# Machine-Level Programming V: Advanced Topics

CS2011: Introduction to Computer Systems
Lecture 10 (3.10)

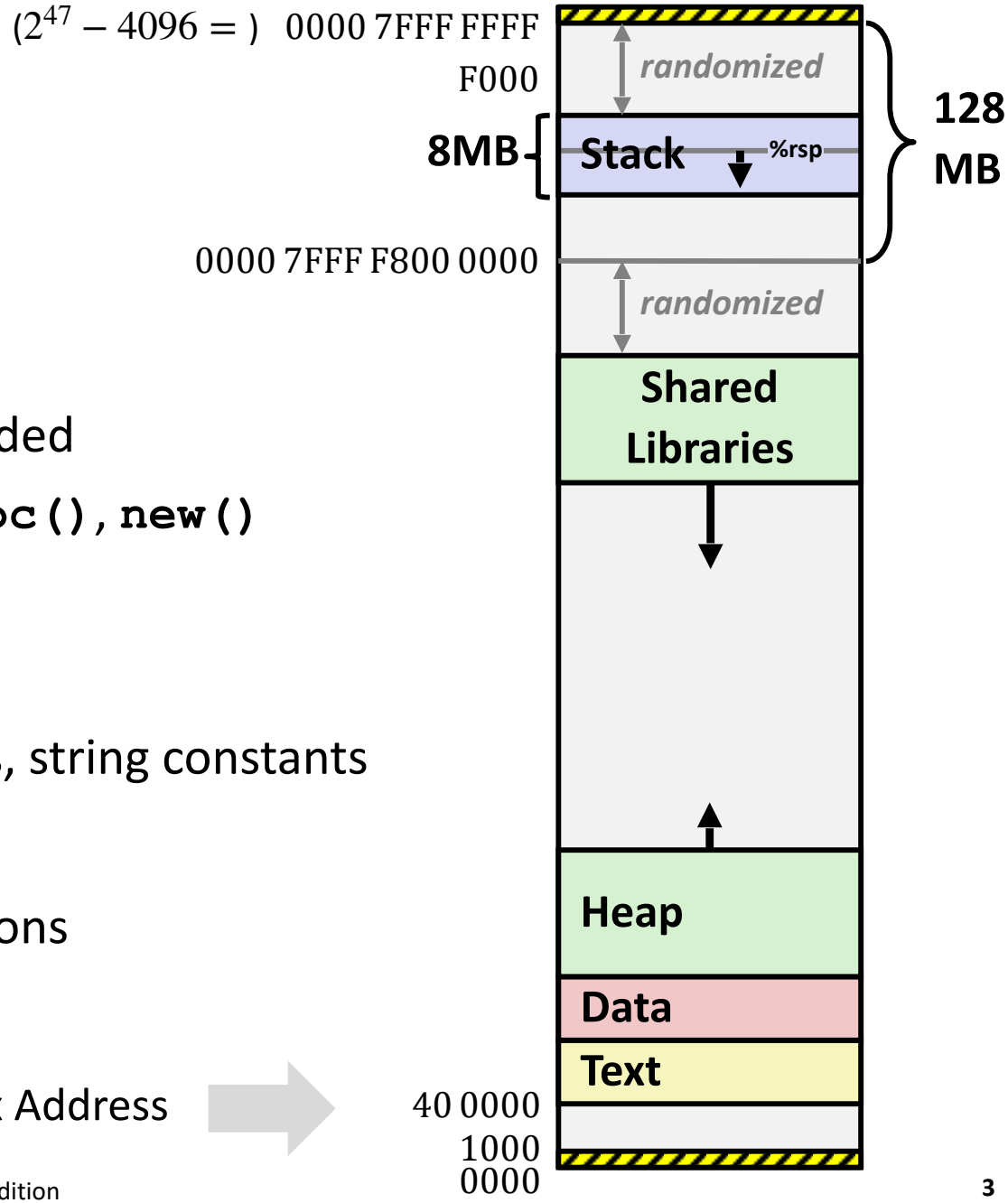
# Machine-Level Programming V: Advanced Topics

- **Memory Layout**
- **Buffer Overflow**
  - Vulnerability
  - Protection
  - Bypassing Protection

# x86-64 Linux Memory Layout

*not drawn to scale*

- **Stack**
  - Runtime stack (8MB limit)
  - e.g., local variables
- **Heap**
  - Dynamically allocated as needed
  - When call **`malloc()`**, **`calloc()`**, **`new()`**
- **Data**
  - Statically allocated data
  - e.g., global vars, **`static`** vars, string constants
- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only



*not drawn to scale*

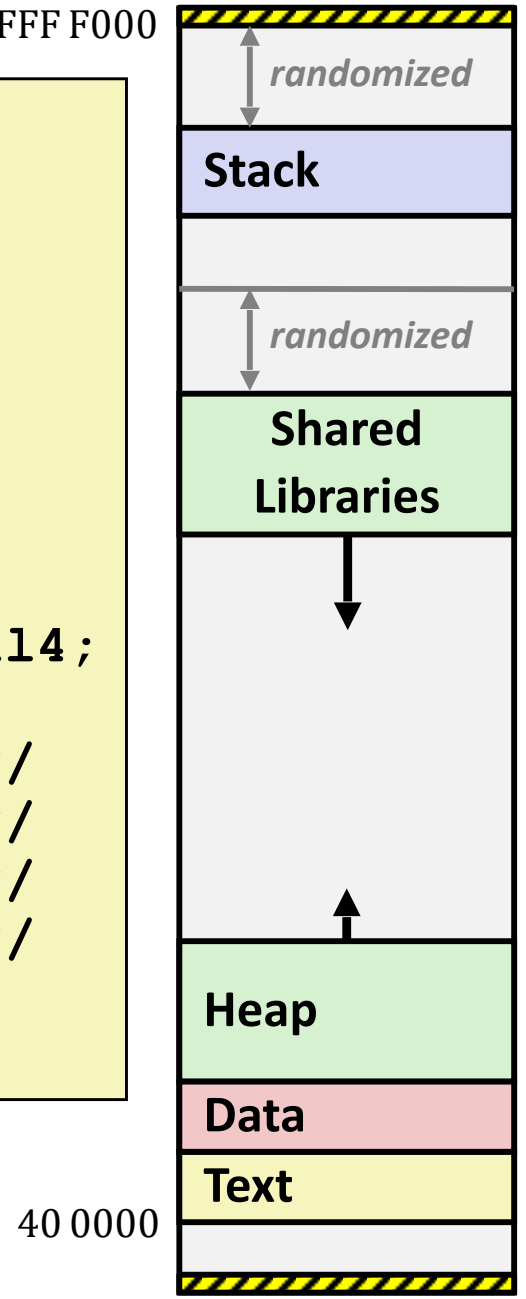
# Memory Allocation Example

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

int global = 0;

int useless() { return 0; }

int main ()
{
    void *phuge1, *psmall2, *phuge3, *psmall4;
    int local = 0;
    phuge1 = malloc(1L << 28);  /* 256 MB */
    psmall2 = malloc(1L << 8);  /* 256  B */
    phuge3 = malloc(1L << 32);  /*   4 GB */
    psmall4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```




*Where does everything go?*

*not drawn to scale*

# x86-64 Example Addresses

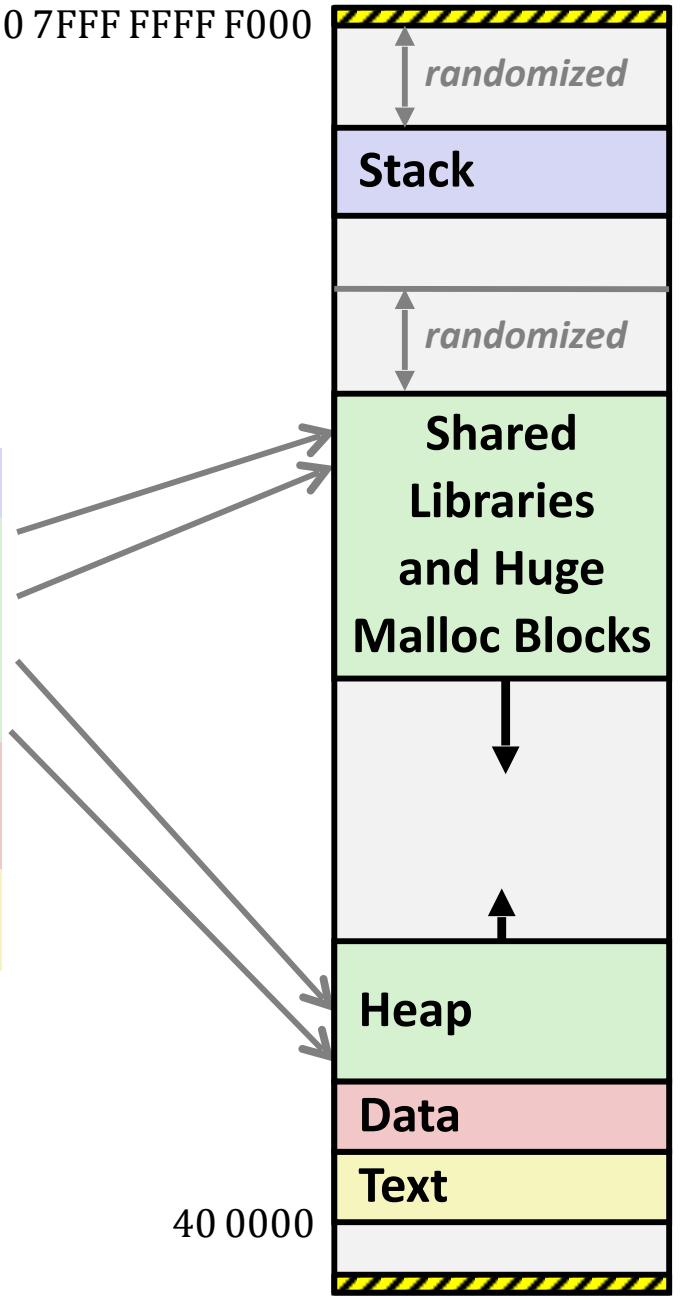
*address range ~$2^{47}$*

0000 7FFF FFFF F000

**White regions are inaccessible —> segmentation fault**

| | |
|---|---|
| `local` | `0x00007ffe4d3be87c` |
| `phuge1` | `0x00007f7262a1e010` |
| `phuge3` | `0x00007f7162a1d010` |
| `psmall4` | `0x0000000008359d120` |
| `psmall2` | `0x0000000008359d010` |
| `big_array` | `0x0000000080601060` |
| `huge_array` | `0x0000000000601060` |
| `main()` | `0x000000000040060c` |
| `useless()` | `0x0000000000400590` |

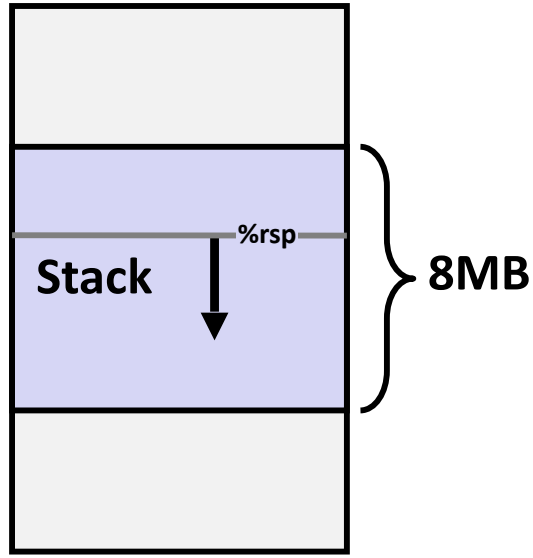
**(Exact values can vary)**



*not drawn to scale*

# Runaway Stack Example

```
int recurse(int x) {
    int a[1<<15];  // 4*2^15 =  128 KiB
    printf("x = %d.  a at %p\n", x, a);
    a[0] = (1<<14)-1;
    a[a[0]] = x-1;
    if (a[a[0]] == 0)
        return -1;
    return recurse(a[a[0]]) - 1;
}
```




- Functions store local data in stack frame
- Recursive functions cause deep nesting of frames
- What happens when we run out of space?

```
./runaway 67
x = 67.  a at 0x7ffd18aba930
x = 66.  a at 0x7ffd18a9a920
x = 65.  a at 0x7ffd18a7a910
x = 64.  a at 0x7ffd18a5a900
. . .
x = 4.  a at 0x7ffd182da540
x = 3.  a at 0x7ffd182ba530
x = 2.  a at 0x7ffd1829a520
Segmentation fault (core dumped)
```

# Machine-Level Programming V: Advanced Topics

- Memory Layout
- **Buffer Overflow**
  - Vulnerability
  - Protection
  - Bypassing Protection

# Recall: Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)   ->    3.1400000000
fun(1)   ->    3.1400000000
fun(2)   ->    3.1399998665
fun(3)   ->    2.0000006104
fun(6)   ->    Stack smashing detected
fun(8)   ->    Segmentation fault
```

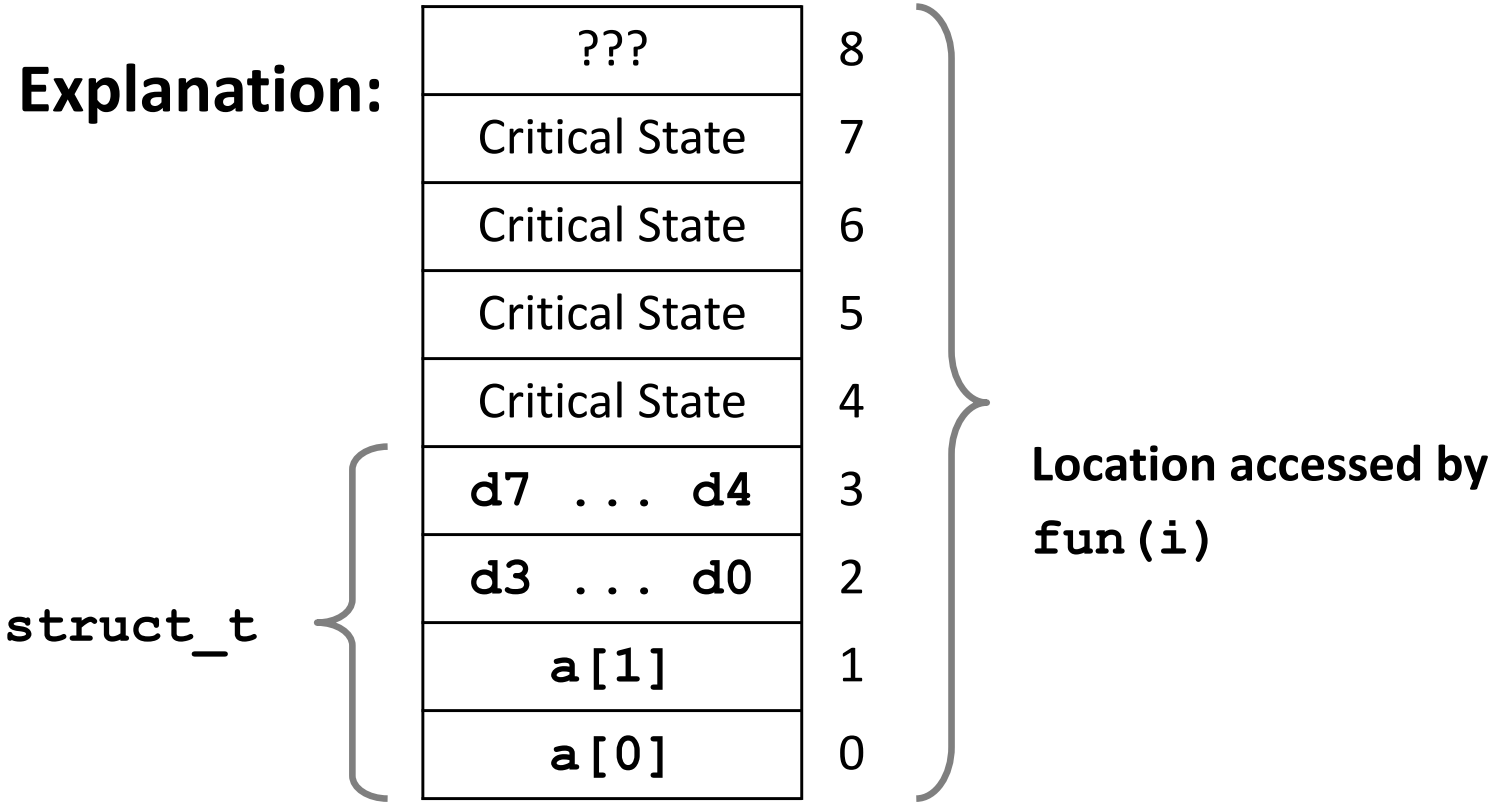
- Result is system specific

# Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0)   ->   3.1400000000
fun(1)   ->   3.1400000000
fun(2)   ->   3.1399998665
fun(3)   ->   2.0000006104
fun(4)   ->   3.1400000000
fun(8)   ->   Segmentation fault
```

**Explanation:**



# Such Problems are a BIG Deal

- **Generally called a "buffer overflow"**
  - When exceeding the memory size allocated for an array
- **Why a big deal?**
  - It's the #1 technical cause of security vulnerabilities
    - #1 overall cause is social engineering / user ignorance
- **Most common form**
  - Unchecked lengths on string user inputs
  - Particularly for bounded character arrays on the stack
    - sometimes referred to as stack smashing

# 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 library functions

- **`strcpy`, `strcat`**: Copy strings of arbitrary length
- **`scanf`, `fscanf`, `sscanf`,** when given **`%s`** conversion specification

# Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

← BTW, how big is big enough?

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

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901 2
0123456789012345678901 2
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
Segmentation Fault
```

# Buffer Overflow Disassembly

## echo:

```
000000000040069c <echo>:
  40069c:  48 83 ec 18           sub    $0x18,%rsp
  4006a0:  48 89 e7              mov    %rsp,%rdi
  4006a3:  e8 a5 ff ff ff        callq  40064d <gets>
  4006a8:  48 89 e7              mov    %rsp,%rdi
  4006ab:  e8 50 fe ff ff        callq  400500 <puts@plt>
  4006b0:  48 83 c4 18           add    $0x18,%rsp
  4006b4:  c3                    retq
```

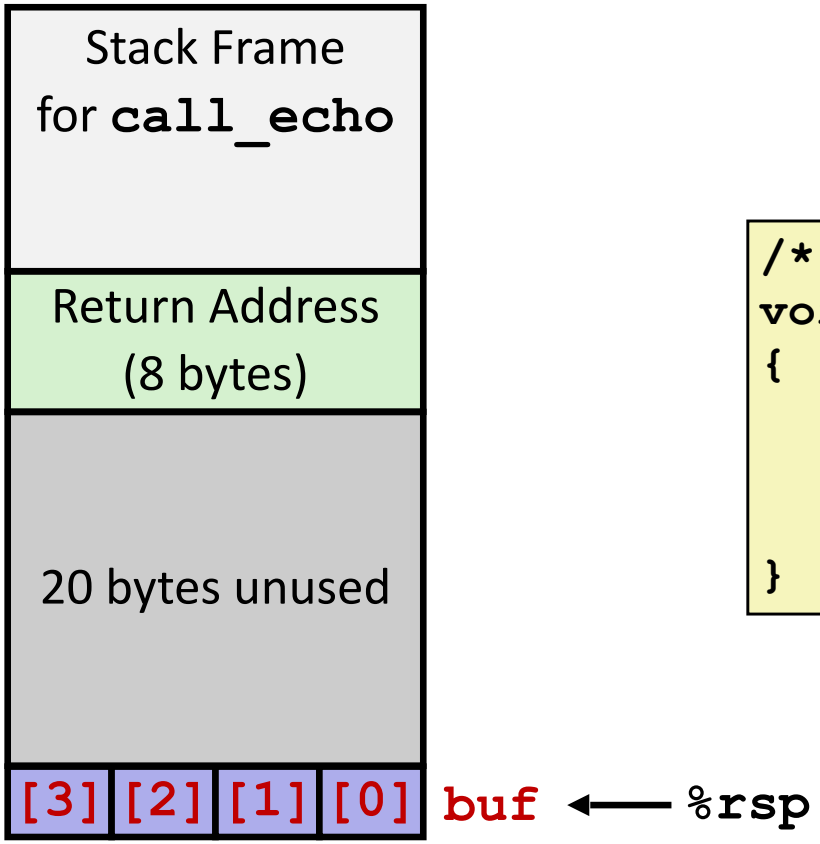
**Code is system dependent - you may see something different but idea is same**

## call_echo:

```
  4006b5:  48 83 ec 08           sub    $0x8,%rsp
  4006b9:  b8 00 00 00 00        mov    $0x0,%eax
  4006be:  e8 d9 ff ff ff        callq  40069c <echo>
  4006c3:  48 83 c4 08           add    $0x8,%rsp
  4006c7:  c3                    retq
```

# Buffer Overflow Stack Example

***Before call to gets***

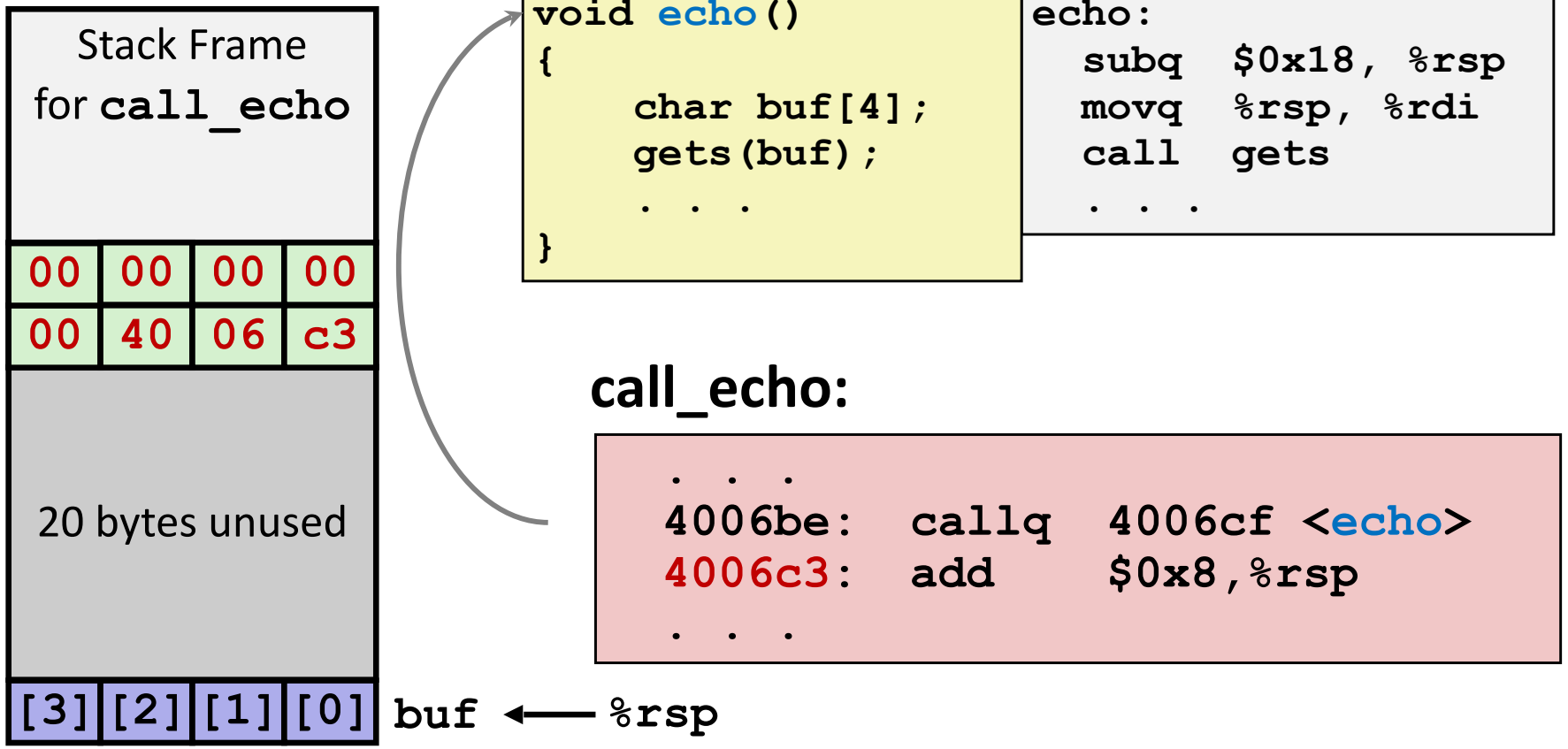



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq $0x18, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

# Buffer Overflow Stack Example

***Before call to gets***



# Buffer Overflow Stack Example #1

*After call to gets*

Stack Frame for `call_echo`

| | | | |
|---|---|---|---|
| 00 | 00 | 00 | 00 |
| 00 | 40 | 06 | c3 |
| 00 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

buf ⟵ %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $0x18, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

**call_echo:**

```
   . . .
   4006be:  callq  4006cf <echo>
   4006c3:  add    $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901 2
0123456789012345678901 2
```

"0123456789012345678901 2\0"

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

# Buffer Overflow Stack Example #2

*After call to gets*

Stack Frame for `call_echo`

| | | | |
|---|---|---|---|
| 00 | 00 | 00 | 00 |
| 00 | 40 | 06 | 00 |
| 33 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

buf ⟵ %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $0x18, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

**call_echo:**

```
   . . .
   4006be:  callq  4006cf <echo>
   4006c3:  add    $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901230123456789012345678901230
Segmentation fault
```

**Program "returned" to 0x0400600, and then crashed.**

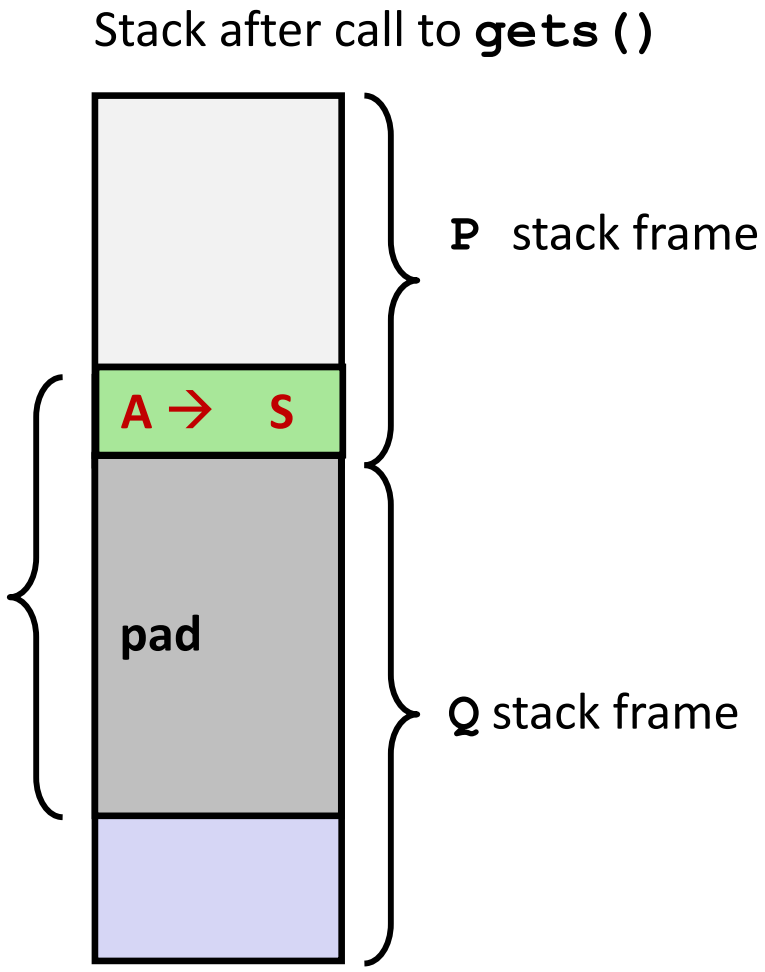
# Stack Smashing Attacks

```
void P(){
  Q();
  ...
}
```

return address A

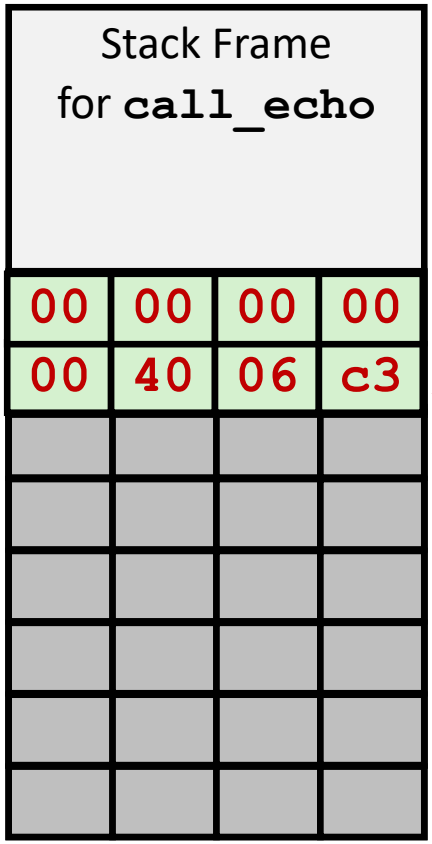
```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

```
void S(){
/* Something
   unexpected */
  ...
}
```




- Overwrite normal return address A with address of some other code S
- When Q executes `ret`, will jump to other code

# Crafting Smashing String




```
int echo() {
   char buf[4];
   gets(buf);
   ...
   return ...;
}
```

*Target Code*

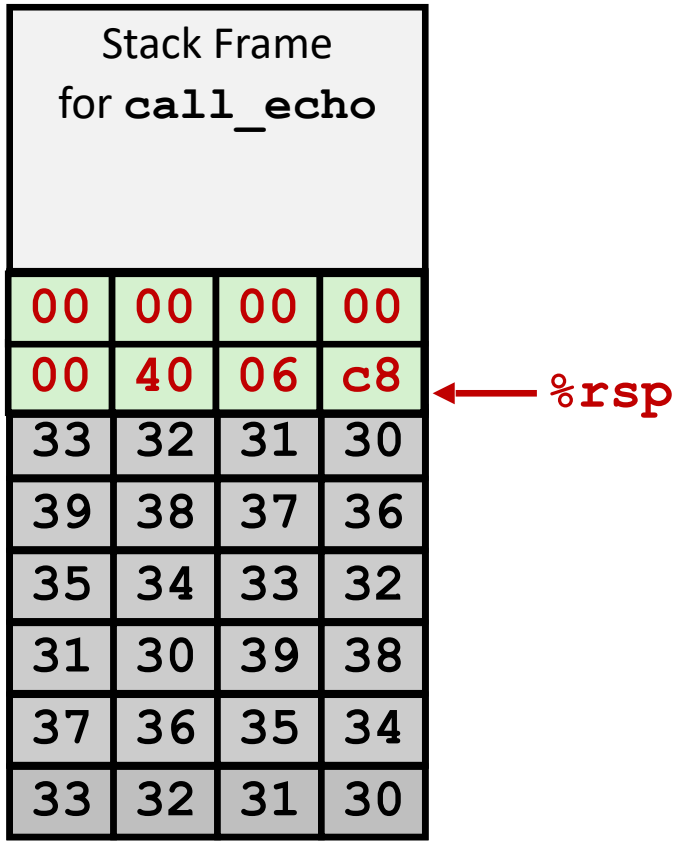
```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

```
00000000004006c8 <smash>:
  4006c8:        48 83 ec 08
```

*Attack String (Hex)*

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
c8 06 40 00 00 00 00 00
```

# Smashing String Effect




*Target Code*

```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

```
00000000004006c8 <smash>:
  4006c8:        48 83 ec 08
```

*Attack String (Hex)*

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
c8 06 40 00 00 00 00 00
```

# Performing Stack Smash

```
linux> cat smash-hex.txt
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 c8 06 40 00 00 00 00 00
linux> cat smash-hex.txt | ./hexify | ./bufdemo-nsp
Type a string:012345678901234567890123?@
I've been smashed!
```

- Put hex sequence in file smash-hex.txt
- Use hexify program to convert hex digits to characters
  - Some of them are non-printing
- Provide as input to vulnerable program

```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

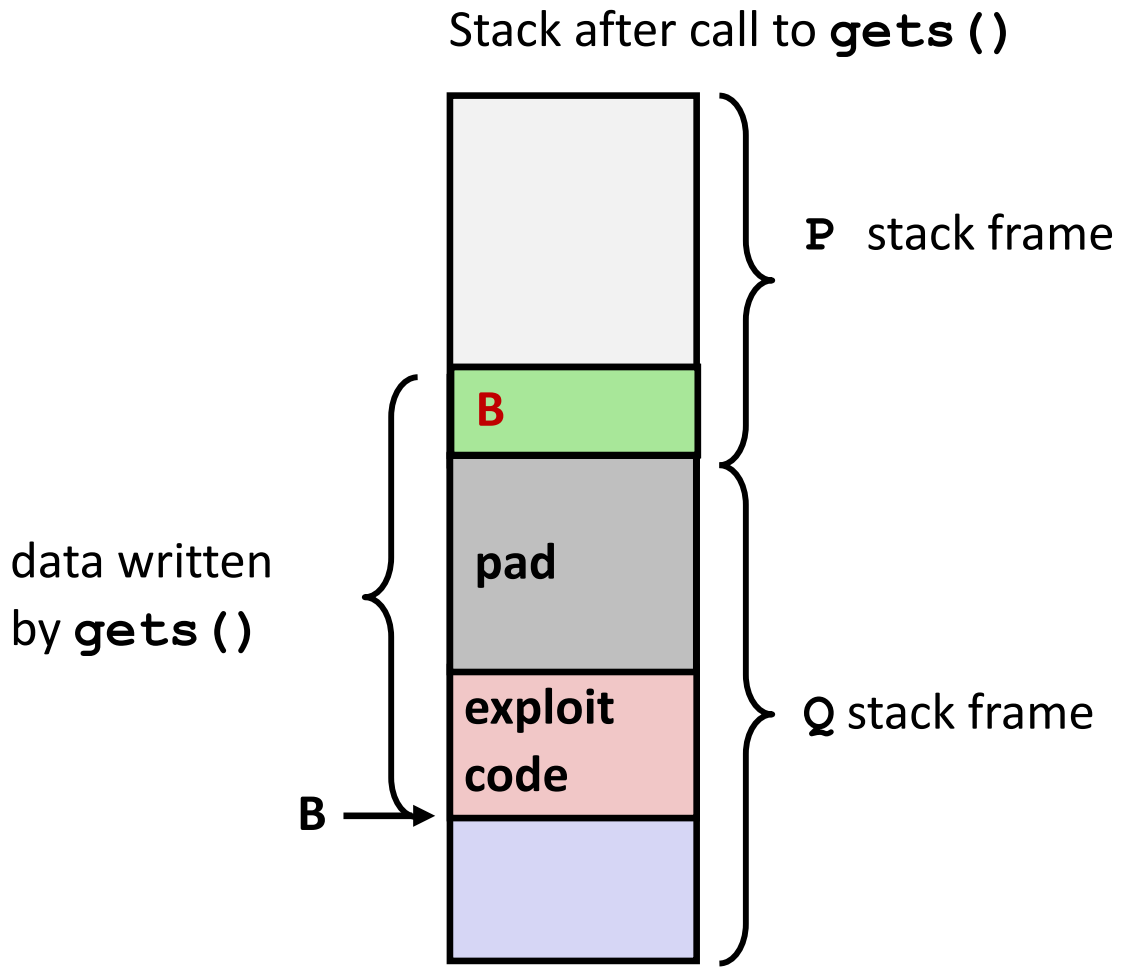
```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
c8 06 40 00 00 00 00 00
```

# Code Injection Attacks

```
void P(){
  Q();
  ...
}
```

return address A

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



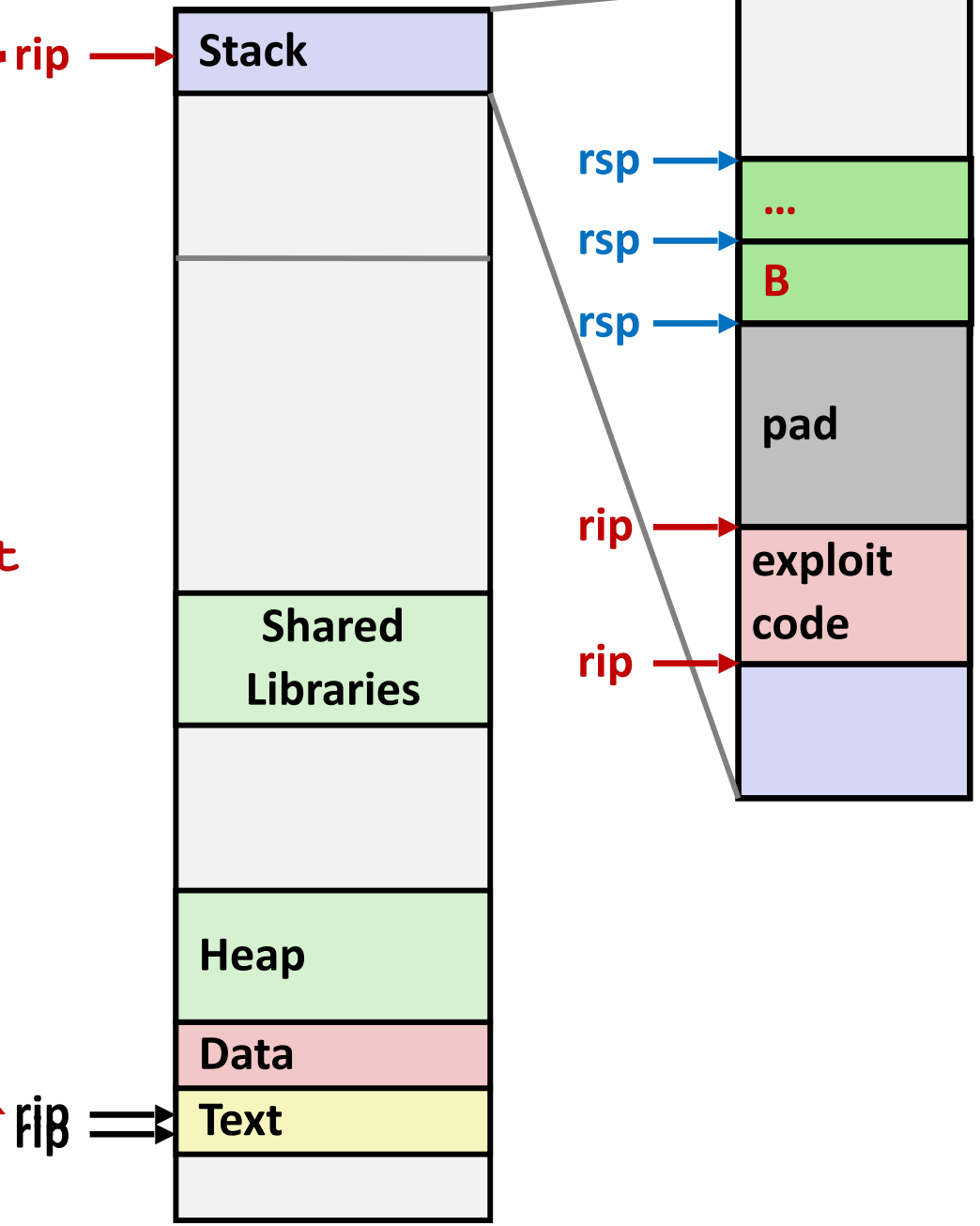

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes `ret`, will jump to exploit code

# How Does The Attack Code Execute?

```
void P(){
  Q();
  ...
}
```

```
int Q() {
  char buf[64];
  gets(buf); // A->B
  ...
  return ...;
}
```

ret ret



# Machine-Level Programming V: Advanced Topics

- Memory Layout
- **Buffer Overflow**
  - Vulnerability
  - Protection
  - Bypassing Protection

# What to Do About Buffer Overflow Attacks

- Avoid overflow vulnerabilities
- Employ system-level protections
- Have compiler use “stack canaries”
- Lets talk about each…

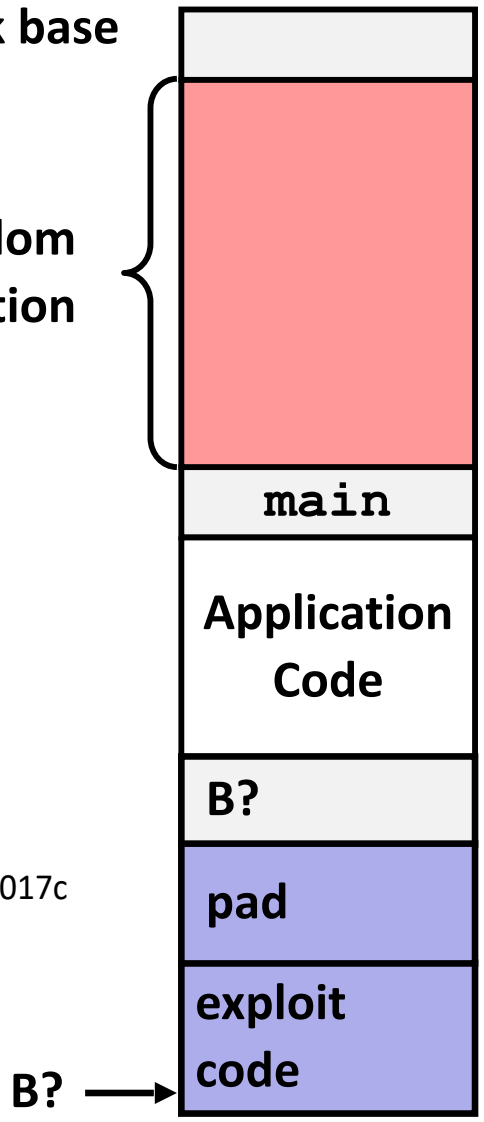
# 1. Avoid Overflow Vulnerabilities in Code (!)

```
/* Echo Line */
void echo()
{
    char buf[4];
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- **For example, use library routines that limit string lengths**
  - **fgets** instead of **gets**
  - **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 Can Help

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
  - e.g., 5 executions of memory allocation code
    - Stack repositioned each time program executes

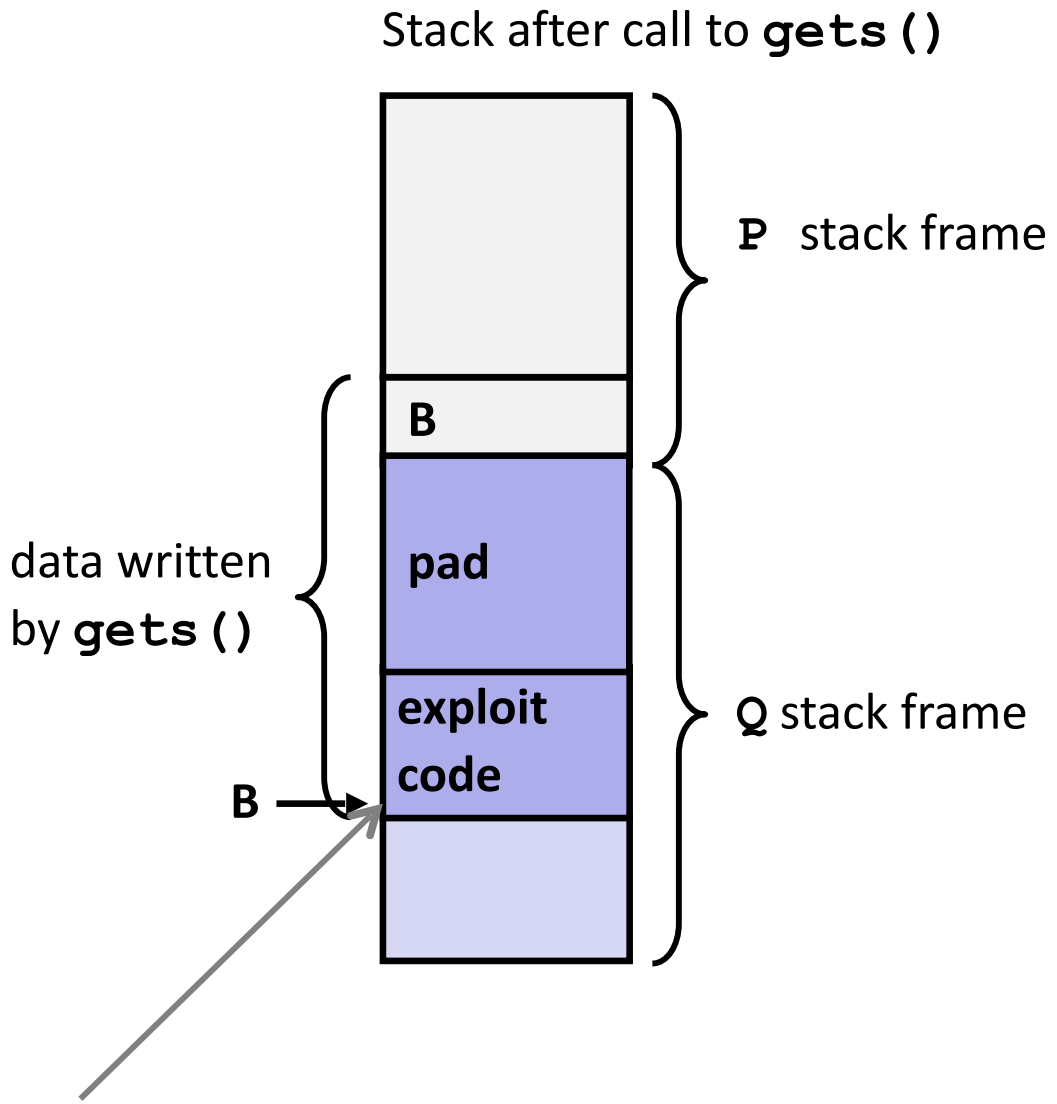



| | | | | | |
|---|---|---|---|---|---|
| local | 0x7ffe4d3be87c | 0x7fff75a4f9fc | 0x7ffeadb7c80c | 0x7ffeaea2fdac | 0x7ffcd452017c |

# 2. System-Level Protections Can Help

## Non-executable memory

- Older x86 CPUs would execute machine code from any readable address
- x86-64 added a way to mark regions of memory as *not executable*
- Immediate crash on jumping into any such region
- Current Linux and Windows mark the stack this way




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

# 3. Stack Canaries Can Help

## Idea

- Place special value ("canary") on stack just beyond buffer
- Check for corruption before exiting function

## GCC Implementation

- `-fstack-protector-all`
- Now the default (disabled earlier)
  - To disable on newer versions of gcc: -fno-stack-protector

```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:012345678
*** stack smashing detected ***
```

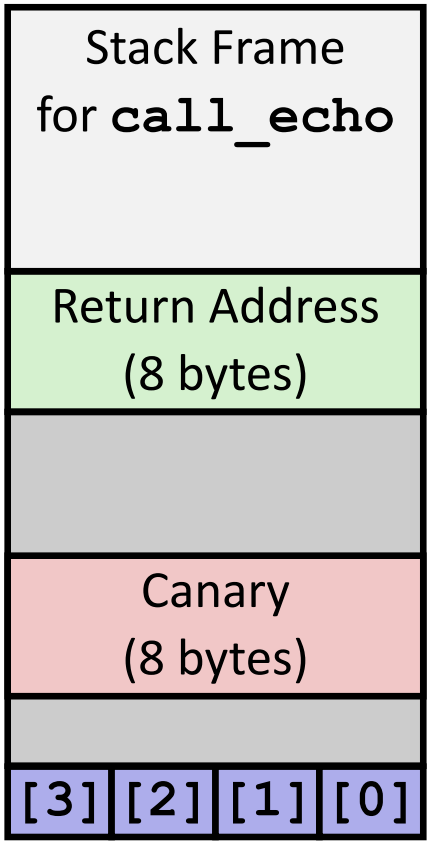
# Protected Buffer Disassembly

**echo:**

```
40072f:  sub    $0x18,%rsp
400733:  mov    %fs:0x28,%rax # random value at %fs:0x28
40073c:  mov    %rax,0x8(%rsp)
400741:  xor    %eax,%eax
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax
400758:  xor    %fs:0x28,%rax # checking if value changed
400761:  je     400768 <echo+0x39>
400763:  callq  400580 <__stack_chk_fail@plt>
400768:  add    $0x18,%rsp
40076c:  retq
```

# Setting Up Canary

***Before call to gets***

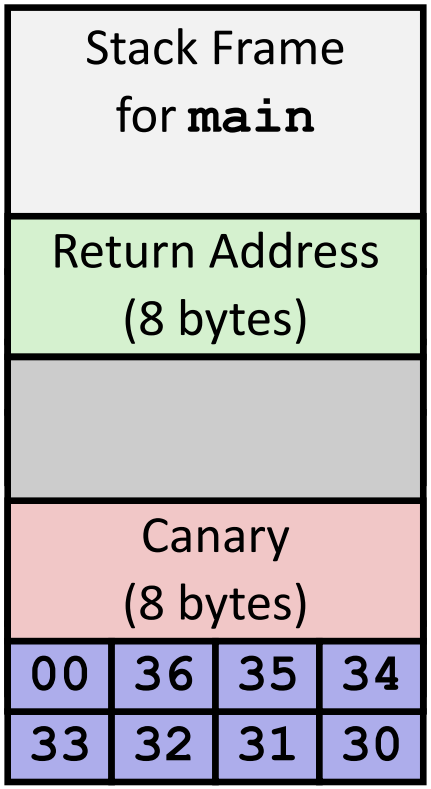



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    mov     %fs:0x28, %rax  # Get canary
    mov     %rax, 0x8(%rsp) # Place on stack
    xor     %eax, %eax      # Erase register
    . . .
```

# Checking Canary

*After call to gets*




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

**Input:** ***0123456***

***Some systems:***
***LSB of canary is 0x00***
***Allows input 01234567***

```
echo:
    . . .
    mov     0x8(%rsp),%rax    # Retrieve from stack
    xor     %fs:0x28,%rax     # Compare to canary
    je      .L6               # If same, OK
    call    __stack_chk_fail  # FAIL
```