

Machine-Level Programming V: Advanced Topics

Introduction to Computer Systems
8th Lecture, Oct. 21, 2020

Instructors:

Class 1: Chen Xiangqun, Sun Guangyu, Liu Xianhua

Class 2: Guan Xuetao

Class 3: Lu Junlin

Today

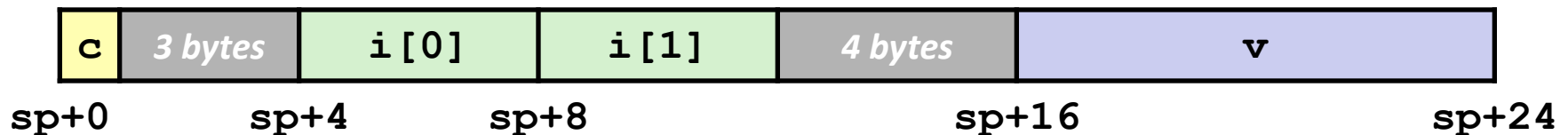
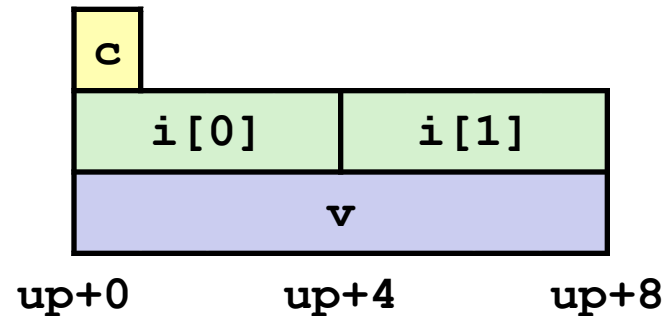
- **Unions**
- **Memory Layout**
- **Buffer Overflow**
 - Vulnerability
 - Protection

Union Allocation

- Allocate according to largest element
- Can only use one field at a time

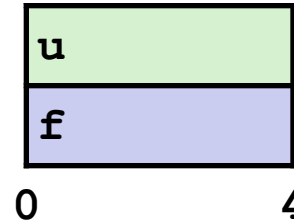
```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Using Union to Access Bit Patterns

```
typedef union {  
    float f;  
    unsigned u;  
} bit_float_t;
```



```
float bit2float(unsigned u)  
{  
    bit_float_t arg;  
    arg.u = u;  
    return arg.f;  
}
```

Same as (float) u?

```
unsigned float2bit(float f)  
{  
    bit_float_t arg;  
    arg.f = f;  
    return arg.u;  
}
```

Same as (unsigned) f?

Byte Ordering Revisited

■ Idea

- Short/long/quad words stored in memory as 2/4/8 consecutive bytes
- Which byte is most (least) significant?
- Can cause problems when exchanging binary data between machines

■ Big Endian

- Most significant byte has lowest address
- Sparc

■ Little Endian

- Least significant byte has lowest address
- Intel x86, ARM Android and IOS

■ Bi Endian

- Can be configured either way
- ARM

Byte Ordering Example

```
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;
```

32-bit

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

64-bit

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Byte Ordering on IA32

Little Endian

f0	f1	f2	f3	f4	f5	f6	f7
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

LSB
MSB
LSB
MSB

←

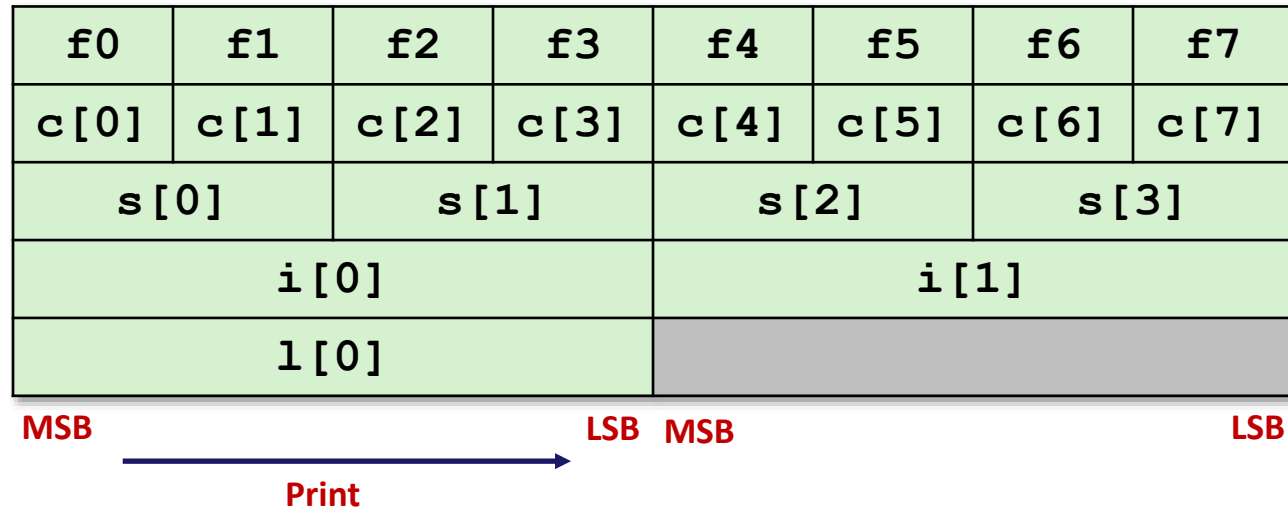
Print

Output:

Characters	0-7	==	[0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts	0-3	==	[0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints	0-1	==	[0xf3f2f1f0,0xf7f6f5f4]
Long	0	==	[0xf3f2f1f0]

Byte Ordering on Sun

Big Endian



Output on Sun:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]

Shorts 0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]

Ints 0-1 == [0xf0f1f2f3,0xf4f5f6f7]

Long 0 == [0xf0f1f2f3]

Byte Ordering on x86-64

Little Endian

f0	f1	f2	f3	f4	f5	f6	f7
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

LSB

MSB

Print

Output on x86-64:

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]

```
Shorts    0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
```

```
Ints      0-1 == [0xf3f2f1f0,0xf7f6f5f4]
```

```
Long      0      == [0xf7f6f5f4f3f2f1f0]
```

Summary of Compound Types in C

■ Arrays

- Contiguous allocation of memory
- Aligned to satisfy every element's alignment requirement
- Pointer to first element
- No bounds checking

■ Structures

- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment

■ Unions

- Overlay declarations
- Way to circumvent type system

Today

- Unions
- **Memory Layout**
- Buffer Overflow
 - Vulnerability
 - 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

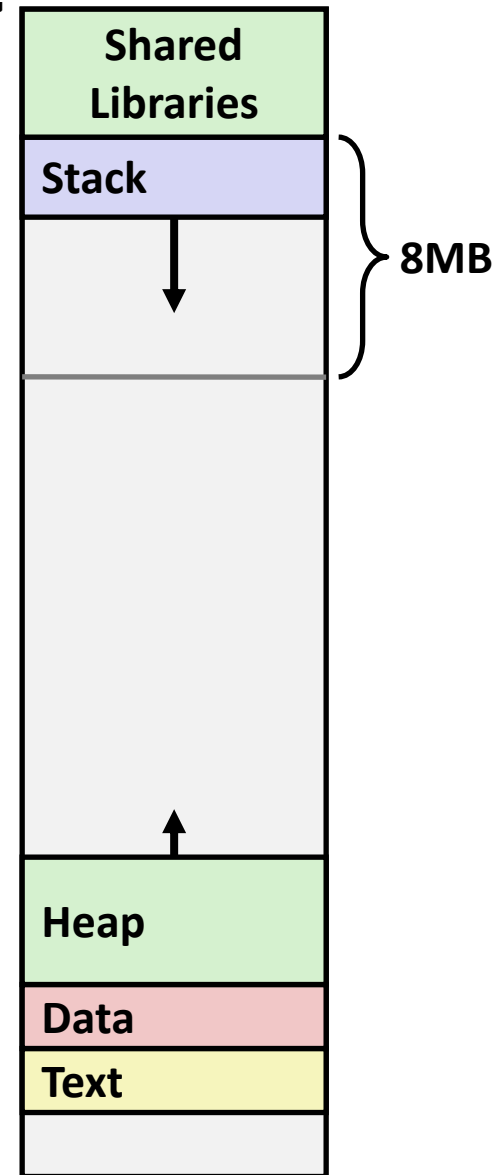
■ Text / Shared Libraries

- Executable machine instructions
- Read-only

Hex Address



400000
000000



not drawn to scale

Memory Allocation Example

00007FFFFFFFFFFFFF

```

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

x86-64 Example Addresses

address range $\sim 2^{47}$

```

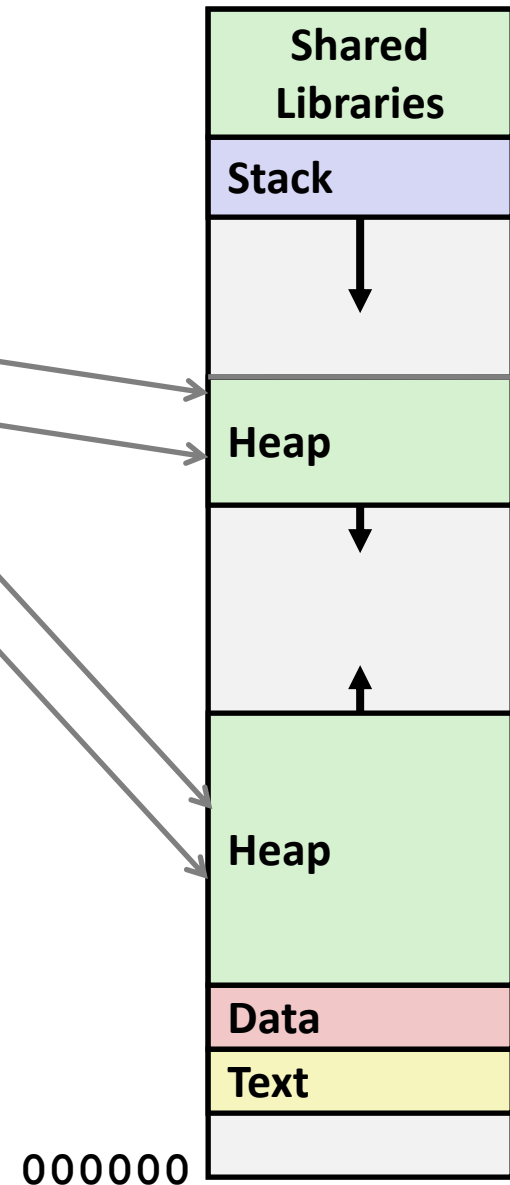
local
p1
p3
p4
p2
big_array
huge_array
main()
useless()

```

```

0x00007ffe4d3be87c
0x00007f7262a1e010
0x00007f7162a1d010
0x000000008359d120
0x000000008359d010
0x0000000080601060
0x0000000000601060
0x000000000040060c
0x0000000000400590

```



not drawn to scale

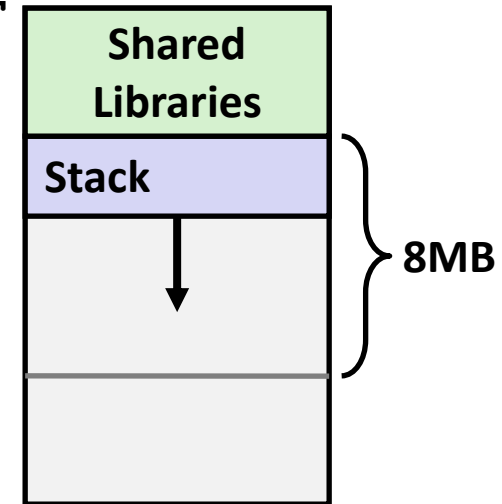
Runaway Stack Example

00007FFFFFFFFFFFFFFF

```

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 on in stack frame
- Recursive functions cause deep nesting of frames

```

./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)

```


Today

- Unions
- Memory Layout
- **Buffer Overflow**
 - Vulnerability
 - 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

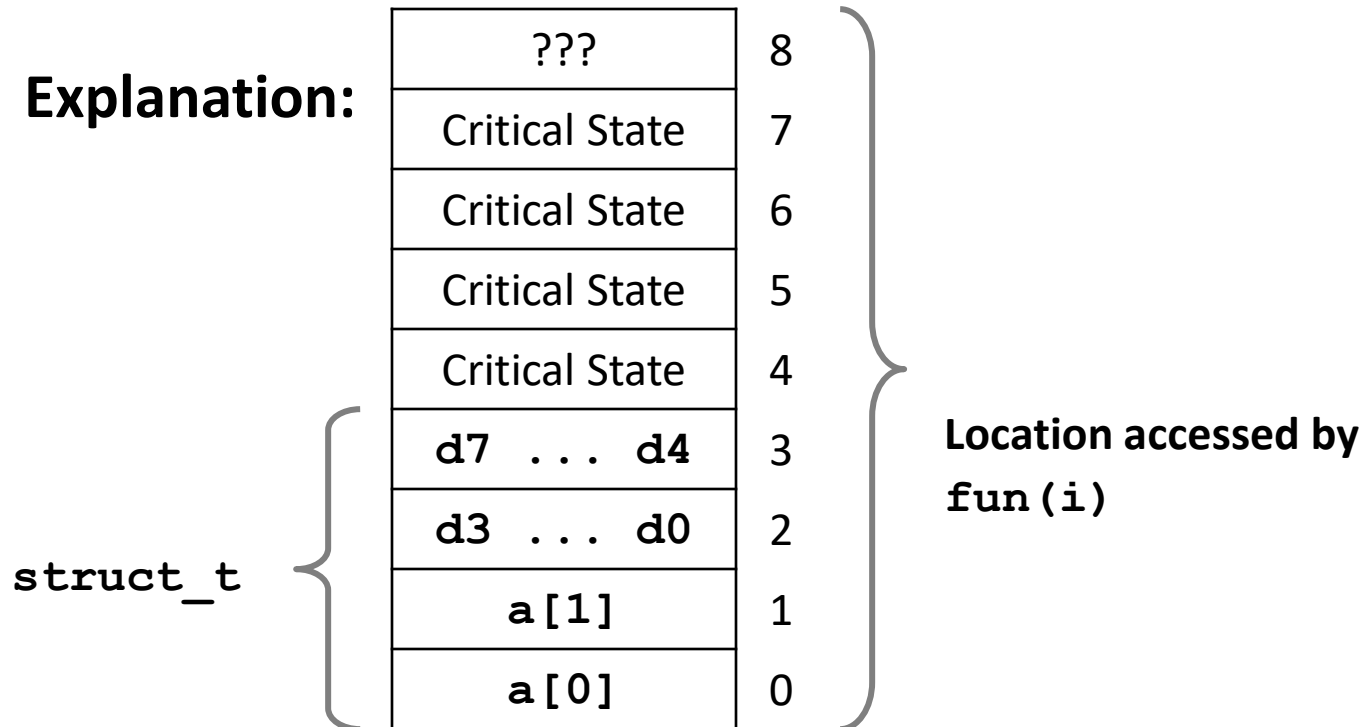
- Result is system specific

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)	->	Segmentation fault
fun(8)	->	3.1400000000

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 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:01234567890123456789012  
01234567890123456789012
```

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

Buffer Overflow Disassembly

echo:

00000000004006cf <echo>:

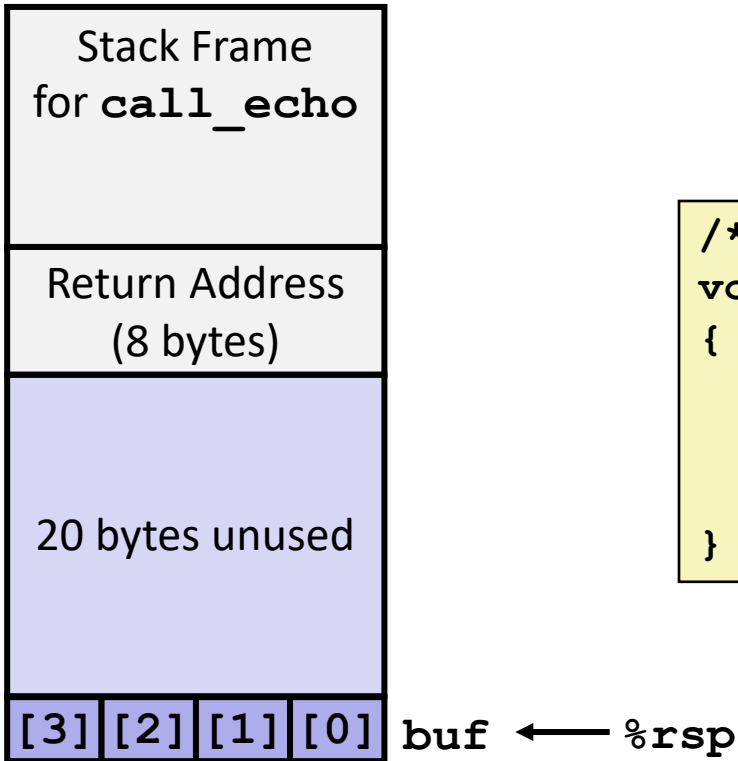
4006cf:	48 83 ec 18	sub	\$0x18 , %rsp
4006d3:	48 89 e7	mov	%rsp , %rdi
4006d6:	e8 a5 ff ff ff	callq	400680 <gets>
4006db:	48 89 e7	mov	%rsp, %rdi
4006de:	e8 3d fe ff ff	callq	400520 <puts@plt>
4006e3:	48 83 c4 18	add	\$0x18, %rsp
4006e7:	c3	retq	

call_echo:

4006e8:	48 83 ec 08	sub	\$0x8, %rsp
4006ec:	b8 00 00 00 00	mov	\$0x0, %eax
4006f1:	e8 d9 ff ff ff	callq	4006cf <echo>
4006f6:	48 83 c4 08	add	\$0x8, %rsp
4006fa:	c3	retq	

Buffer Overflow Stack

Before call to gets

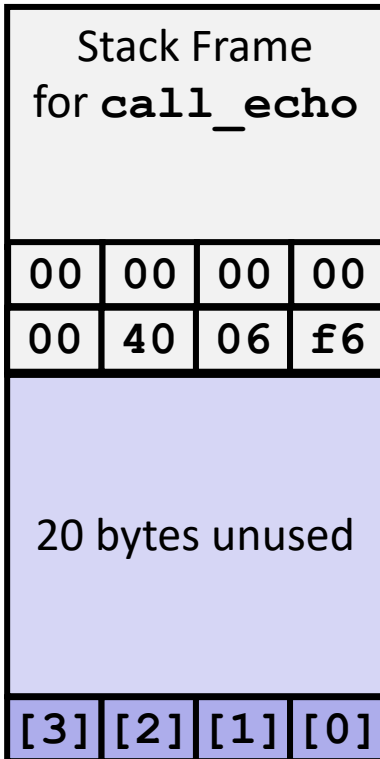


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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


Buffer Overflow Stack Example

Before call to gets



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

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

call_echo:

```
. . .
4006f1:  callq    4006cf <echo>
4006f6:  add      $0x8,%rsp
. . .
```

Buffer Overflow Stack Example #1

After call to gets

Stack Frame for <code>call_echo</code>			
00	00	00	00
00	40	06	f6
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    $24, %rsp
    movq    %rsp, %rdi
    call    gets
    . . .
```

`call_echo:`

```
. . .
4006f1:  callq    4006cf <echo>
4006f6:  add      $0x8, %rsp
. . .
```

```
unix> ./bufdemo-nsp
Type a string: 01234567890123456789012
01234567890123456789012
```

```
"01234567890123456789012\0"
```

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack Example #2

After call to gets

Stack Frame for <code>call_echo</code>			
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    $24, %rsp
    movq    %rsp, %rdi
    call    gets
    . . .
```

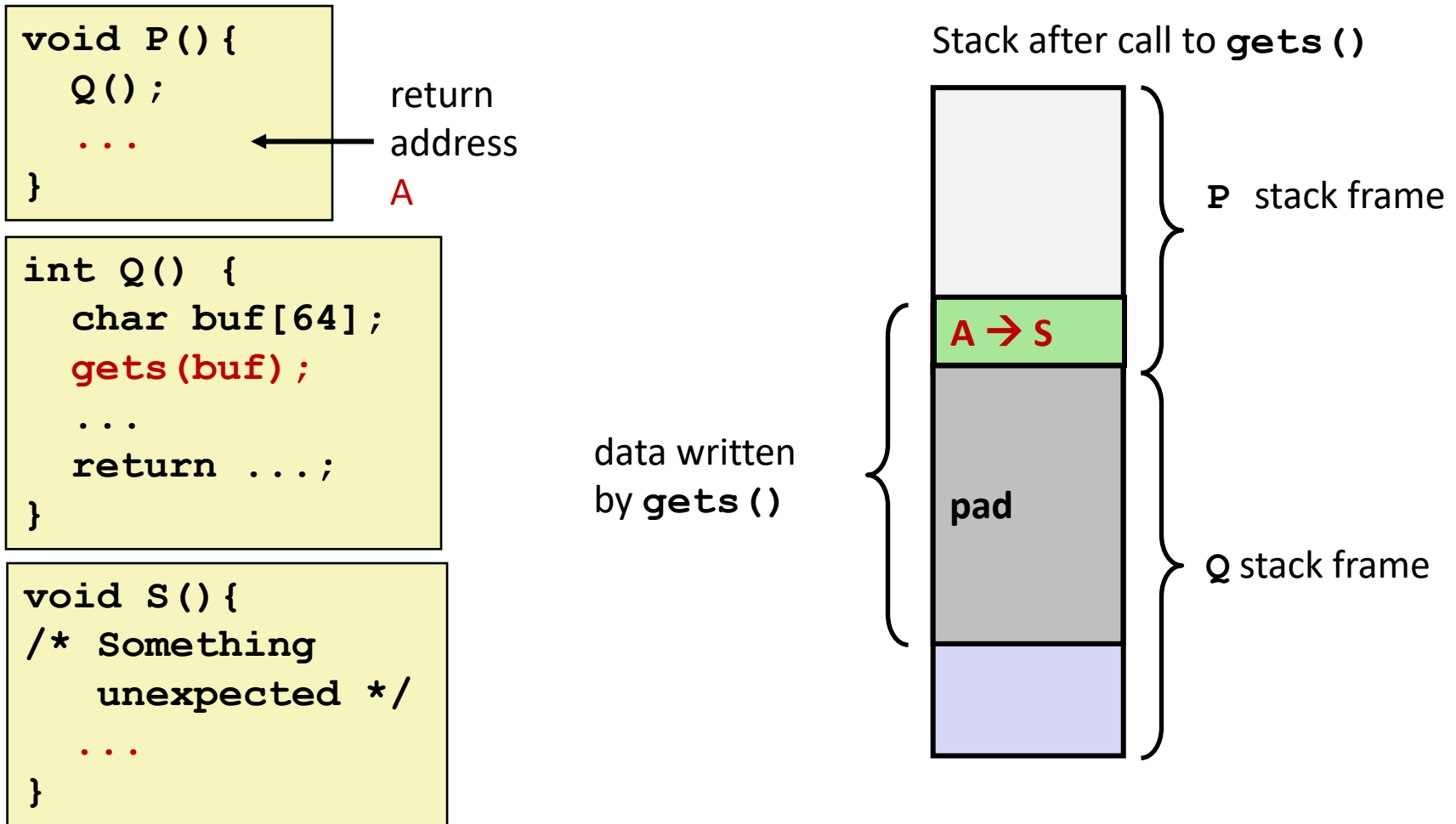
`call_echo:`

```
. . .
4006f1:  callq    4006cf <echo>
4006f6:  add      $0x8, %rsp
. . .
```

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

Program “returned” to 0x0400600, and then crashed.

Stack Smashing Attacks



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

Crafting Smashing String

Stack Frame for call echo			
00	00	00	00
00	48	83	80
00	00	00	00
00	40	06	fb

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

← %rsp

24 bytes

Target Code

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

```
00000000004006fb <smash>:
4006fb:          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
fb 06 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Smashing String Effect

Stack Frame for <code>call echo</code>			
00	00	00	00
00	48	83	80
00	00	00	00
00	40	06	fb
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

← `%rsp`

Target Code

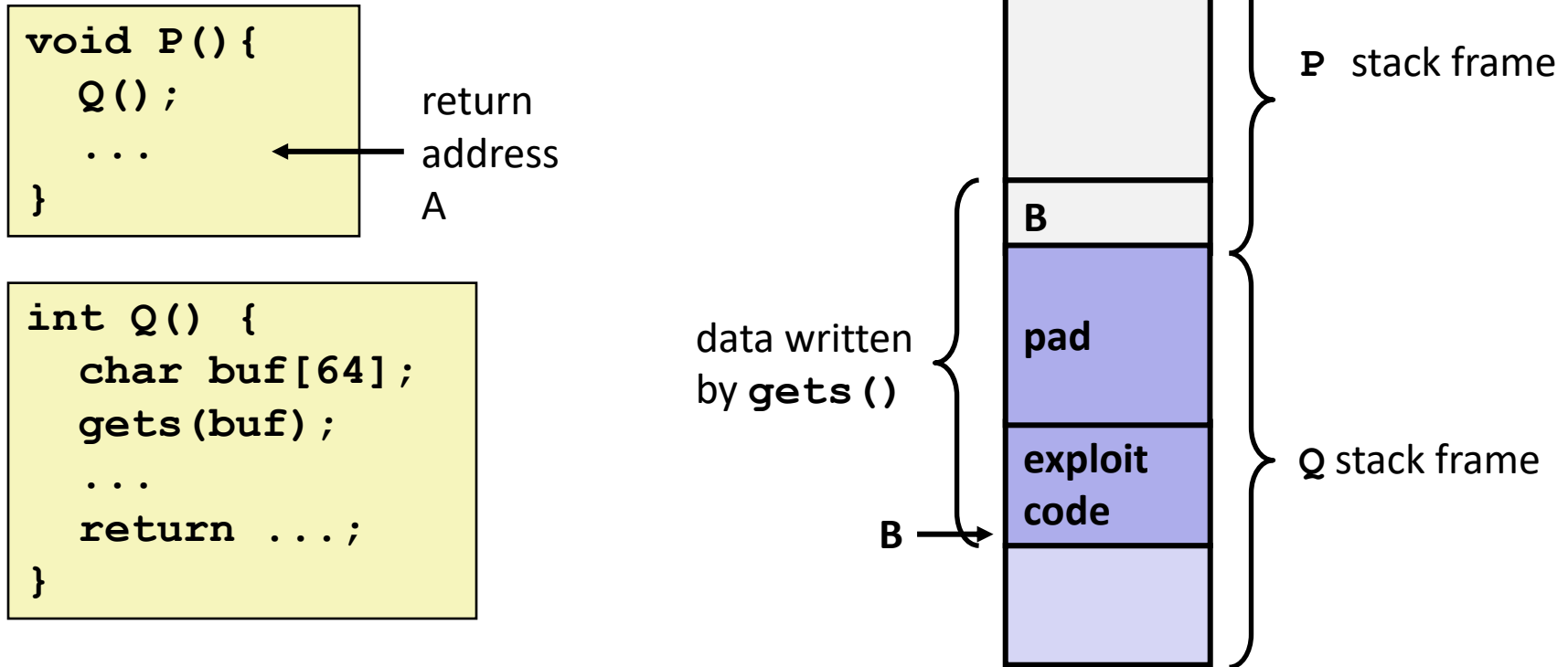
```
void smash() {
    printf("I've been smashed!\n");
    exit(0);
}
```

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

Attack String (Hex)

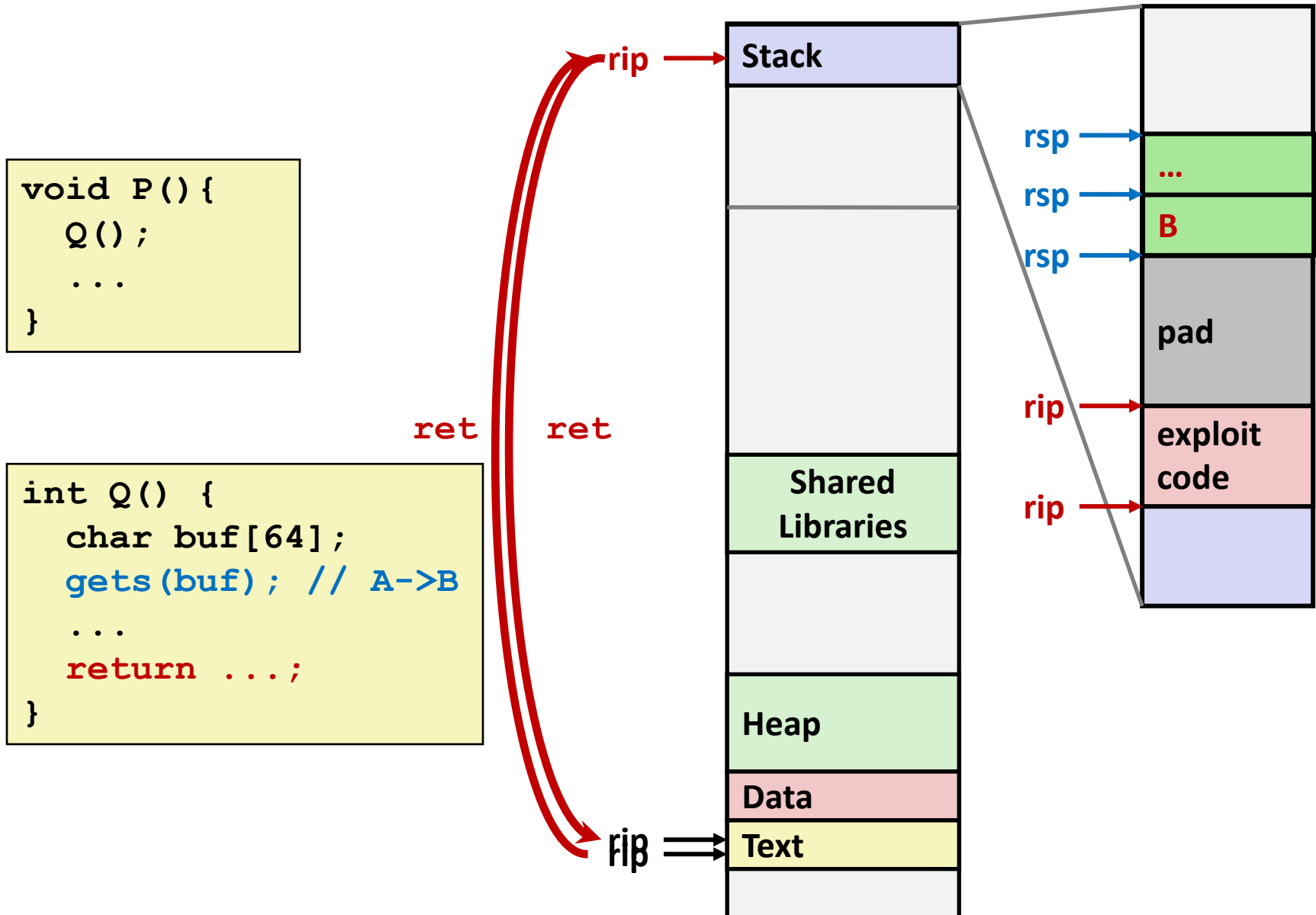
```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
fb 06 40 00 00 00 00 00 00
```

Code Injection Attacks



- 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?



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]; /* Way too small! */  
    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

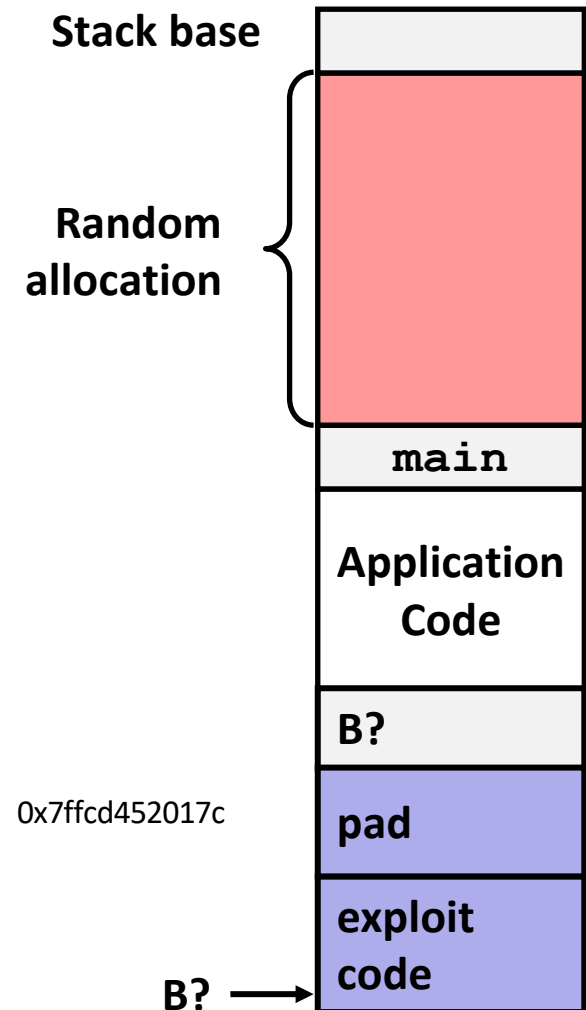
2. System-Level Protections can help

■ 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

local 0x7ffe4d3be87c 0x7fff75a4f9fc 0x7ffeadb7c80c 0x7ffeaea2fdac 0x7ffcd452017c

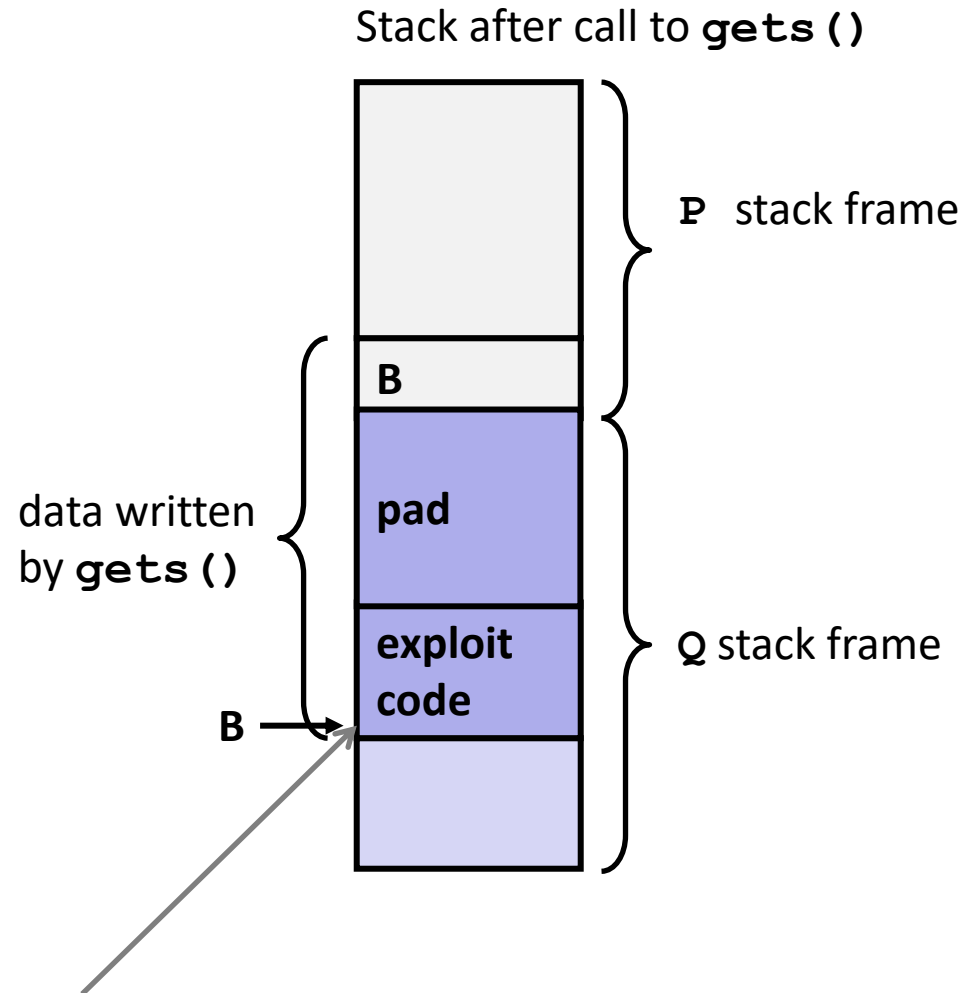
- Stack repositioned each time program executes



2. System-Level Protections can help

■ Nonexecutable 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



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`
- Now the default (disabled earlier)

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

```
unix>./bufdemo-sp  
Type a string:01234567  
*** stack smashing detected ***
```

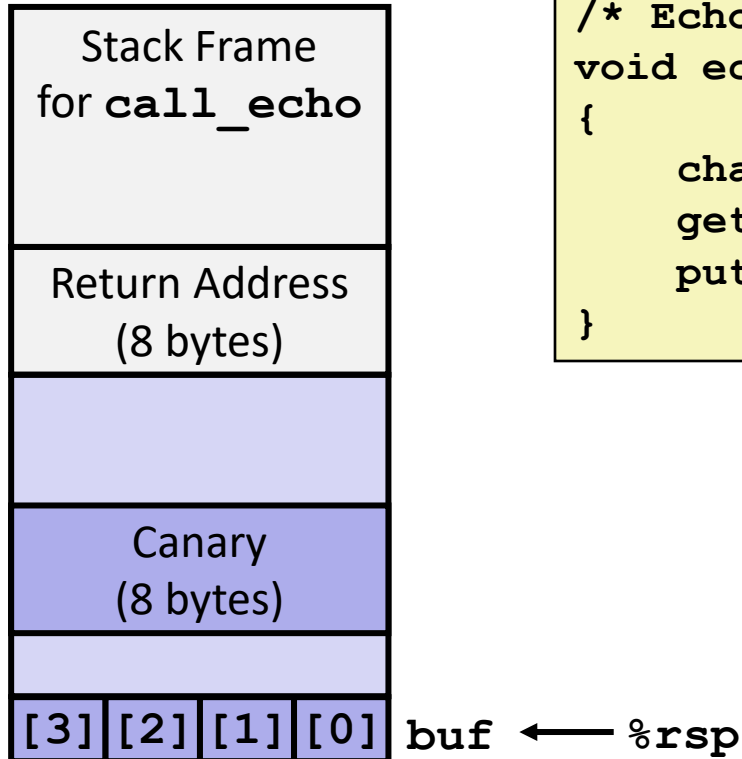
Protected Buffer Disassembly

echo:

```
40072f:  sub    $0x18,%rsp
400733:  mov    %fs:0x28,%rax
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
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:
    . . .
    movq    %fs:40, %rax    # Get canary
    movq    %rax, 8(%rsp)  # Place on stack
    xorl    %eax, %eax     # Erase canary
    . . .
```

Checking Canary

After call to gets

Stack Frame for <code>call_echo</code>			
Return Address (8 bytes)			
Canary (8 bytes)			
00	36	35	34
33	32	31	30

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

Input: **0123456**

`buf ← %rsp`

`echo:`

```
    . . .
    movq    8(%rsp), %rax    # Retrieve from stack
    xorq    %fs:40, %rax    # Compare to canary
    je      .L6             # If same, OK
    call    __stack_chk_fail # FAIL
.L6:    . . .
```


Return-Oriented Programming Attacks

■ Challenge (for hackers)

- Stack randomization makes it hard to predict buffer location
- Marking stack nonexecutable makes it hard to insert binary code

■ Alternative Strategy

- Use existing code
 - E.g., library code from `stdlib`
- String together fragments to achieve overall desired outcome
- *Does not overcome stack canaries*

■ Construct program from *gadgets*

- Sequence of instructions ending in `ret`
 - Encoded by single byte `0xc3`
- Code positions fixed from run to run
- Code is executable

Gadget Example #1

```
long ab_plus_c  
  (long a, long b, long c)  
{  
    return a*b + c;  
}
```

```
00000000004004d0 <ab_plus_c>:  
4004d0: 48 0f af fe  imul %rsi,%rdi  
4004d4: 48 8d 04 17  lea (%rdi,%rdx,1),%rax  
4004d8: c3           retq
```

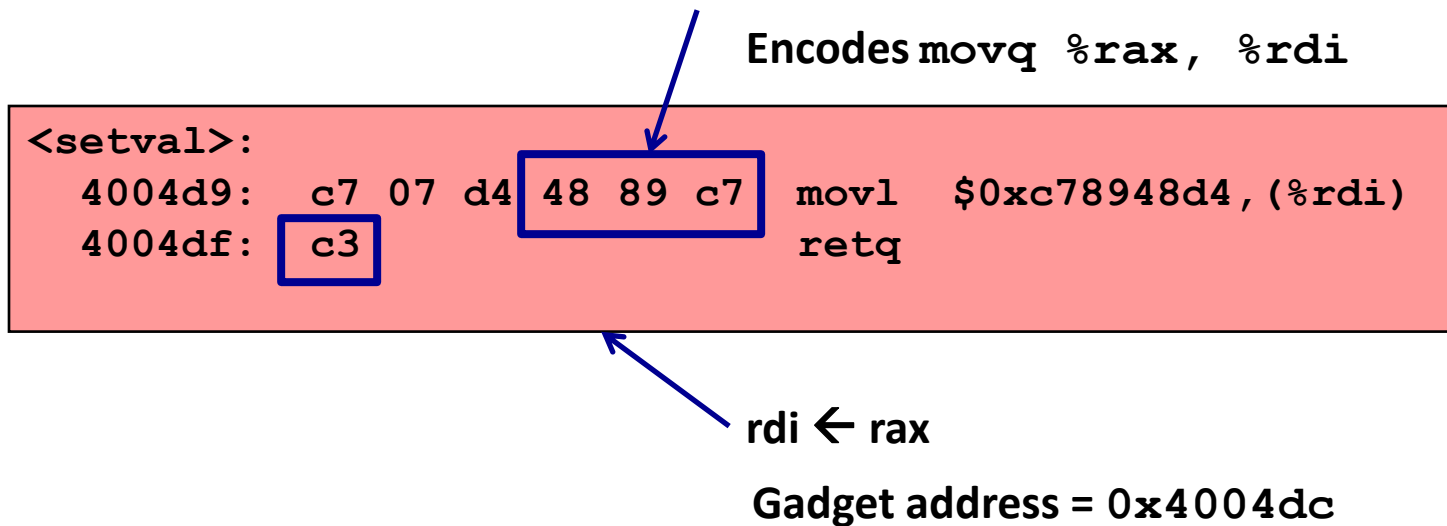
$\text{rax} \leftarrow \text{rdi} + \text{rdx}$

Gadget address = 0x4004d4

- Use tail end of existing functions

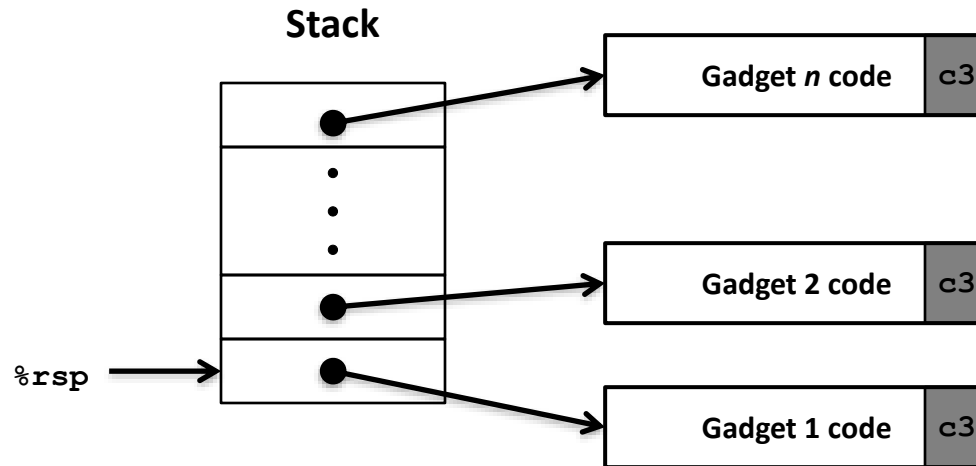
Gadget Example #2

```
void setval(unsigned *p) {
    *p = 3347663060u;
}
```



- Repurpose byte codes

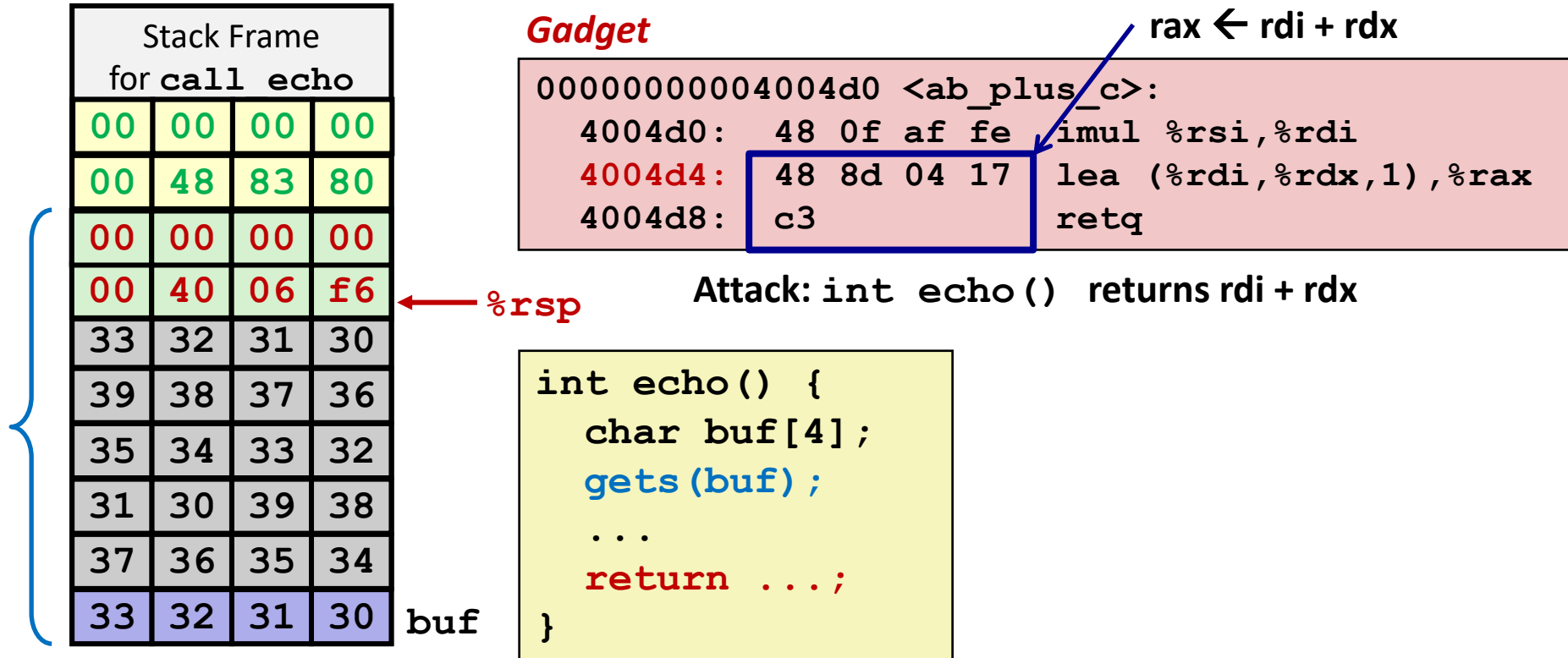
ROP Execution



- **Trigger with `ret` instruction**
 - Will start executing Gadget 1
- **Final `ret` in each gadget will start next one**

Shacham, H. (October 2007). "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)". *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*. pp. 552–561. ISBN 978-1-59593-703-2. doi:[10.1145/1315245.1315313](https://doi.org/10.1145/1315245.1315313)

Crafting an ROP Attack String



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
d4 04 40 00 00 00 00 00
  
```

Multiple gadgets will corrupt stack upwards