

Machine-Level Programming V: Advanced Topics

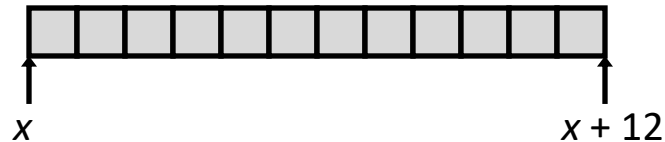
COMP400727: Introduction to Computer Systems

Danfeng Shan
Xi'an Jiaotong University

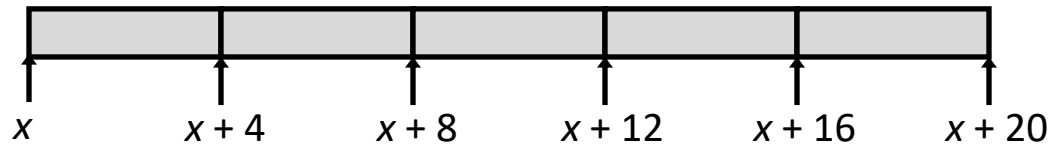
Chapter: Array Allocation

- C declaration *Type* name [*Length*] ;
 - Array of data type *Type* and length *Length*
 - Contiguously allocated region of $Length * \text{sizeof}(Type)$ bytes in memory

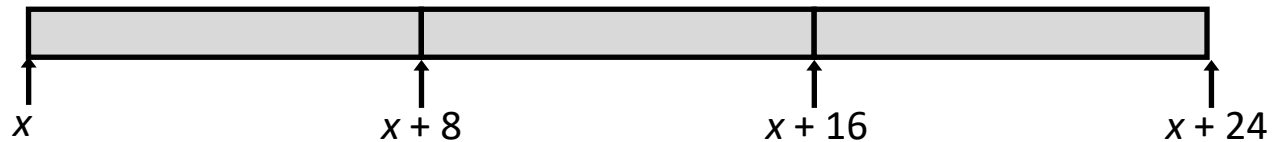
`char string[12];`



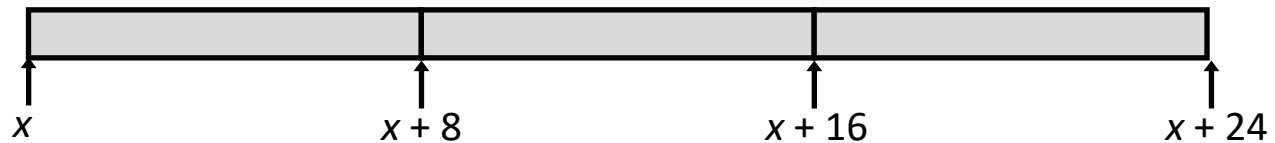
`int val[5];`



`double a[3];`



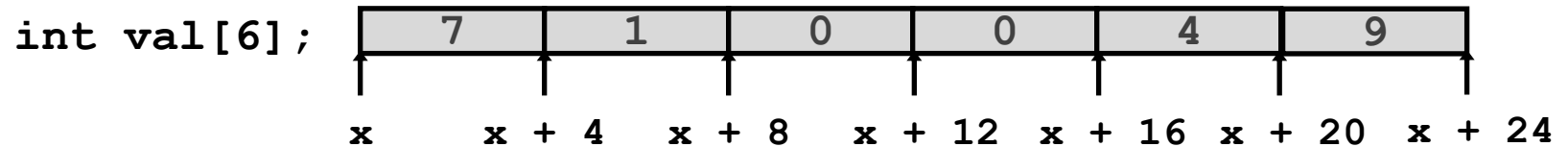
`char *p[3];`



Reminder: Array Access

■ C declaration *Type* **name** [*Length*] ;

- Array of data type *Type* and length *Length*
- Identifier **name** acts like¹ a pointer to array element 0



Expression	Type	Value	
<code>val[4]</code>	<code>int</code>	4	
<code>val[6]</code>	<code>int</code>	??	// access past end
<code>val</code>	<code>int *</code>	<code>x</code>	
<code>val+1</code>	<code>int *</code>	<code>x + 4</code>	
<code>&val[2]</code>	<code>int *</code>	<code>x + 8</code>	// same as <code>val+2</code>
<code>*(val+3)</code>	<code>int</code>	0	// same as <code>val[3]</code>
<code>val + i</code>	<code>int *</code>	<code>x + 4*i</code>	// same as <code>&val[i]</code>

¹ in most contexts (but not all)

Reminder : Nested Arrays

■ Declaration

$T \ A[R][C];$

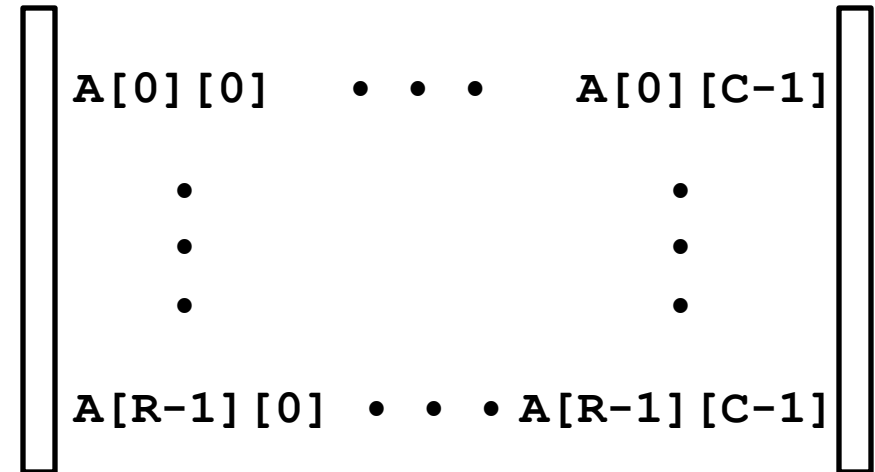
- 2D array of data type T
- R rows, C columns

■ Array Size

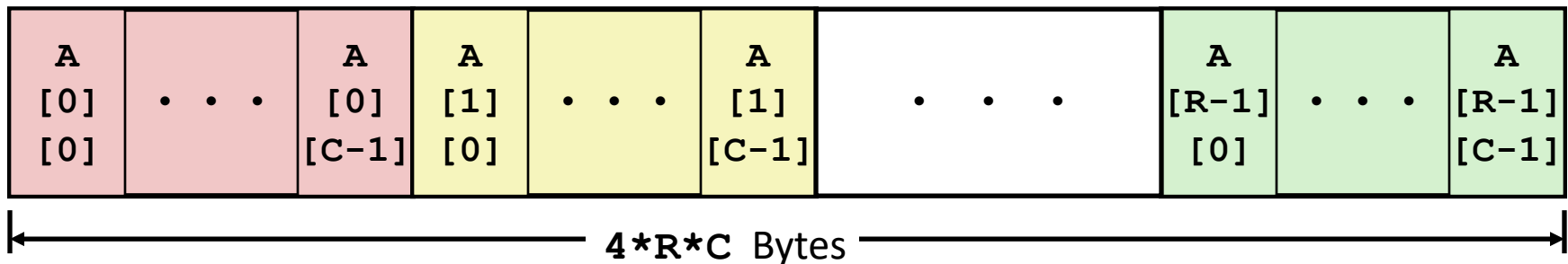
- $R * C * \text{sizeof}(T)$ bytes

■ Arrangement

- Row-Major Ordering



`int A[R][C];`

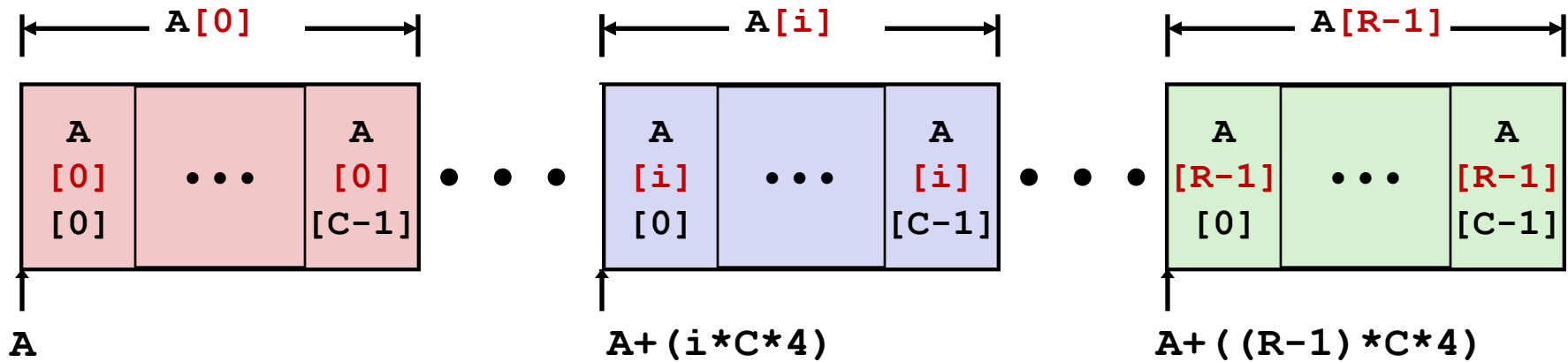


Reminder: Nested Array Row Access

■ Row Vectors

- $A[i]$ is array of C elements
- Each element of type T requires $sizeof(T)$ bytes
- Starting address $A + i * (C * sizeof(T))$

```
int A[R][C];
```

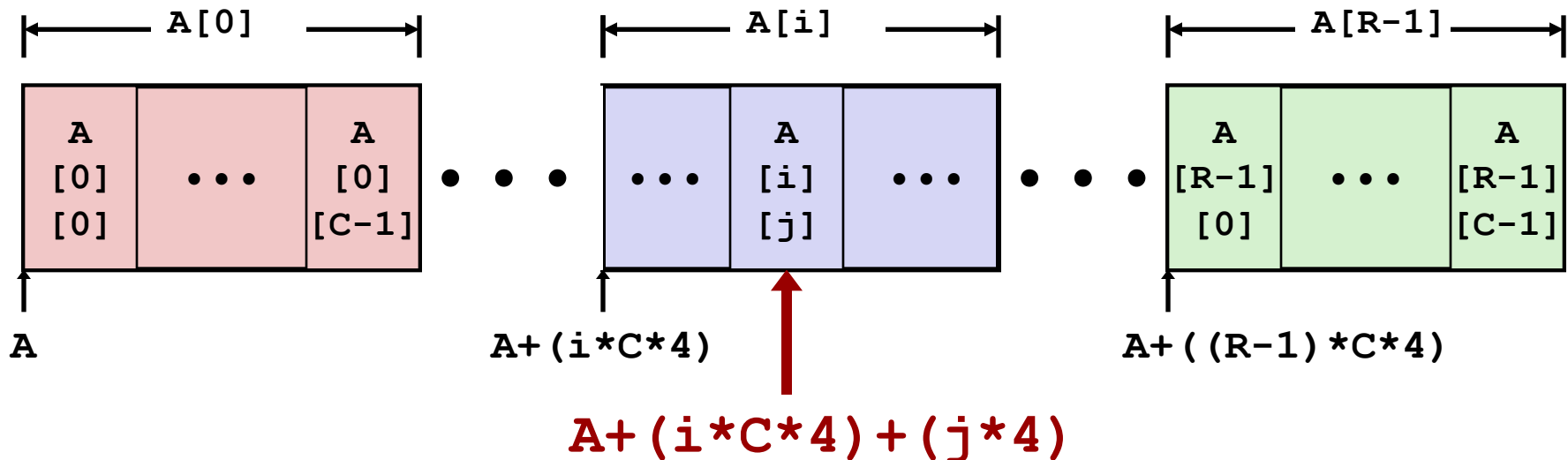


Reminder: Nested Array Element Access

■ Array Elements

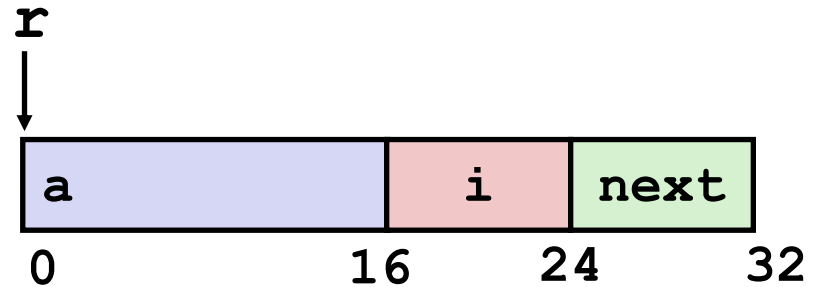
- $A[i][j]$ is element of type T , which requires $\text{sizeof}(T)$ bytes
- Address $A + i * (C * \text{sizeof}(T)) + j * \text{sizeof}(T)$
 $= A + (i * C + j) * \text{sizeof}(T)$

```
int A[R][C];
```



Structure Representation

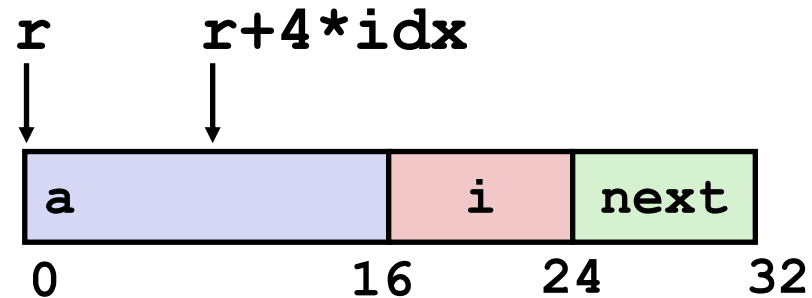
```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- Structure represented as block of memory
 - Big enough to hold all the fields
- Fields ordered according to declaration
 - Even if another ordering could be more compact
- Compiler determines overall size + positions of fields
 - In assembly, we see only offsets, not field names

Reminder: Access to Structure Member

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



■ Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Compute as $r + 4 * idx$

```
int *get_ap
(struct rec *r, size_t idx)
{
    return &r->a[idx];
}
```

```
# r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```


Reminder: Alignment with Structures

Within structure:

Must satisfy each element's alignment requirement ^{对齐}

Overall structure placement

Each structure has alignment requirement **K**

K = Largest alignment of any element

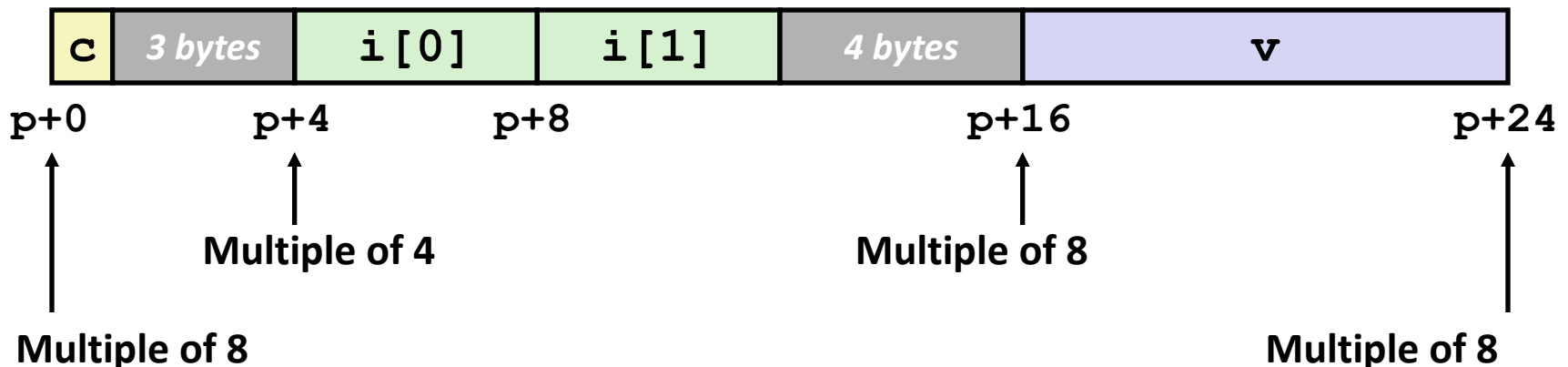
Initial address & structure length must be multiples of **K**

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

Example:

结构体的起始位置是内部数据类型max的数的倍数

K = 8, due to **double** element



Saving Space

贪心策略：实现相同情况下的总占用内存min

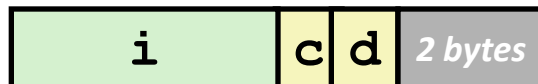
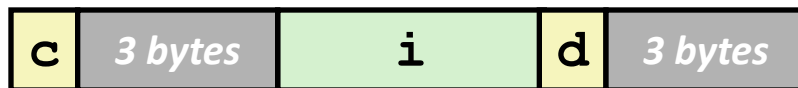
Put large data types first

```
struct S4 {
    char c;
    int i;
    char d;
} *p;
```



```
struct S5 {
    int i;
    char c;
    char d;
} *p;
```

Effect (K=4)



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

Linux Memory Layout

not drawn to scale

$(2^{47} - 4096 =)$ 0000 7FFF FFFF F000

stack的大小是
8MB(2^{20})

stack的活动区域
=> stack 随机化

0000 7FFF F800 0000

内存映射
在linker中会实行

malloc分配策略:

- 将小的内存集中分配在一起

- 将大的内存集中分配在一起

new/malloc分配的空间
定位在此

程序的机器指令...

Hex Address

40 0000

1000

0000

Stack

- Runtime stack (8MB limit)
- e.g., local variables

Heap

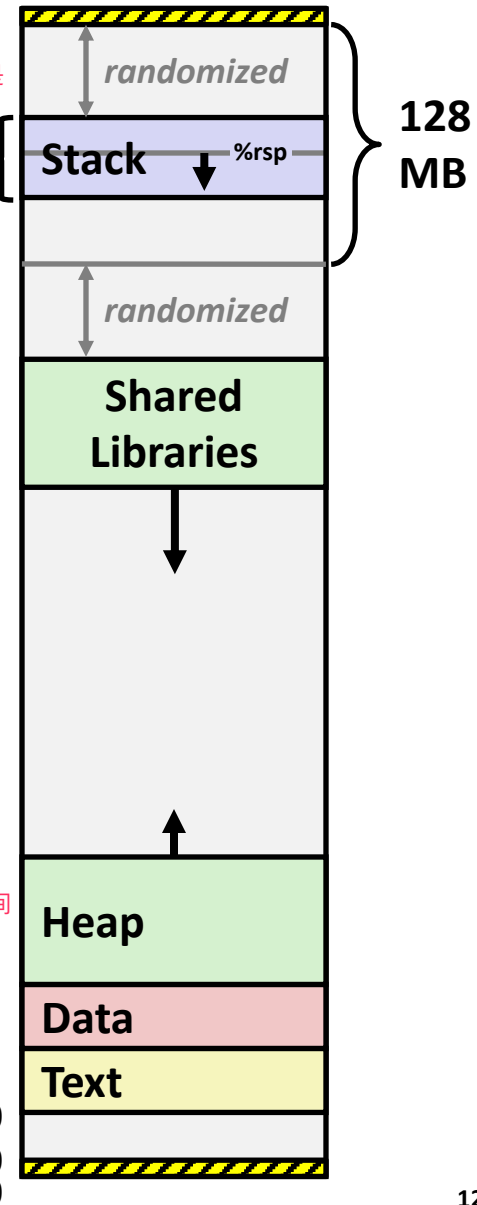
- Dynamically allocated as needed
- When call `malloc()`, `calloc()`, `new()`

Data

- Statically allocated data
- e.g., global vars, `static` vars, string constants

Text / Shared Libraries

- Executable machine instructions
- Read-only



Memory Allocation Example

not drawn to scale

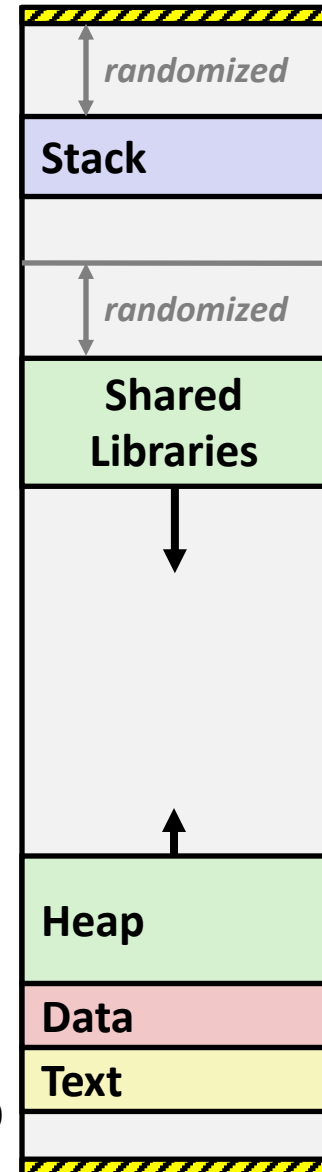
0000 7FFF FFFF F000

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



40 0000

Where does everything go?

not drawn to scale

Example Addresses

0000 7FFF FFFF F000

address range $\sim 2^{47}$

```

local
phuge1
phuge3
psmall4
psmall2
big_array
huge_array
main()
useless()

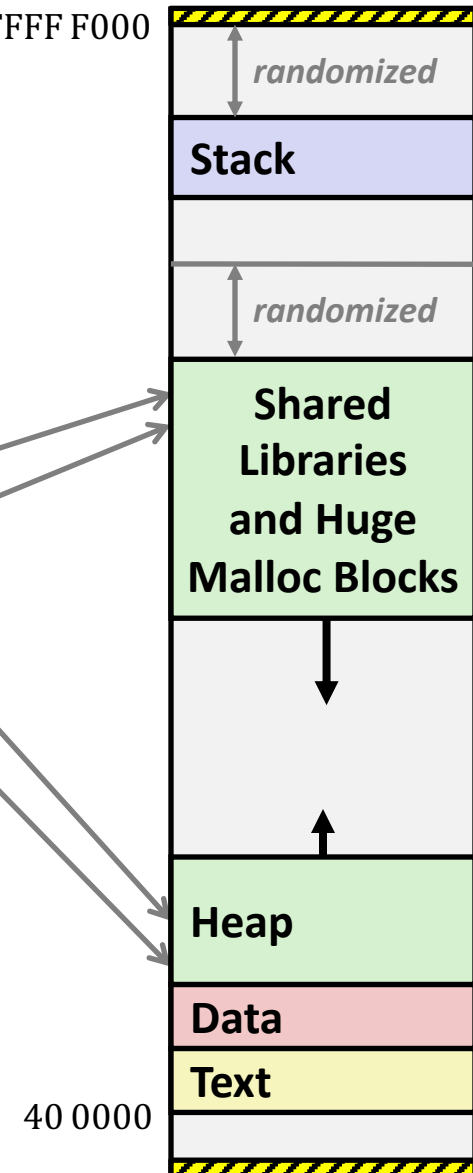
```

```

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

```

(Exact values can vary)



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

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

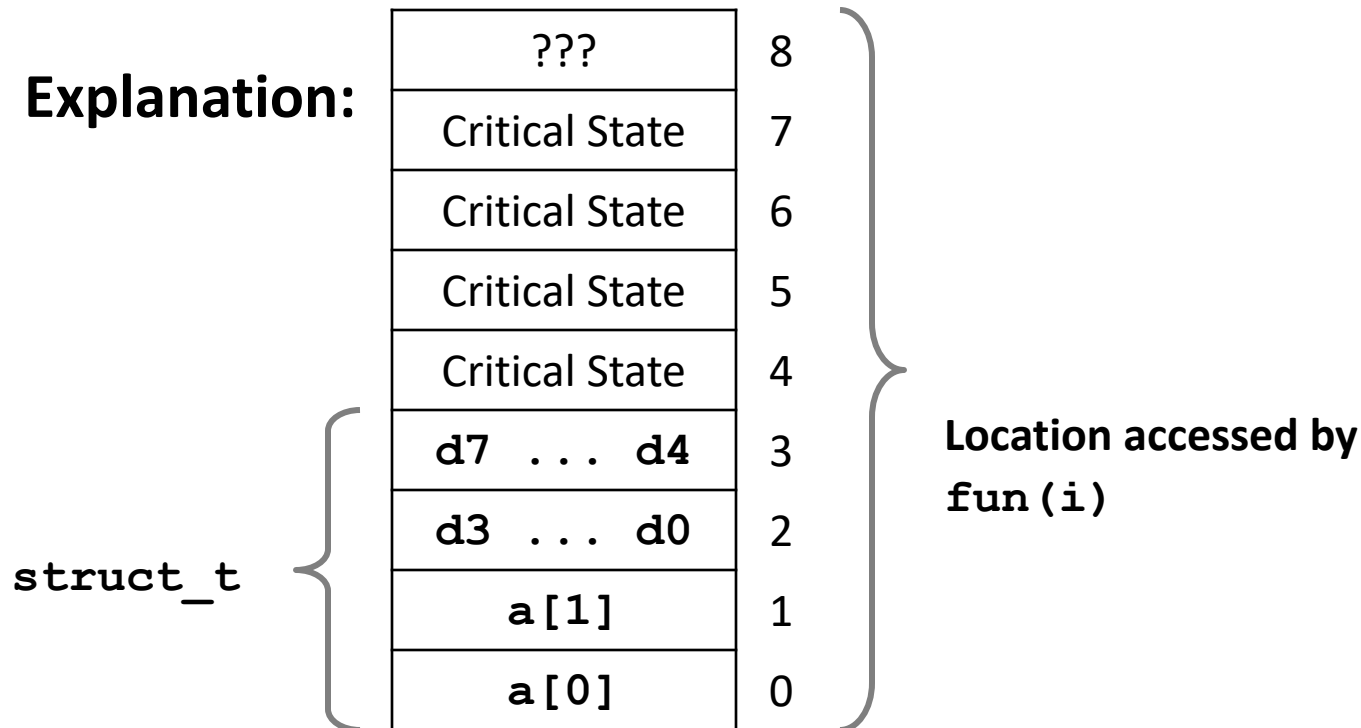
- Result is system specific

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

Explanation:



Buffer 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
- **Most common form**
 - Unchecked lengths on string inputs
 - Particularly for bounded character arrays on the stack
 - sometimes referred to as stack smashing

Library Code

■ Implementation of Unix function gets ()

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar(); 获取一个字符
    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); 从键盘缓冲区读入  
    puts(buf); 从键盘缓冲区输出  
}
```

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

```
unix>./bufdemo-nsp  
Type a string: 01234567890  
01234567890
```

```
unix>./bufdemo-nsp  
Type a string: 012345678901  
012345678901  
Segmentation Fault
```

Buffer Overflow Disassembly

echo:

0000000000001159 <echo>:

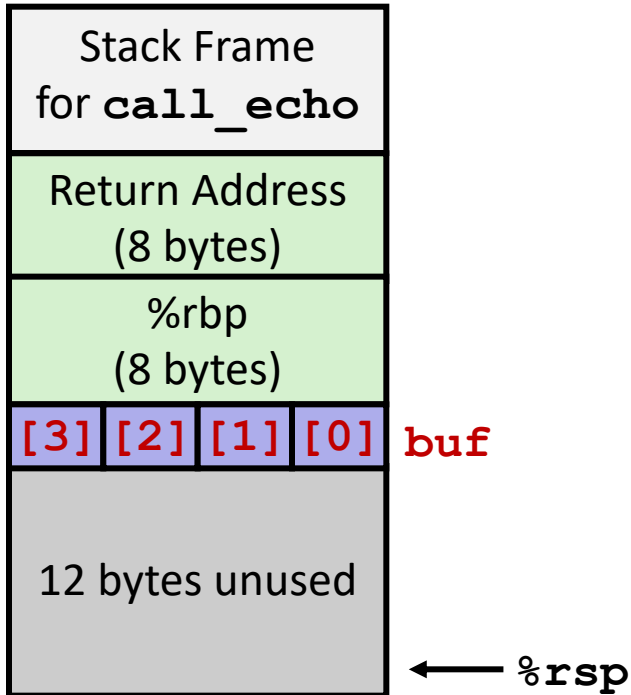
1159: 53	push	%rbx	callee saved
115a: 48 83 ec 10	sub	\$0x10, %rsp	栈指针下移16位
115e: 48 8d 5c 24 0c	lea	0xc(%rsp), %rbx	
1163: 48 89 df	mov	%rbx, %rdi	
1166: b8 00 00 00 00	mov	\$0x0, %eax	
116b: e8 d0 fe ff ff	call	1040 <gets@plt>	
1170: 48 89 df	mov	%rbx, %rdi	
1173: e8 b8 fe ff ff	call	1030 <puts@plt>	
1178: 48 83 c4 10	add	\$0x10, %rsp	
117c: 5b	pop	%rbx	
117d: c3	ret		

call_echo:

117e: 48 83 ec 08	sub	\$0x8, %rsp
1182: b8 00 00 00 00	mov	\$0x0, %eax
1187: e8 c5 ff ff ff	callq	1189 <echo>
118c: 48 83 c4 08	add	\$0x8, %rsp
1190: c3	ret	

Buffer Overflow Stack Example

Before call to gets

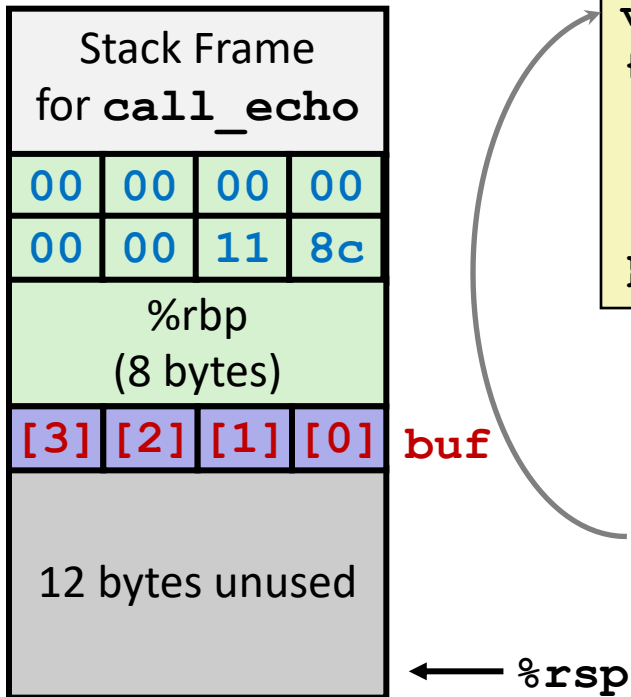


```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    push    %rbx
    subq    $0x10, %rsp
    lea     0xc(%rsp), %rbx
    mov     %rbx, %rdi
    mov     $0x0, %eax
    call    gets
    . . .
```

Buffer Overflow Stack Example

Before call to gets



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

```
echo:
    push    %rbx
    subq    $0x10, %rsp
    lea     0xc(%rsp), %rbx
    mov     %rbx, %rdi
    mov     $0x0, %eax
    call    gets
    . . .
```

call_echo:

```
. . .
1187:    callq   4006cf <echo>
118c:    add     $0x8, %rsp
. . .
```

Buffer Overflow Stack Example #1

After call to gets

Stack Frame for call_echo			
00	00	00	00
00	00	11	8c
00	30	39	38
37	36	35	34
33	32	31	30
12 bytes unused			

buf

← %rsp

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

```
echo:
    push    %rbx
    subq    $0x10, %rsp
    lea     0xc(%rsp), %rbx
    mov     %rbx, %rdi
    mov     $0x0, %eax
    call    gets
    . . .
```

call_echo:

```
. . .
1187:    callq   4006cf <echo>
118c:    add     $0x8, %rsp
. . .
```

```
unix> ./bufdemo-nsp
Type a string: 01234567890
01234567890
```

```
"01234567890\0"
```

Overflowed buffer, but did not corrupt state

Buffer Overflow Stack Example #1

After call to gets

Stack Frame for call_echo			
00	00	00	00
00	00	11	00
31	30	39	38
37	36	35	34
33	32	31	30
12 bytes unused			

buf

← %rsp

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

```
echo:
    push    %rbx
    subq    $0x10, %rsp
    lea     0xc(%rsp), %rbx
    mov     %rbx, %rdi
    mov     $0x0, %eax
    call    gets
    . . .
```

call_echo:

```
. . .
1187:    callq   4006cf <echo>
118c:    add     $0x8, %rsp
. . .
```

```
unix> ./bufdemo-nsp
Type a string: 012345678901
012345678901
segmentation fault
```

Program “returned” to 0x1100, and then crashed.

Code Injection Attacks

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

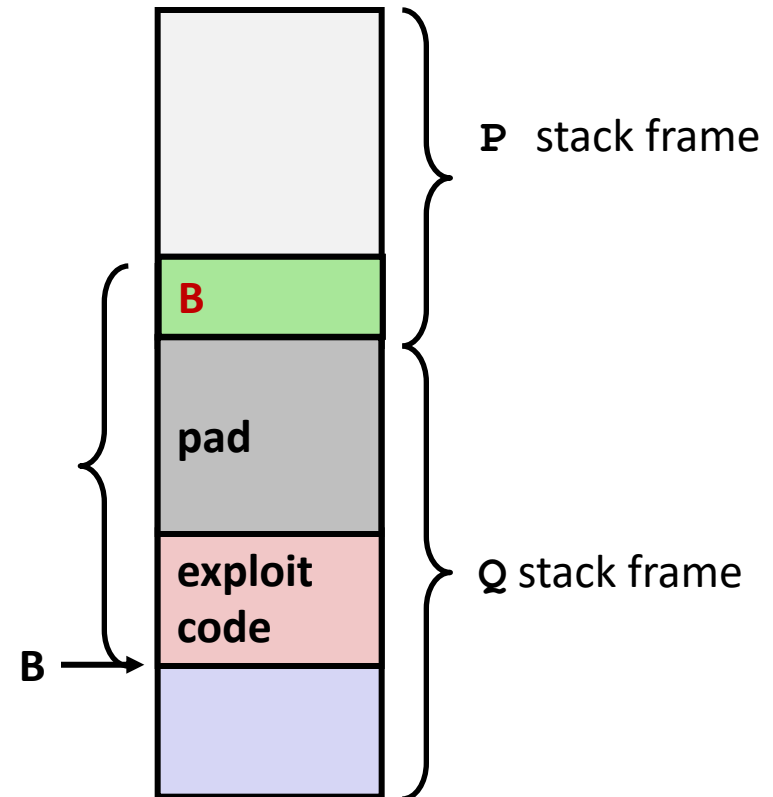
return
address
A

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

自行修改返回地址

data written
by gets ()

Stack after call to gets ()



- 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

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)
 - “IM wars” (1999)
 - Twilight hack on Wii (2000s)
 - ... and many, many more
- **You will learn some of the tricks in attacklab**
 - Hopefully to convince you to never leave such holes in your programs!!

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 dfshan@cs.xjtu.edu.cn`
- Worm attacked fingerd server by sending phony argument:
 - `finger "exploit-code padding new-return-address"`
 - exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

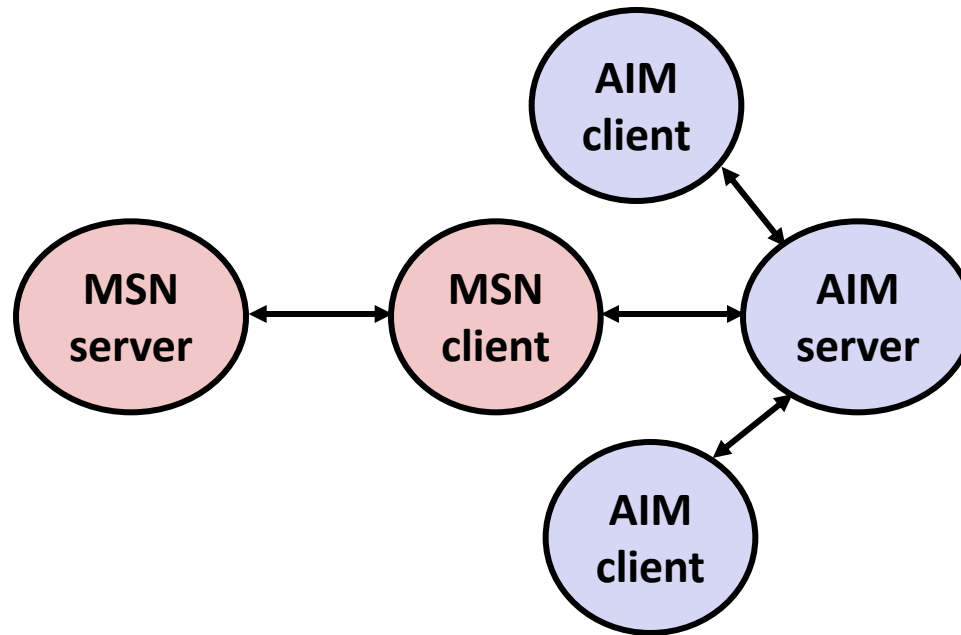
■ Once on a machine, scanned for other machines to attack

- 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...
- and CERT was formed... still homed at CMU

Example 2: IM War

■ July, 1999

- Microsoft launches MSN Messenger (instant messaging system).
- Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



(cont.)

■ August 1999

- Mysteriously, Messenger clients can no longer access AIM servers
- Microsoft and AOL begin the IM war:
 - AOL changes server to disallow Messenger clients
 - Microsoft makes changes to clients to defeat AOL changes
 - At least 13 such skirmishes
- What was really happening?
 - AOL had discovered a buffer overflow bug in their own AIM clients
 - They exploited it to detect and block Microsoft: the exploit code returned a 4-byte signature (the bytes at some location in the AIM client) to server
 - When Microsoft changed code to match signature, AOL changed signature location



Date: Aug 1999 11:30:57 -0700 (PDT)
From: Phil Bucking <philbucking@yahoo.com>

Subject: AOL exploiting buffer overrun bug in their own software!

To: rms@pharlap.com

Mr. Smith,

I am writing you because I have discovered something that I think you might find interesting because you are an Internet security expert with experience in this area. I have also tried to contact AOL but received no response.

I am a developer who has been working on a revolutionary new instant messaging client that should be released later this year.

...

It appears that the AIM client has a buffer overrun bug. By itself this might not be the end of the world, as MS surely has had its share. But AOL is now *exploiting their own buffer overrun bug* to help in its efforts to block MS Instant Messenger.

....

Since you have significant credibility with the press I hope that you can use this information to help inform people that behind AOL's friendly exterior they are nefariously compromising peoples' security.

Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com

***It was later determined that this
email originated from within
Microsoft!***

- 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

System-Level Protections Can Help

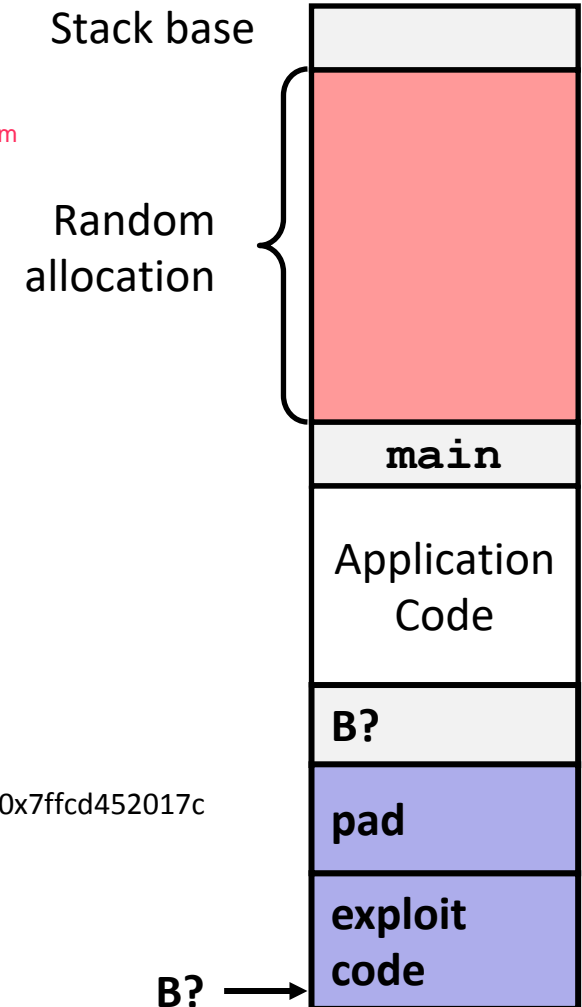
■ Randomized stack offsets

- 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

ASLR
Address Stack Layout Random

local 0x7ffe4d3be87c 0x7fff75a4f9fc 0x7ffeadb7c80c 0x7ffeaea2fdac 0x7ffcd452017c

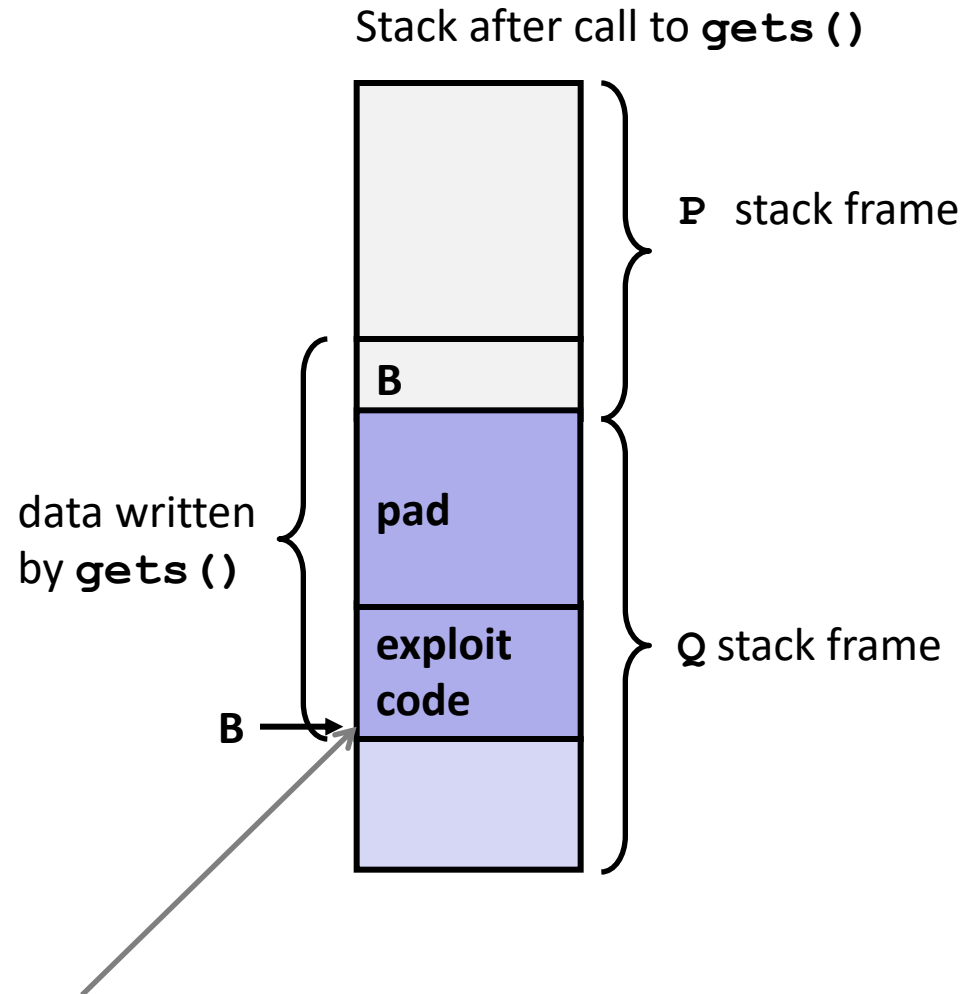
- Stack repositioned each time program executes



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

Canaries Can Help

■ Idea

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

■ GCC Implementation

canary的值每一次执行都会随机化

- **-fstack-protector**
- Now the default (disabled earlier)

```
unix> ./bufdemo-sp  
Type a string: 0123  
0123456
```

```
unix> ./bufdemo-sp  
Type a string: 01234  
*** stack smashing detected ***
```

Protected Buffer Disassembly

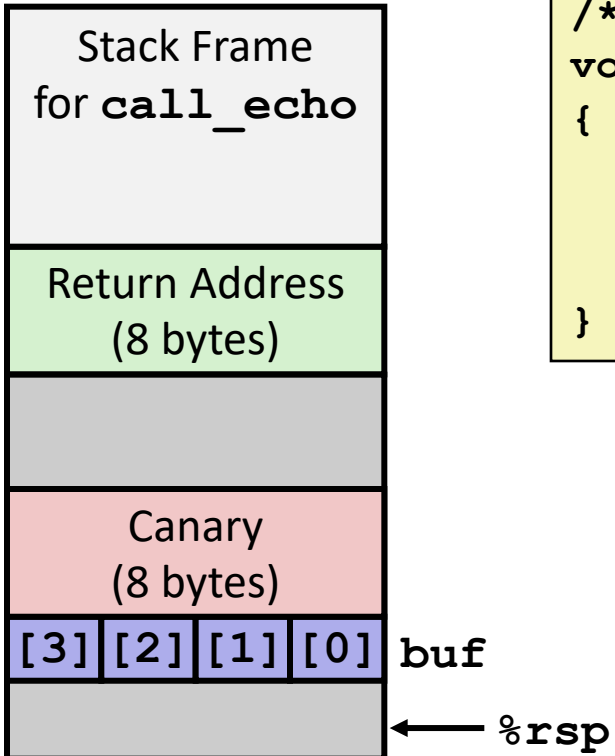
echo:

```

1169:    push    %rbx
116a:    sub     $0x10,%rsp
116e:    mov     %fs:0x28,%rax
1177:    mov     %rax,0x8(%rsp)  加入canary后新增的是“red line”
117c:    xor     %eax,%eax
117e:    lea     0x4(%rsp),%rbx
1183:    mov     %rbx,%rdi
1186:    callq   1050 <gets@plt>
118b:    mov     %rbx,%rdi
118e:    call    1030 <puts@plt>
1193:    mov     0x8(%rsp),%rax
1198:    sub     %fs:0x28,%rax
11a1:    jne     11a9 <echo+0x40>
11a3:    add     $0x10,%rsp
11a7:    pop     %rbx
11a8:    ret
11a9:    call    1040 <__stack_chk_fail@plt>
  
```

Up Canary

Before call to gets



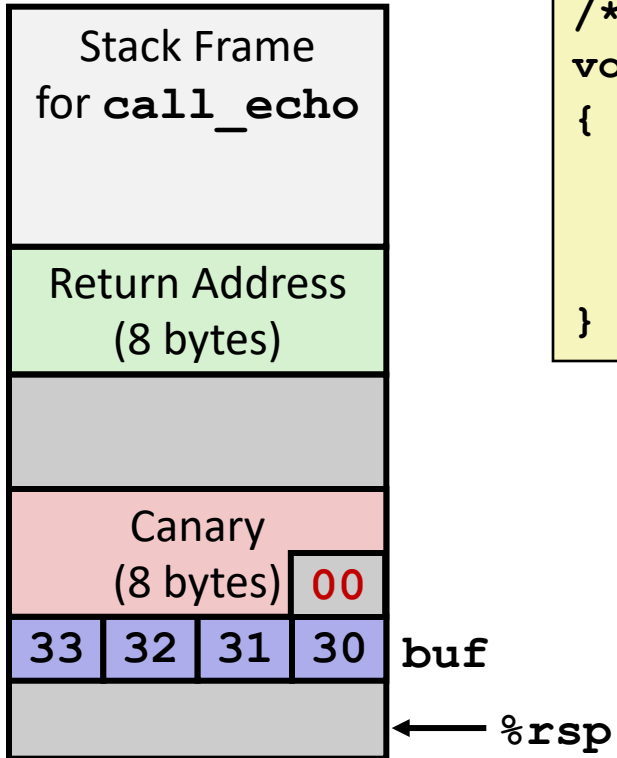
```
/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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: 0123

LSB of canary is 0x00

```
echo:
. . .
1193    mov     0x8(%rsp),%rax    # Retrieve from stack
1198    sub     %fs:0x28,%rax     # Compare to canary
11a1    jne     11a9 <echo+0x40> # If not same, error
. . .
11a9    call    __stack_chk_fail # FAIL
```


Return-Oriented Programming Attacks

没法攻破canary的保护

但是其它的保护可以突破

■ Challenge (for hackers)

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

■ Alternative Strategy

- Use existing code
 - Part of the program or the C library
- 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
```

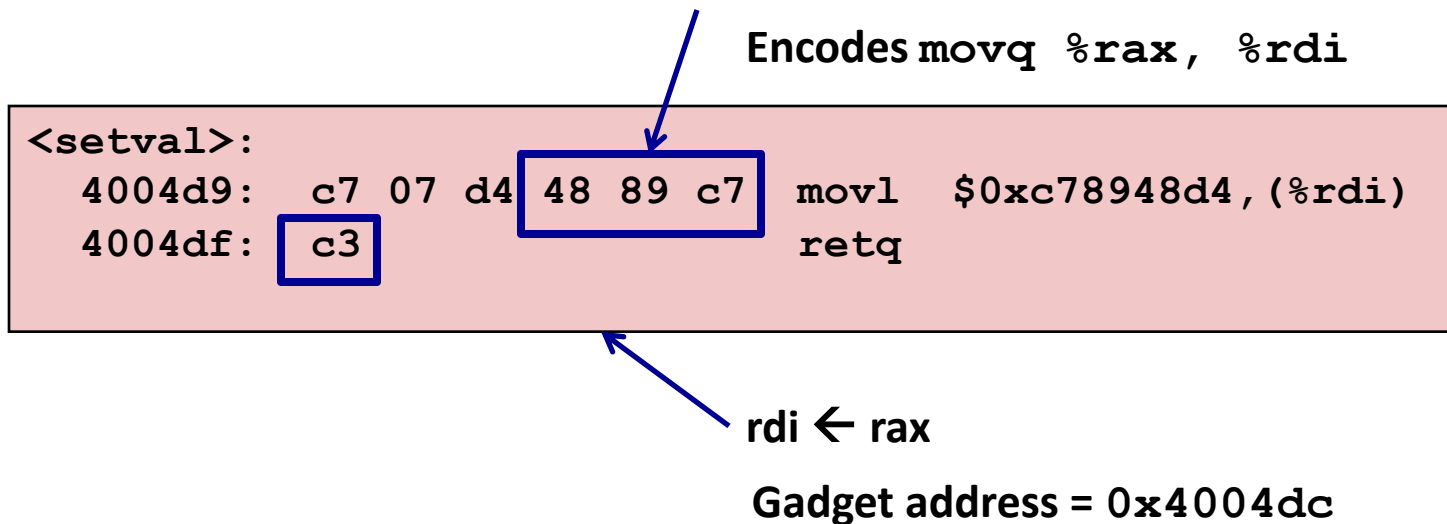
$\text{rax} \leftarrow \text{rdi} + \text{rdx}$

Gadget address = 0x4004d4

- Use tail end of existing functions

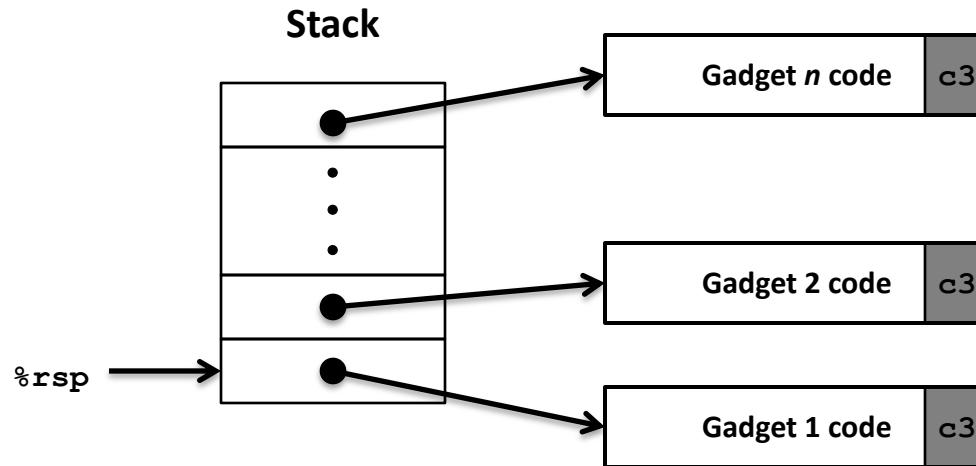
Gadget Example #2

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



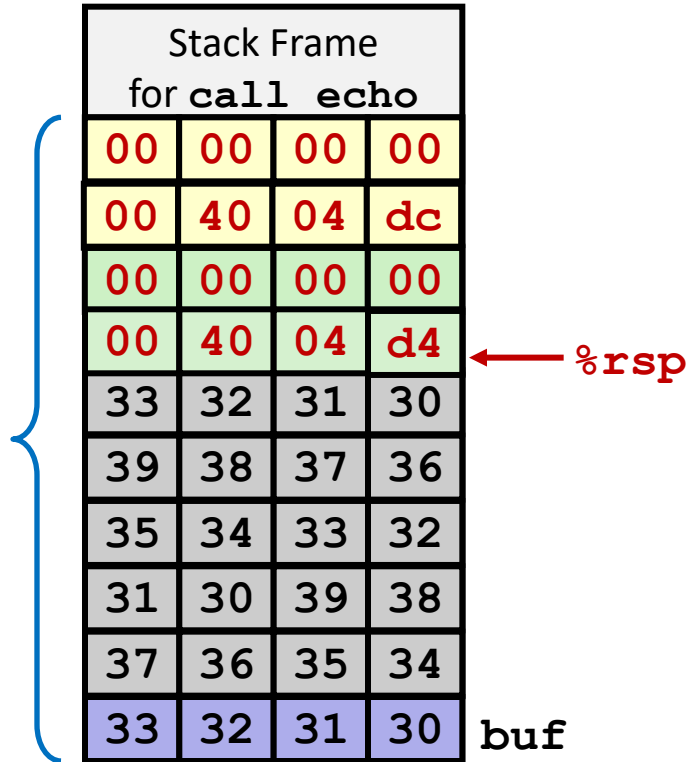
■ Repurpose byte codes

ROP Execution



- Trigger with `ret` instruction
 - Will start executing Gadget 1
- Final `ret` in each gadget will start next one
 - `ret`: pop address from stack and jump to that address

Creating an ROP Attack String



■ Gadget #1

■ `0x4004d4` $\text{rax} \leftarrow \text{rdi} + \text{rdx}$

■ Gadget #2

■ `0x4004dc` $\text{rdi} \leftarrow \text{rax}$

■ Combination

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

Attack String (Hex)

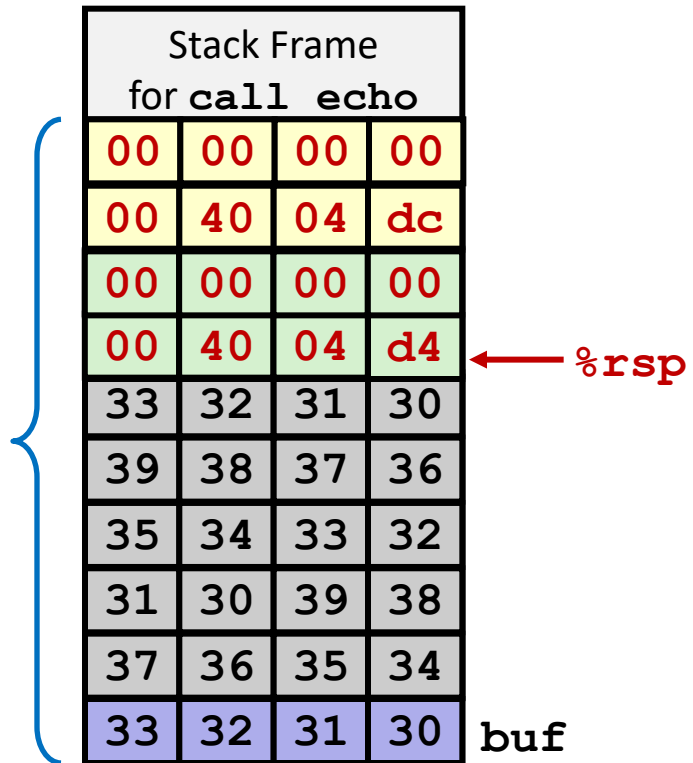
```

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 dc 04 40 00 00 00 00 00

```

Multiple gadgets will corrupt stack upwards

What Happens When echo Returns?



1. **Echo executes `ret`**
 - Starts Gadget #1
2. **Gadget #1 executes `ret`**
 - Starts Gadget #2
3. **Gadget #2 executes `ret`**
 - Goes off somewhere ...

Multiple gadgets will corrupt stack upwards

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