

计算机系统导论

Introduction to Computer System

Lec.06 – Machine-level Programming (IV)

Advanced Topics

Dr. Borui Li

Southeast University · 2025 Spring



Machine-level Programming (IV): Advanced



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



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()`

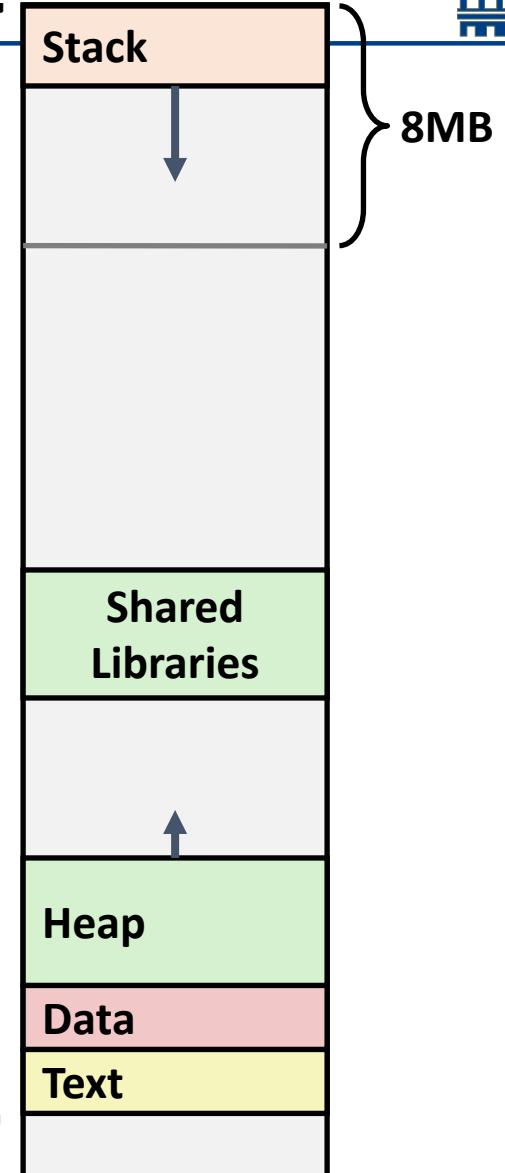
■ Data

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

■ Text / Shared Libraries

- Executable machine instructions
- Read-only

→ 400000



Memory Allocation Example

not drawn to scale

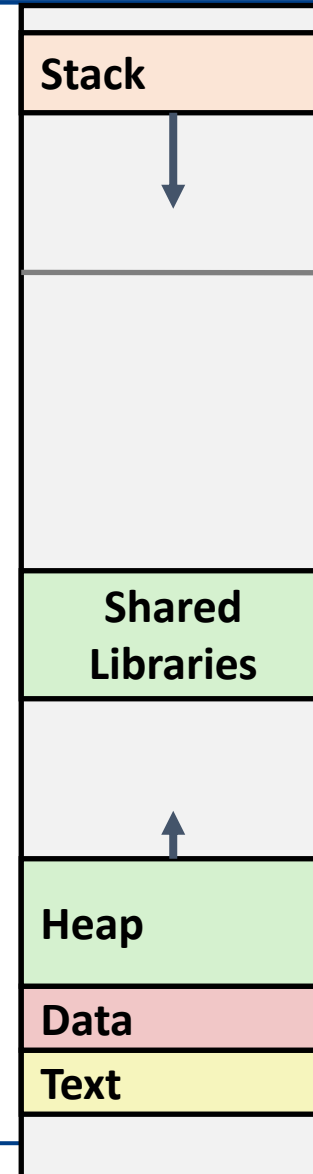


```
char big_array[1L<<24]; /* 16 MB */
char huge_array[1L<<31]; /* 2 GB */

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?



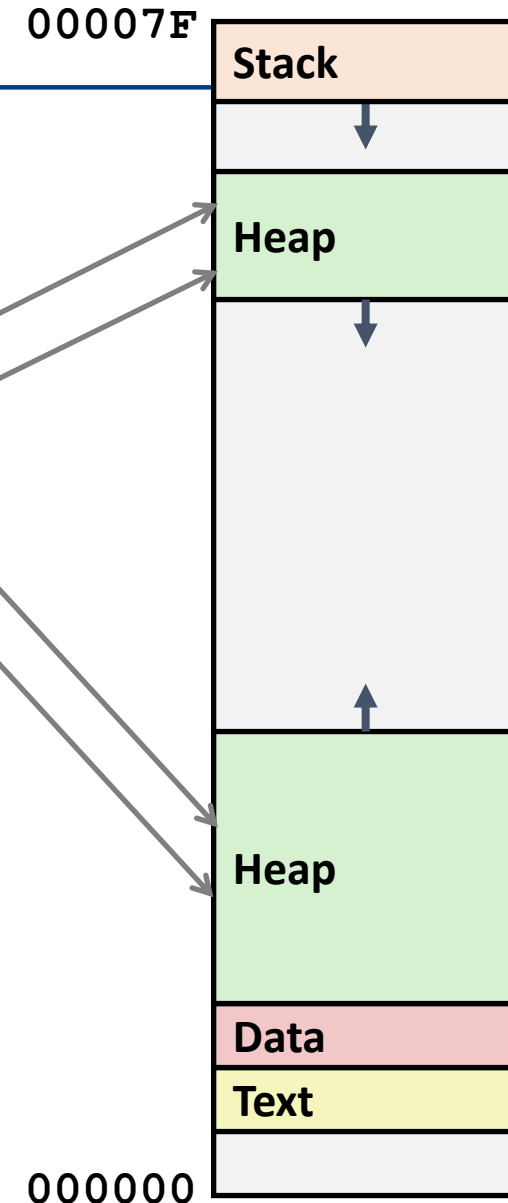
x86-64 Example Addresses



address range $\sim 2^{47}$

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590

ulimit -a



Machine-level Programming (IV): Advanced



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



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)	→	3.14
fun(1)	→	3.14
fun(2)	→	3.1399998664856
fun(3)	→	2.00000061035156
fun(4)	→	3.14
fun(6)	→	Segmentation fault

- Result is system specific



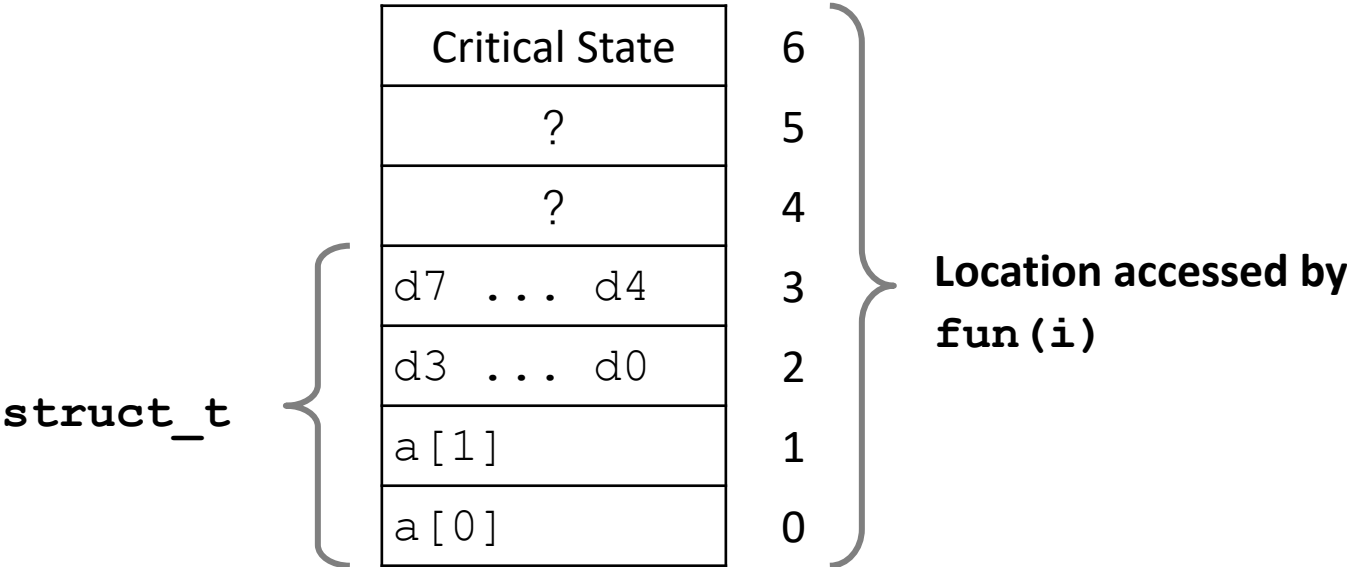
Memory Referencing Bug Example



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

- fun(0) → 3.14
- fun(1) → 3.14
- fun(2) → 3.1399998664856
- fun(3) → 2.00000061035156
- fun(4) → 3.14
- fun(6) → 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 inputs
 - Particularly for bounded character arrays on the stack
 - sometimes referred to as stack smashing



String Library Code



■ Implementation of Unix function gets ()

- 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

```
/* 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;
}
```



Vulnerable Buffer Code



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

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

← btw, how big
is big enough?

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

```
unix> ./bufdemo-nsp  
Type a string: 0123456789012345678901234  
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:

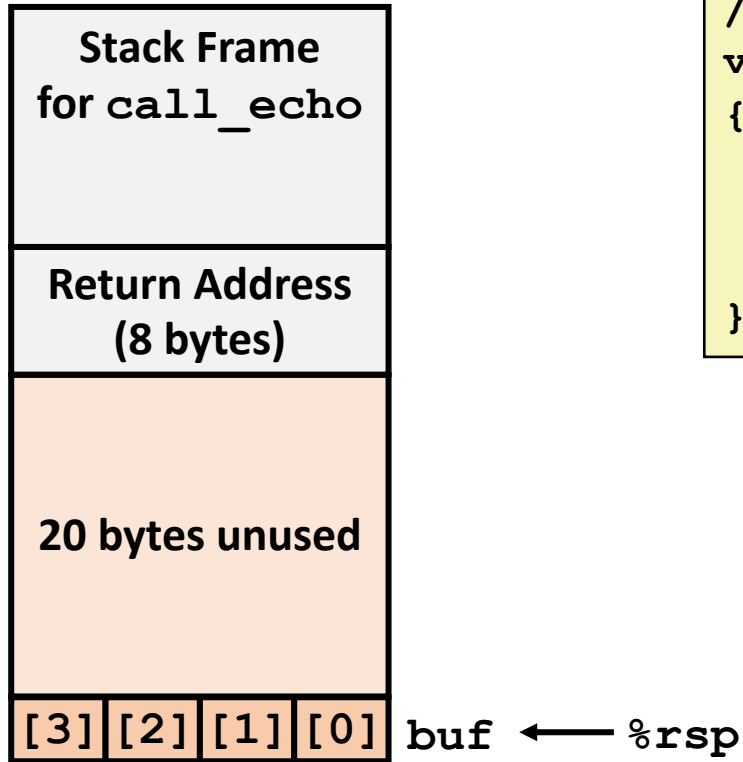
4006e8:	48 83 ec 08	sub	\$0x8,%rsp
4006ec:	b8 00 00 00 00	mov	\$0x0,%eax
4006f1:	e8 d9 ff ff ff	callq	4006cf <echo>
4006f6:	48 83 c4 08	add	\$0x8,%rsp
4006fa:	c3	retq	



Buffer Overflow Stack



Before call to gets



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

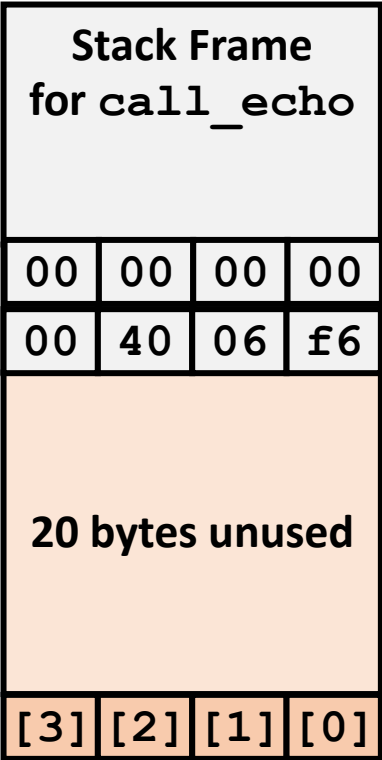
```
echo:  
    subq    $24, %rsp  
    movq    %rsp, %rdi  
    call    gets  
    . . .
```



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 call_echo			
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:

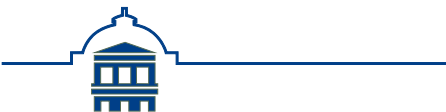
```
. . .  
4006f1:    callq   4006cf <echo>  
4006f6:    add     $0x8,%rsp  
. . .
```

Overflowed buffer, but did not corrupt state

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



Buffer Overflow Stack Example #2



After call to gets

Stack Frame for call_echo			
00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	29	28
27	26	25	24
23	22	21	20

```
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  
    . . .
```

buf ← %rsp

Overflowed buffer and corrupted return pointer

```
unix> ./bufdemo-nsp  
Type a string: 0123456789012345678901234  
Segmentation Fault
```



Buffer Overflow Stack Example #3



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    $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
```

Overflowed buffer, corrupted return pointer, but program seems to work!



Buffer Overflow Stack Example #3 Explained



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

register_tm_clones:

. . .		
400600:	mov	%rsp,%rbp
400603:	mov	%rax,%rdx
400606:	shr	\$0x3f,%rdx
40060a:	add	%rdx,%rax
40060d:	sar	%rax
400610:	jne	400614
400612:	pop	%rbp
400613:	retq	

“Returns” to unrelated code
Lots of things happen, without modifying critical state
Eventually executes `retq` back to `main`



Code Injection Attacks

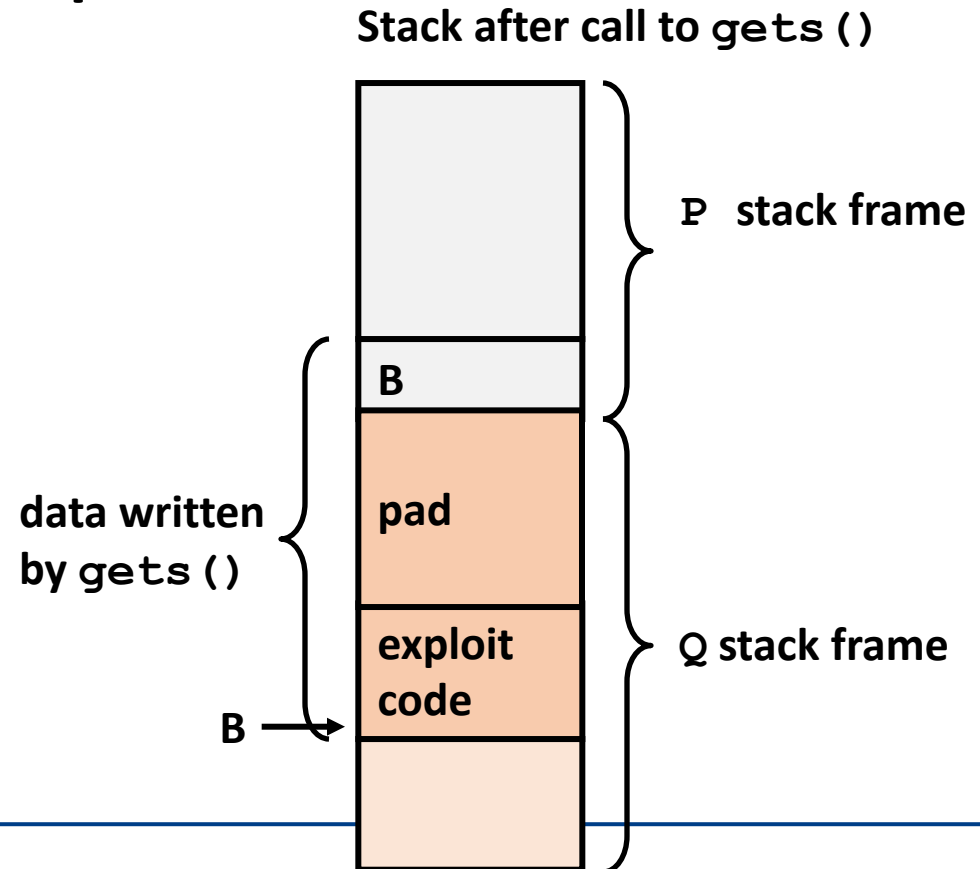


- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer B
- When Q executes `ret`, will jump to exploit code

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

return
address
A

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



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)
 - “IM wars” (1999)
 - Twilight hack on Wii (2000s)
 - ... and many, many more



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 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

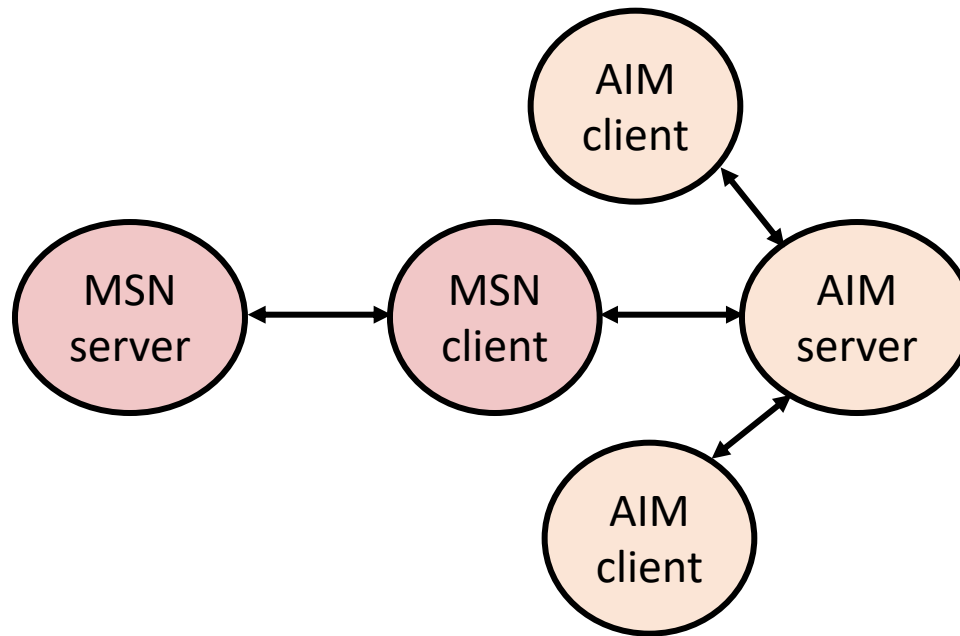


Example 2: IM War



■ July, 1999

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



IM War (cont.)



■ August 1999

- Mysteiously, 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: Wed, 11 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!



Aside: Worms and Viruses



- **Worm: A program that**

- Can run by itself
- Can propagate a fully working version of itself to other computers

- **Virus: Code that**

- Adds itself to other programs
- Does not run independently

- **Both are (usually) designed to spread among computers and to wreak havoc**



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 (!)



- 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

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





函数	严重性	解决方案
gets	最危险	使用 fgets (buf, size, stdin) 。这几乎总是一个大问题!
strcpy	很危险	改为使用 strncpy。
strcat	很危险	改为使用 strncat。
sprintf	很危险	改为使用 snprintf, 或者使用精度说明符。
scanf	很危险	使用精度说明符, 或自己进行解析。
sscanf	很危险	使用精度说明符, 或自己进行解析。
fscanf	很危险	使用精度说明符, 或自己进行解析。
vfscanf	很危险	使用精度说明符, 或自己进行解析。
vsprintf	很危险	改为使用 vsnprintf, 或者使用精度说明符。
vscanf	很危险	使用精度说明符, 或自己进行解析。
vsscanf	很危险	使用精度说明符, 或自己进行解析。
streadd	很危险	确保分配的目的地参数大小是源参数大小的四倍。
strecpy	很危险	确保分配的目的地参数大小是源参数大小的四倍。
strtrns	危险	手工检查来查看目的地大小是否至少与源字符串相等。
realpath	很危险 (或稍小, 取决于实现)	分配缓冲区大小为 MAXPATHLEN。同样, 手工检查参数以确保输入参数不超过 MAXPATHLEN。





syslog	很危险（或稍小，取决于实现）	在将字符串输入传递给该函数之前，将所有字符串输入截成合理的大小。
getopt	很危险（或稍小，取决于实现）	在将字符串输入传递给该函数之前，将所有字符串输入截成合理的大小。
getopt_long	很危险（或稍小，取决于实现）	在将字符串输入传递给该函数之前，将所有字符串输入截成合理的大小。
getpass	很危险（或稍小，取决于实现）	在将字符串输入传递给该函数之前，将所有字符串输入截成合理的大小。
getchar	中等危险	如果在循环中使用该函数，确保检查缓冲区边界。
fgetc	中等危险	如果在循环中使用该函数，确保检查缓冲区边界。
getc	中等危险	如果在循环中使用该函数，确保检查缓冲区边界。
read	中等危险	如果在循环中使用该函数，确保检查缓冲区边界。
bcopy	低危险	确保缓冲区大小与它所说的一样大。
fgets	低危险	确保缓冲区大小与它所说的一样大。
memcpy	低危险	确保缓冲区大小与它所说的一样大。
snprintf	低危险	确保缓冲区大小与它所说的一样大。
strccpy	低危险	确保缓冲区大小与它所说的一样大。
strcadd	低危险	确保缓冲区大小与它所说的一样大。
strncpy	低危险	确保缓冲区大小与它所说的一样大。
vsnprintf	低危险	确保缓冲区大小与它所说的一样大。



2. System-Level Protections can help

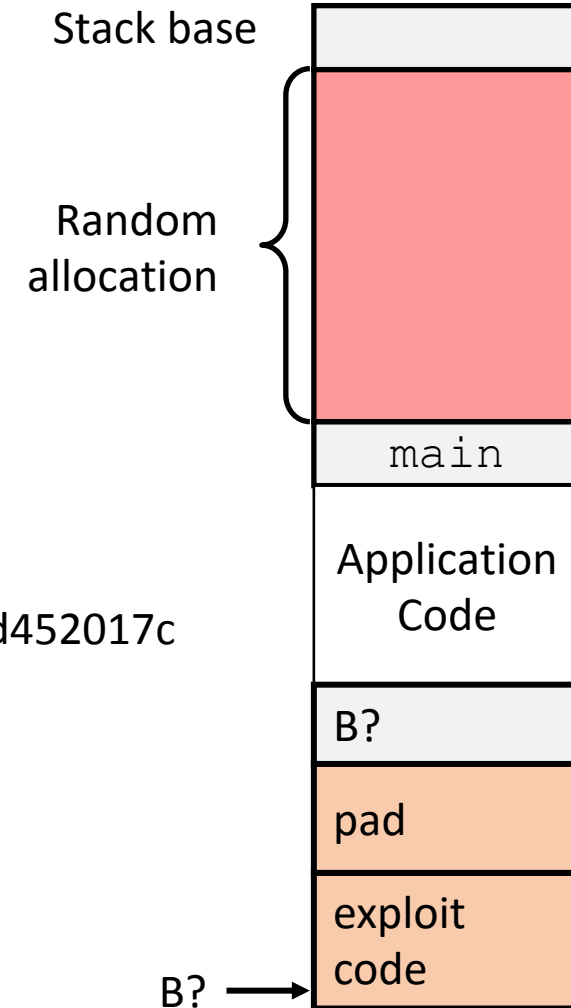


■ Randomized stack offsets

- 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

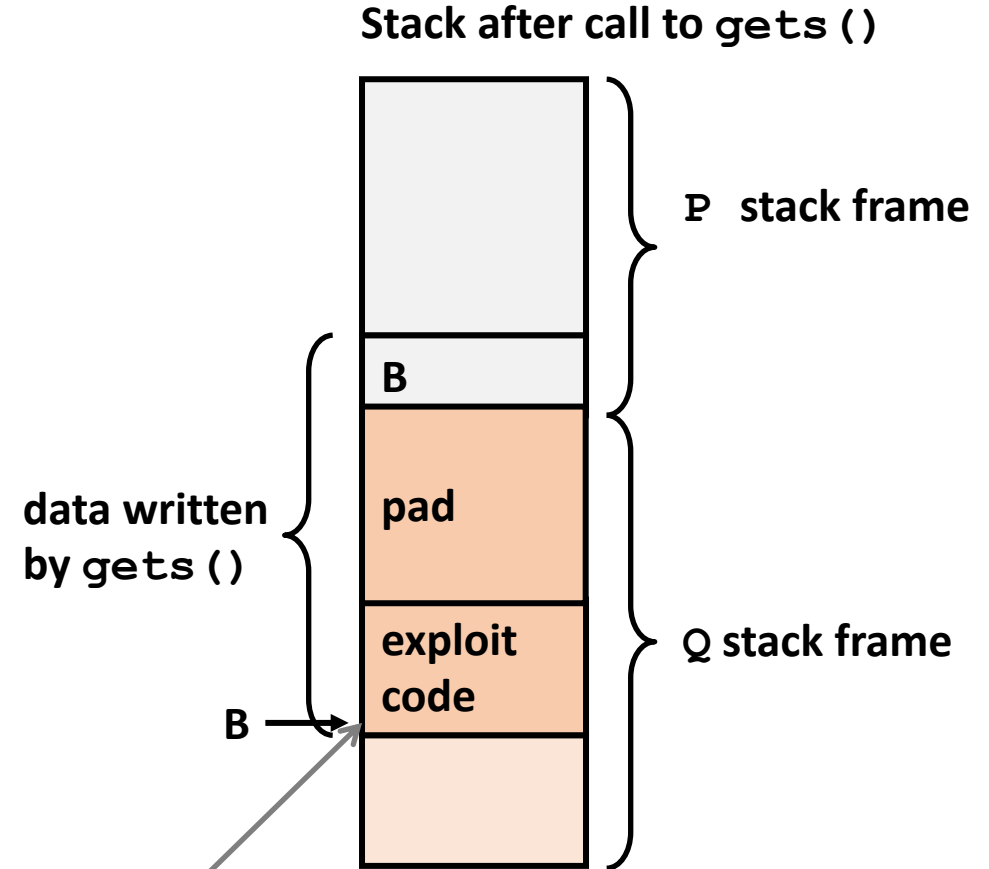


2. System-Level Protections can help



■ 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



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:

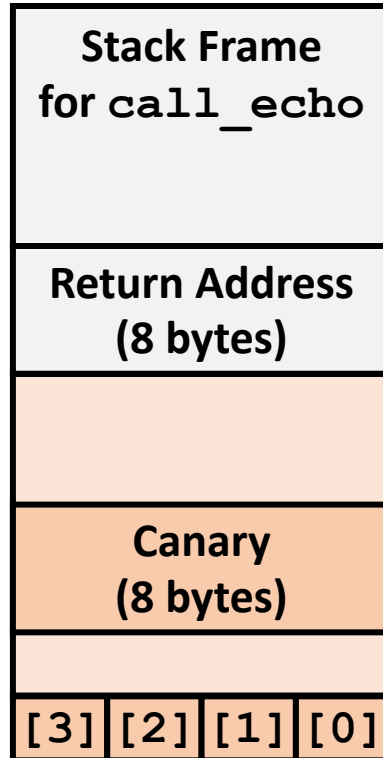
```
40072f:  sub    $0x18,%rsp
400733:  mov     %fs:0x28,%rax
40073c:  mov     %rax,0x8(%rsp)
400741:  xor     %eax,%eax
400743:  mov     %rsp,%rdi
400746:  callq   4006e0 <gets>
40074b:  mov     %rsp,%rdi
40074e:  callq   400570 <puts@plt>
400753:  mov     0x8(%rsp),%rax
400758:  xor     %fs:0x28,%rax
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



Stack Frame for call_echo			
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:  
    . . .
```



Machine-level Programming (IV): Advanced



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



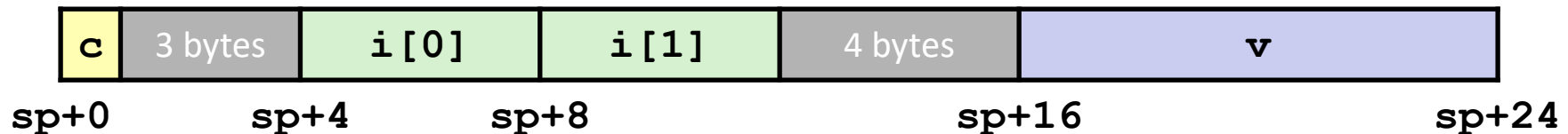
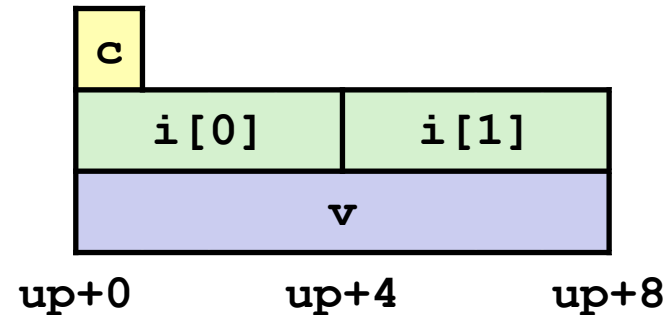
Union Allocation



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

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

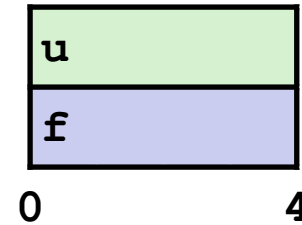
```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *sp;
```



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

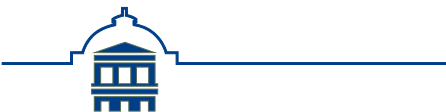
- Least significant byte has lowest address
- Intel x86, ARM Android and IOS

■ 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]);

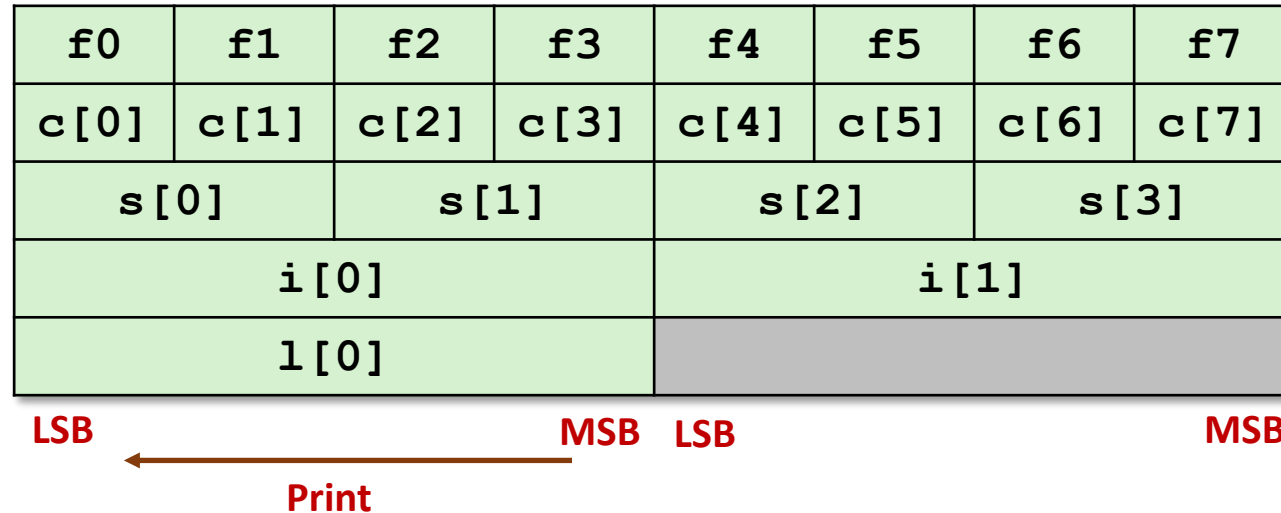
printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```



Byte Ordering on IA32



Little Endian



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

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]							

MSB

LSB MSB

LSB

Print

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

