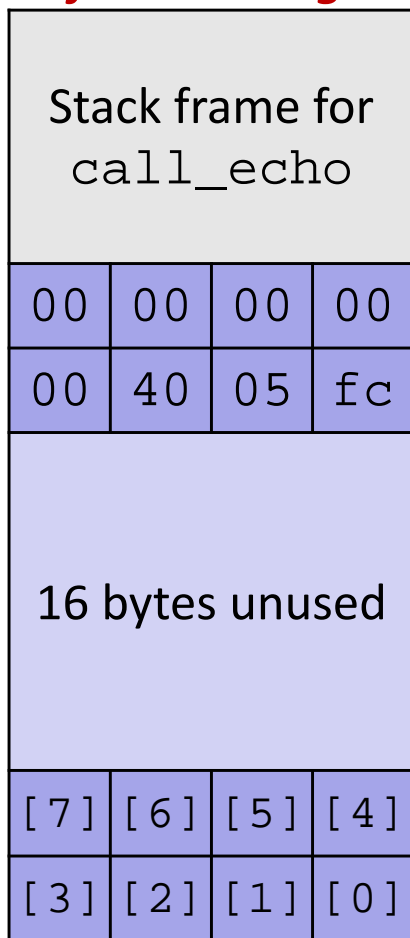
```

echo:
    subq $24, %rsp
    ...
    movq %rsp, %rdi
    call gets
    ...
    
```

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

Buffer Overflow Example

Before call to gets



buf ← %rsp

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

```
echo:
    subq $24, %rsp
    . . .
    movq %rsp, %rdi
    call gets
    . . .
```

call_echo:

```
. . .
4005f7: callq 4005c6 <echo>
4005fc: add $0x8,%rsp
. . .
```


Buffer Overflow Example #1

After call to gets

Stack frame for call_echo			
00	00	00	00
00	40	05	fc
00	33	32	31
30	39	38	37
36	35	34	33
32	31	30	39 ← '9'
38	37	36	35
34	33	32	31 ← '1'

buf ← %rsp

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

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

```
echo:
    subq    $24, %rsp
    . . .
    movq   %rsp, %rdi
    call   gets
    . . .
```

call_echo:

```
. . .
4005f7:    callq   4005c6 <echo>
4005fc:    add     $0x8,%rsp
. . .
```

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

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
34	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

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

```
echo:
    subq $24, %rsp
    ...
    movq %rsp, %rdi
    call gets
    ...
```

call_echo:

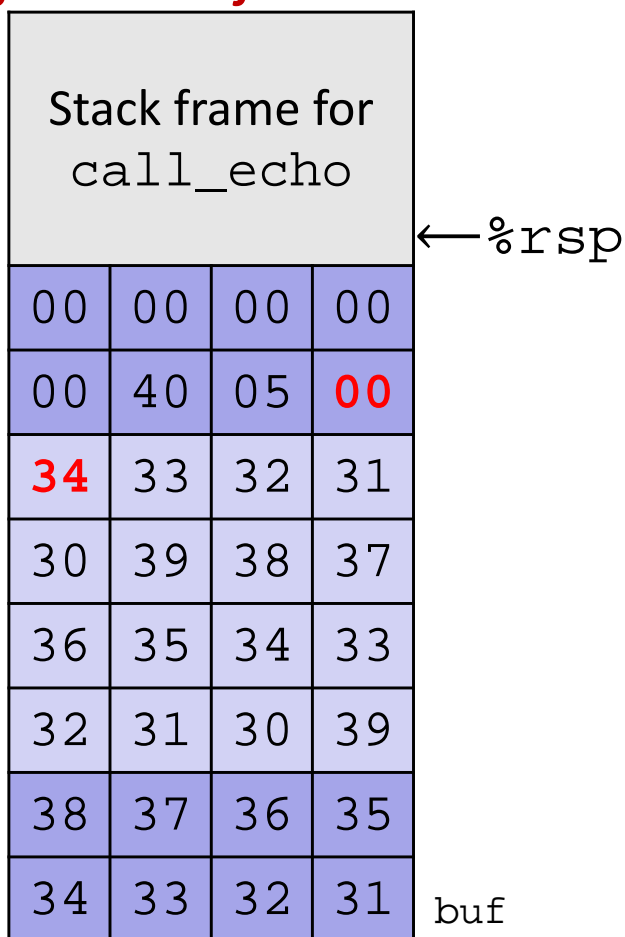
```
. . .
4005f7: callq 4005c8 <echo>
4005fc: add $0x8,%rsp
. . .
```

```
unix> ./buf-nsf
Enter string: 123456789012345678901234
Segmentation Fault
```

Overflowed buffer and corrupted return pointer

Buffer Overflow Example #2 Explained

After return from echo



```

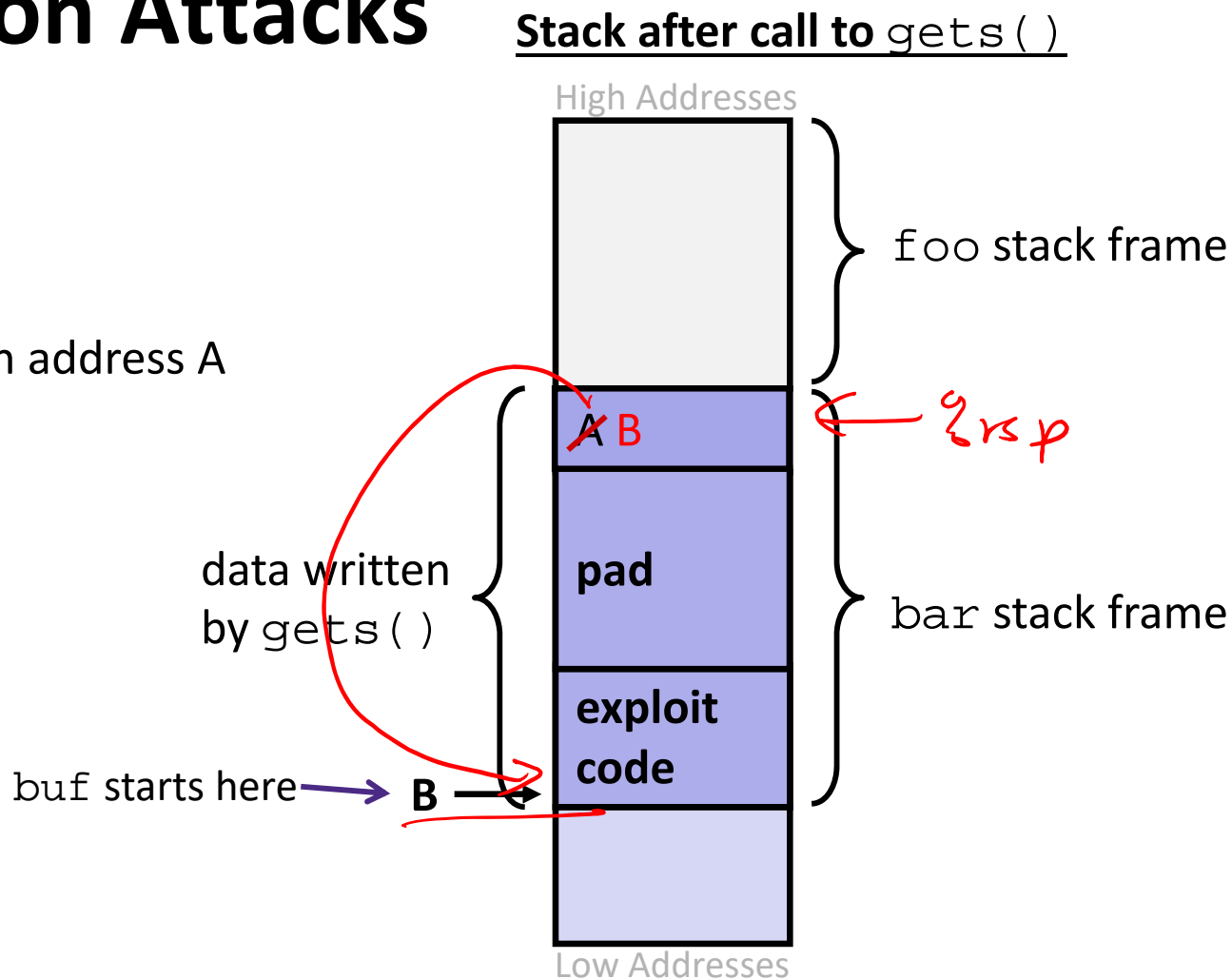
0000000000400500 <deregister_tm_clones>:
400500:  mov    $0x60104f,%eax
400505:  push  %rbp
400506:  sub    $0x601048,%rax
40050c:  cmp    $0xe,%rax
400510:  mov    %rsp,%rbp
400513:  jbe    400530
400515:  mov    $0x0,%eax
40051a:  test  %rax,%rax
40051d:  je     400530
40051f:  pop   %rbp
400520:  mov    $0x601048,%edi
400525:  jmpq  *%rax
400527:  nopw  0x0(%rax,%rax,1)
40052e:  nop
400530:  pop   %rbp
400531:  retq
    
```

“Returns” to unrelated code, but continues!
Eventually segfaults on retq of deregister_tm_clones.

Malicious Use of Buffer Overflow: Code Injection Attacks

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

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

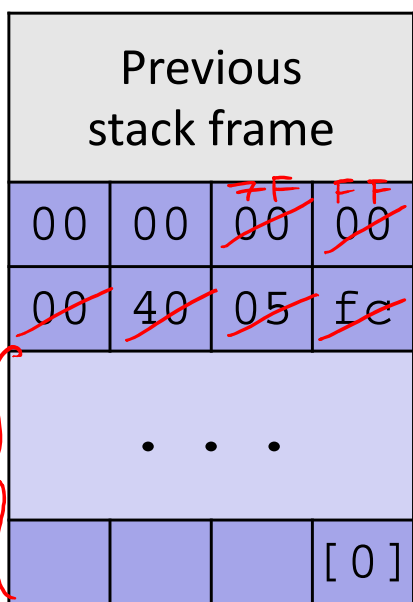


- ❖ Input string contains byte representation of executable code
- ❖ Overwrite return address A with address of buffer B
- ❖ When `bar()` executes `ret`, will jump to exploit code

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 (in Linux)?
 - Vote at <http://PollEv.com/justinh>

0x 00 00 7FFF ?? ?? ?? ??
 always 0s 6 bytes of data



```
smash_me:
    subq   $0x30, %rsp
    ...
    movq   %rsp, %rdi
    call  gets
    ...
```

48 + 6

← %rsp (buf)

~~A. 33~~

~~B. 36~~

C. 51

D. 54

E. We're lost...

Exploits Based on Buffer Overflows

- ❖ *Buffer overflow bugs can allow remote machines 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)
 - *Still happens!!*
 - **Heartbleed** (2014, affected 17% of servers)
 - Cloudblead (2017)
 - *Fun: Nintendo hacks*
 - Using glitches to rewrite code: <https://www.youtube.com/watch?v=TqK-2jUQBUY>
 - FlappyBird in Mario: <https://www.youtube.com/watch?v=hB6eY73sLV0>

Example: the original Internet worm (1988)

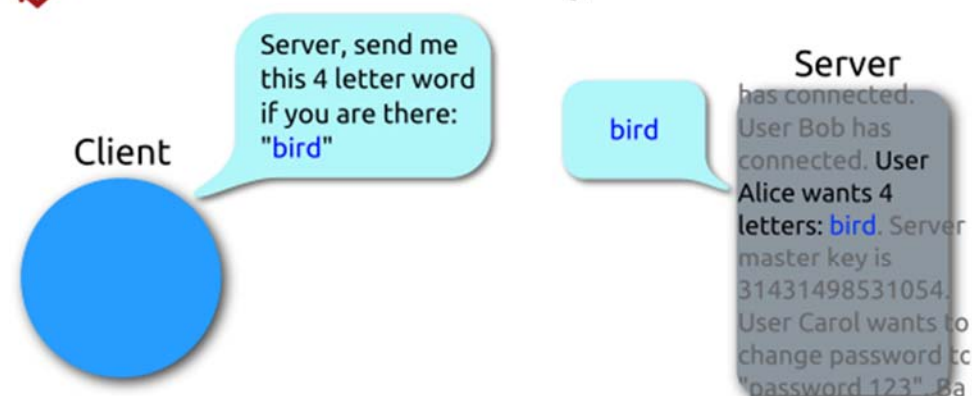
- ❖ Exploited a few vulnerabilities to spread
 - Early versions of the finger server (`fingerd`) used `gets()` to read the argument sent by the client:
 - `finger droh@cs.cmu.edu`
 - Worm attacked `fingerd` server with phony argument:
 - `finger "exploit-code padding new-return-addr"`
 - Exploit code: executed a root shell on the victim machine with a direct TCP 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 young author of the worm was prosecuted...

Heartbleed (2014)

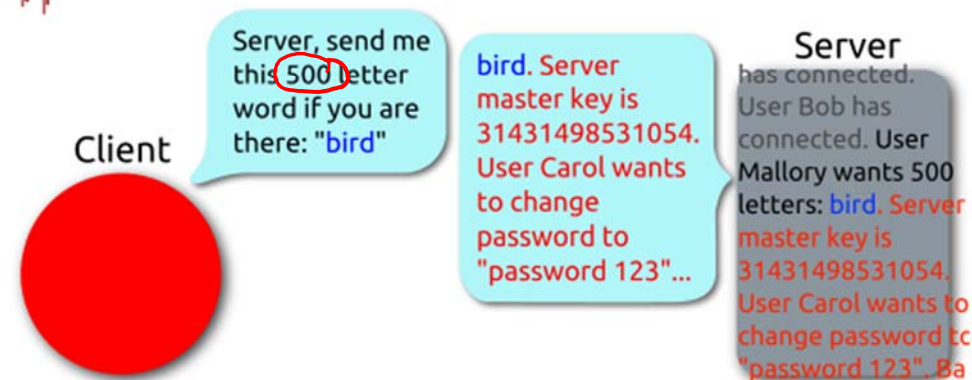
- ❖ Buffer over-read in OpenSSL
 - Open source security library
 - Bug in a small range of versions
- ❖ “Heartbeat” packet
 - Specifies length of message
 - Server echoes it back
 - Library just “trusted” this length
 - Allowed attackers to read contents of memory anywhere they wanted
- ❖ Est. 17% of Internet affected
 - “Catastrophic”
 - Github, Yahoo, Stack Overflow, Amazon AWS, ...



Heartbeat – Normal usage



Heartbeat – Malicious usage



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

Dealing with buffer overflow attacks

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

1) 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

2) System-Level Protections

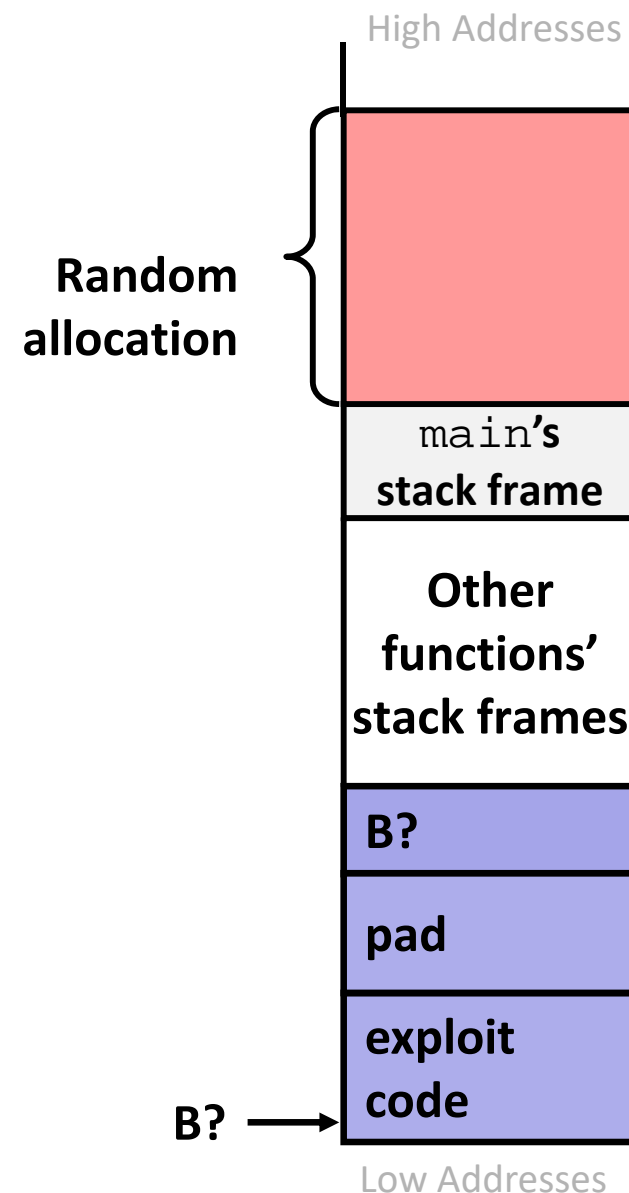
❖ Randomized stack offsets

- 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

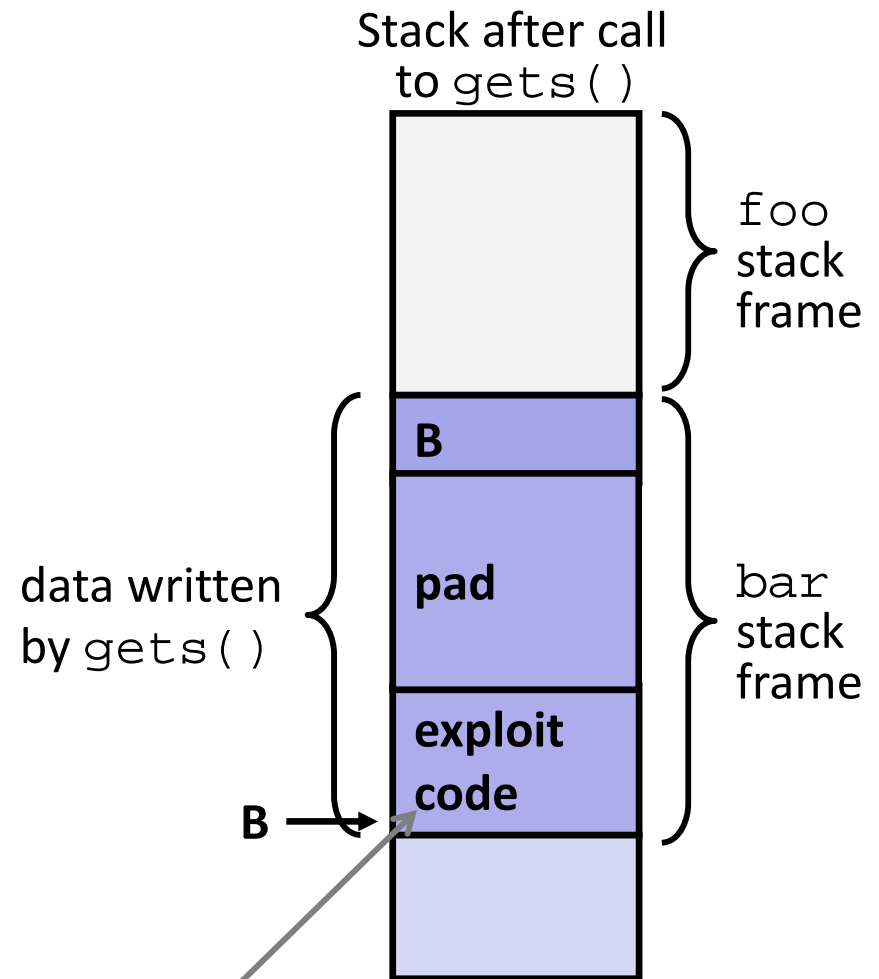
■ **Stack repositioned each time program executes**



2) 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

3) Stack Canaries

- ❖ Basic Idea: place special value (“canary”) on stack just beyond buffer
 - *Secret* value known only to compiler
 - “After” buffer but before return address
 - Check for corruption before exiting function
- ❖ GCC implementation (now default)
 - `-fstack-protector`
 - Code back on Slide 14 (`buf-nsp`) compiled with `-fno-stack-protector` flag

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

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

Protected Buffer Disassembly (buf)

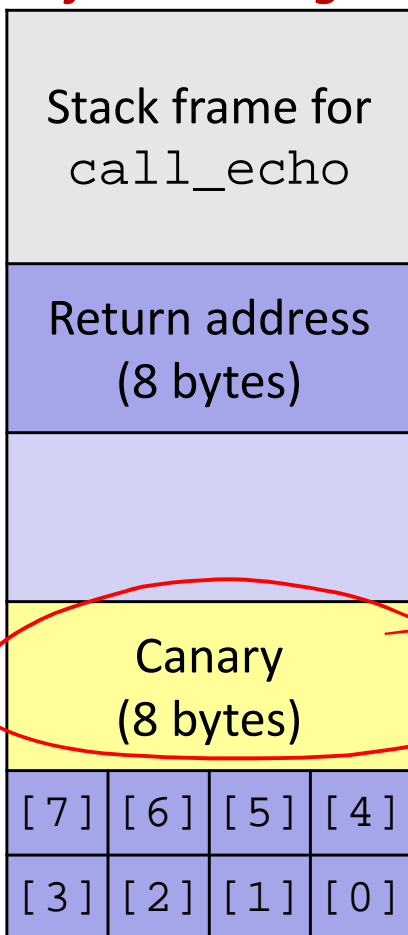
echo:

```
400638:  sub    $0x18,%rsp
40063c:  mov    %fs:0x28,%rax  # read canary value
400645:  mov    %rax,0x8(%rsp) # store canary on Stack
40064a:  xor    %eax,%eax      # erase canary from register
...    ... call printf ...
400656:  mov    %rsp,%rdi
400659:  callq 400530 <gets@plt>
40065e:  mov    %rsp,%rdi
400661:  callq 4004e0 <puts@plt>
400666:  mov    0x8(%rsp),%rax  # read current canary on Stack
40066b:  xor    %fs:0x28,%rax  # compare against original value
400674:  je     40067b <echo+0x43> # if unchanged, then return
400676:  callq 4004f0 <__stack_chk_fail@plt> # stack smashing detected
40067b:  add    $0x18,%rsp
40067f:  retq
```

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);
}
    
```

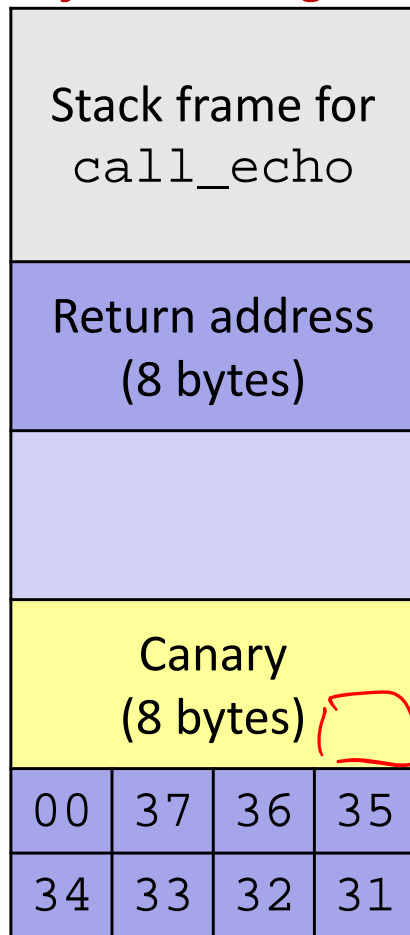
Segment register
(don't worry about it)

```

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

Checking Canary

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
    je     .L2              # if same, OK
    call   __stack_chk_fail # else, FAIL
.L6:
    . . .
    
```

buf ← %rsp

Input: 1234567 &

Summary

- 1) Avoid overflow vulnerabilities
 - Use library routines that limit string lengths

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

- 3) Have compiler use “stack canaries”