进阶话题

计算机系统基础

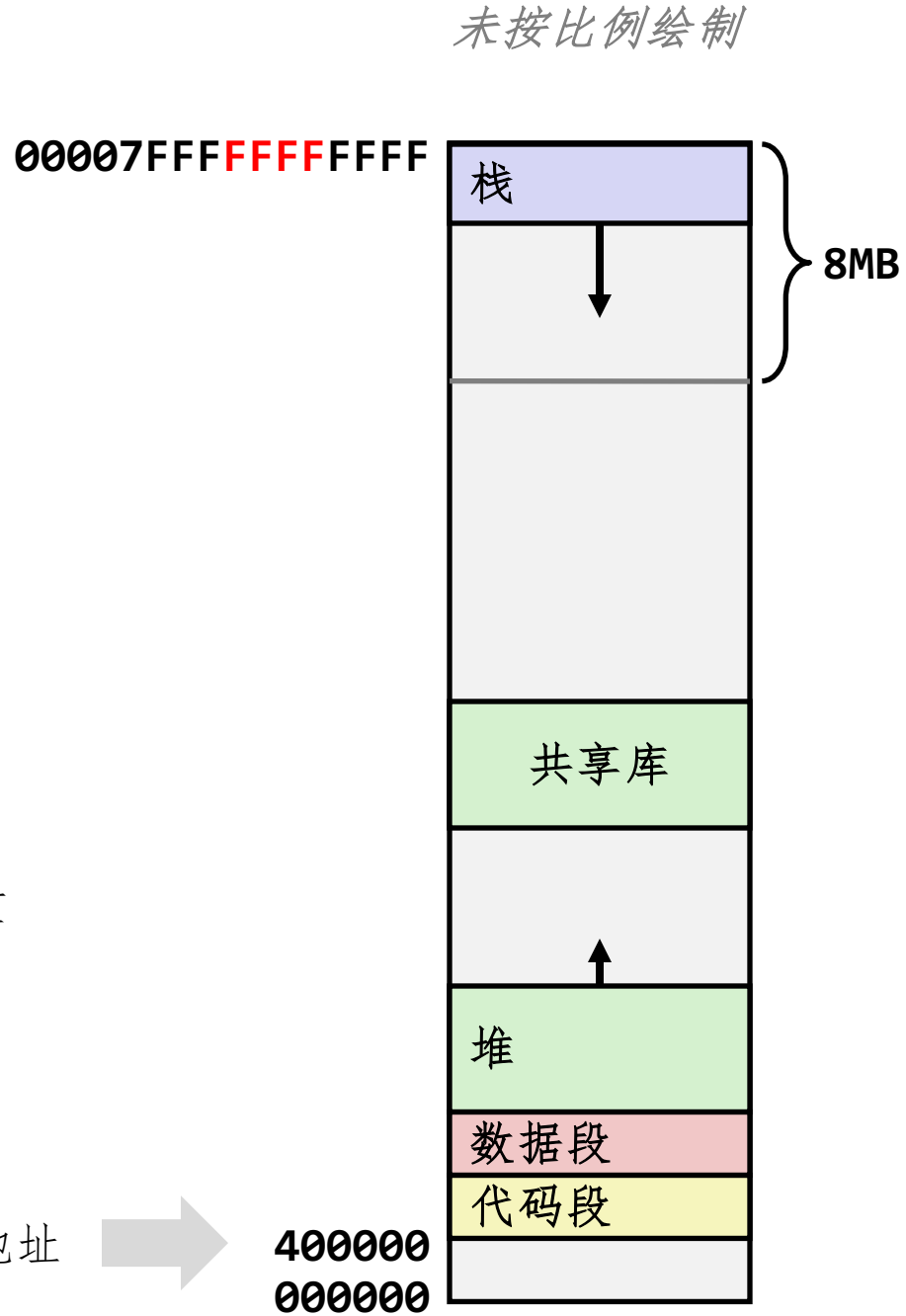
任课教师:
龚奕利
yiligong@whu.edu.cn

本节内容

- 内存布局
- 缓冲区溢出
  - 漏洞
  - 保护
- 联合

x86-64 Linux内存布局

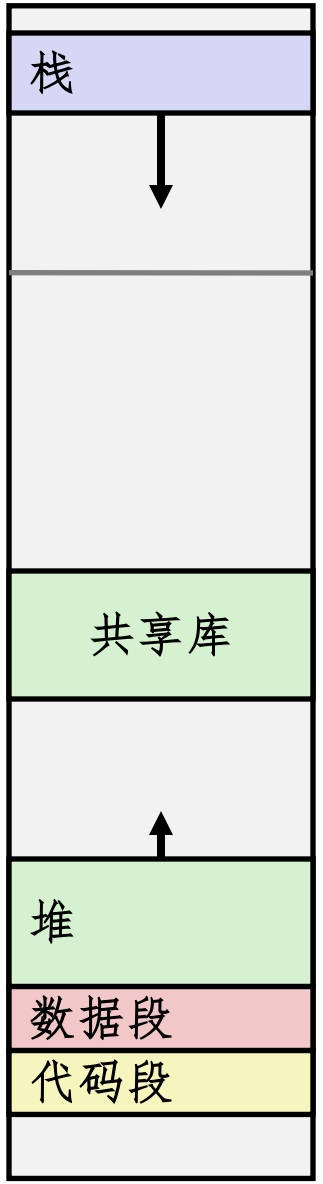
- 栈
  - 运行时栈(8MB限制)
  - 例如，局部变量
- 堆
  - 根据需要动态分配
  - 调用函数malloc(), calloc(), new()时
- 数据
  - 静态分配的数据
  - 例如，全局变量，static变量，字符串常量
- 代码段/共享库
  - 可执行的机器指令
  - 只读



内存分配示例

未按比例绘制

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



都被分配到哪去了呢？

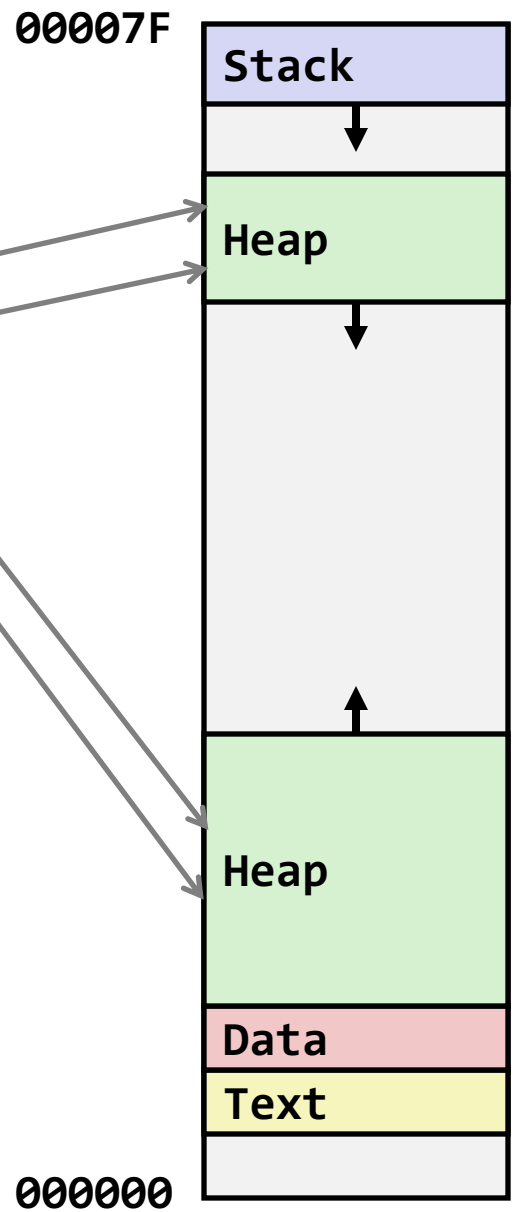
x86-64地址示例

地址范围0~2^{47}

local	0x00007ffe4d3be87c
p1 (256MB)	0x00007f7262a1e010
p3 (4GB)	0x00007f7162a1d010
p4 (256B)	0x000000008359d120
p2 (256B)	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590

local	0x7ff7bca78884
p1(256MB)	0x7fccec700000
p3(4GB)	0x7fcbec700000
p4(256B)	0x600002c58100
p2(256B)	0x600002c58000
big_array	0x10348f020
huge_array	0x10448f020
main	0x10348add0
useless	0x10348adc0
global	0x10348f018

未按比例绘制



本节内容

- 内存布局
- 缓冲区溢出
  - 漏洞
  - 保护
- 联合

回想：内存引用的BUG

```
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
```

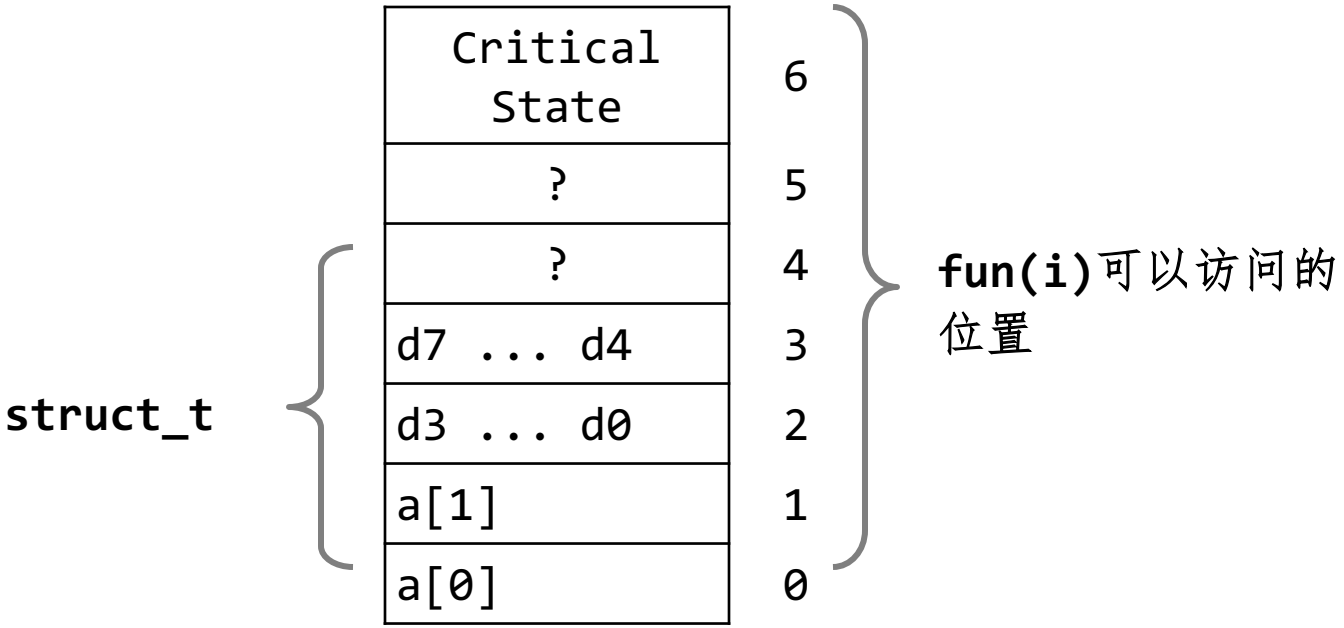
- 结果具有系统特异性

内存引用BUG示例

```
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
```

解释：



这些问题事关重大

- 通常被称作“缓冲区溢出”
  - 超出内存为数组所分配大小时发生
- 为什么事关重大？
  - It’s the #1 technical cause of security vulnerabilities这是导致安全漏洞的#1技术原因
    - #1 overall cause is social engineering / user ignorance
- 最常见的形式
  - 未检查输入字符串的长度
  - 特别是对于栈上有界的字符数组
    - 有时也被称为堆栈粉碎

String库源代码

- Unix函数gets()的实现

```
/* 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;
}
```

  - 没有办法对读取的字符数做限制
- 其它库函数也有类似的问题
  - strcpy, strcat:复制任意长度的字符串
  - scanf, fscanf, sscanf, when given %s conversion specification当使用%s的转换规范时

缓冲区漏洞代码

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

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

那么，要多大才足够大呢？

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

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

缓冲区溢出汇编代码

echo:

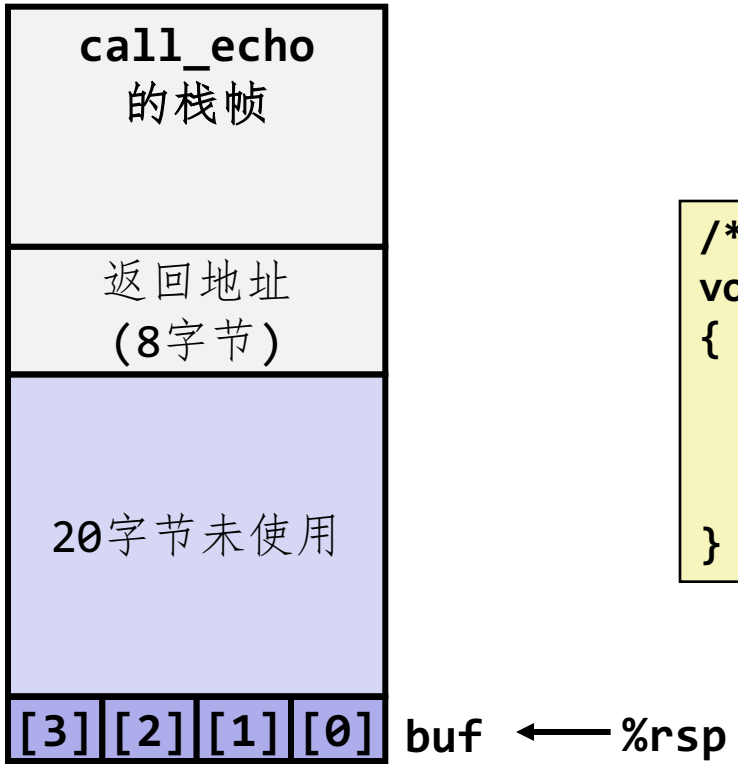
```
0000000004006cf <echo>:
 4006cf:   48 83 ec 18              sub    $0x18,%rsp
 4006d3:   48 89 e7                 mov    %rsp,%rdi
 4006d6:   e8 a5 ff ff ff           callq  400680 <gets>
 4006db:   48 89 e7                 mov    %rsp,%rdi
 4006de:   e8 3d fe ff ff           callq  400520 <puts@plt>
 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
```

缓冲区溢出的栈

调用gets前

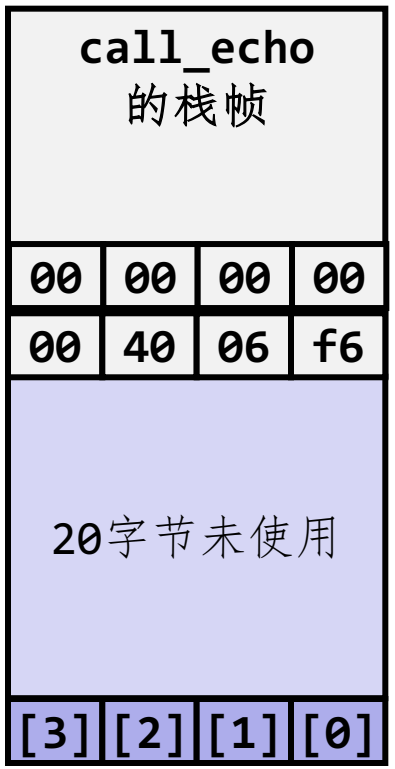



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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

缓冲区溢出的栈示例

调用gets前




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

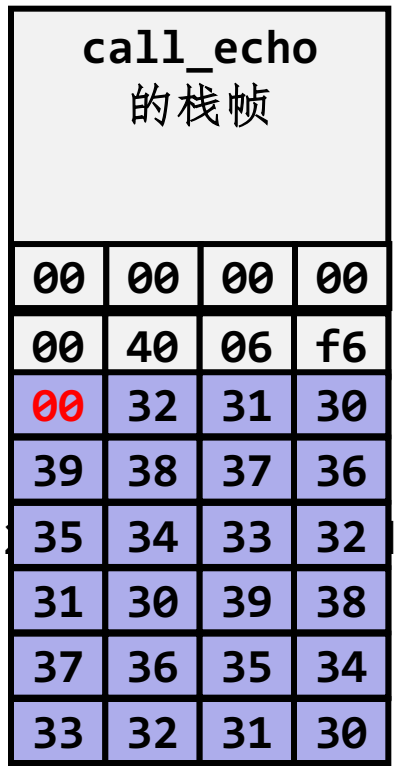
```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

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

缓冲区溢出的栈示例#1

调用gets后




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

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

call_echo:

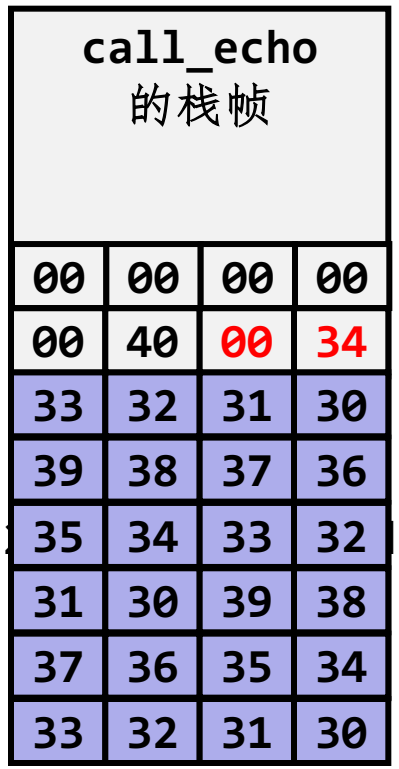
```
  . . .
  4006f1:   callq  4006cf <echo>
  4006f6:   add    $0x8,%rsp
  . . .
```

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

Overflowed buffer, but did not corrupt state溢出的缓冲区，但没有造成状态崩溃

缓冲区溢出的栈示例#2

调用gets后




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

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

call_echo:

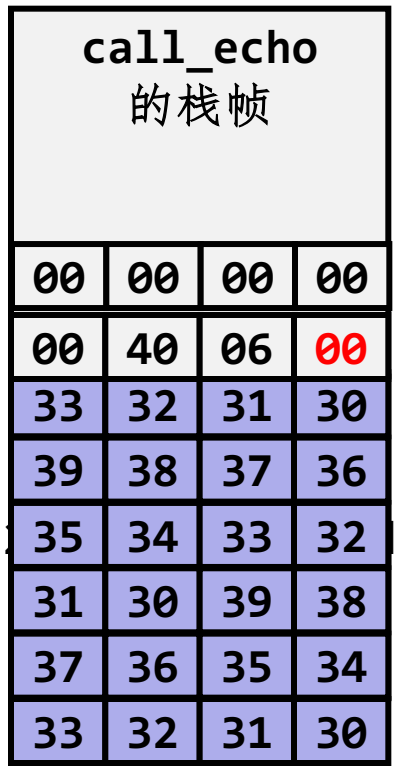
```
  . . .
  4006f1:   callq  4006cf <echo>
  4006f6:   add    $0x8,%rsp
  . . .
```

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

溢出的缓冲区，且破坏了返回指针

缓冲区溢出的栈示例#3

调用gets后




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

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

call_echo:

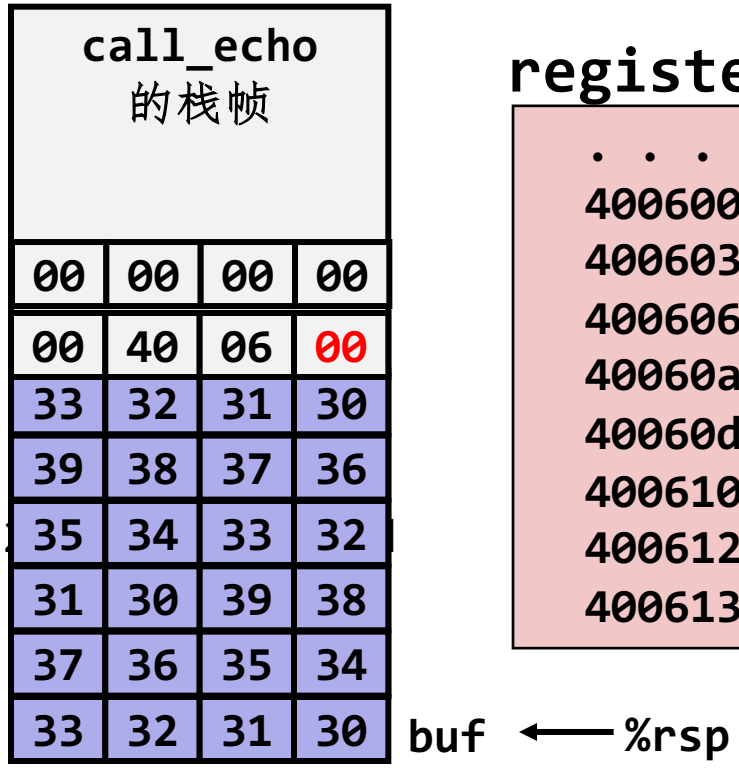
```
  . . .
  4006f1:   callq  4006cf <echo>
  4006f6:   add    $0x8,%rsp
  . . .
```

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

缓冲区溢出，破坏了返回指针，但程序似乎可以运作！

缓冲区溢出的示例#3 解释

调用gets后




```
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
```

“返回”不相关的代码

Lots of things happen, without modifying critical state

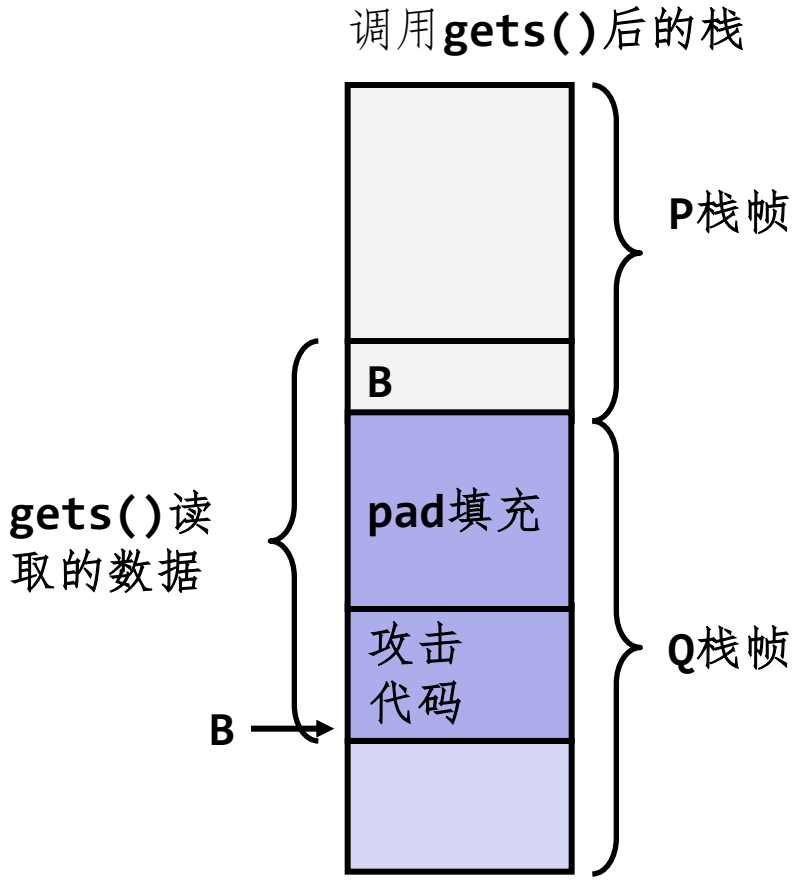
Eventually executes retq back to main

代码注入攻击

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

返回地址A

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



- 输入的字符串包括了可执行代码的字节表示
- 用B的地址覆写返回地址A
- 当Q执行ret，就会跳转到攻击代码

利用缓冲区溢出

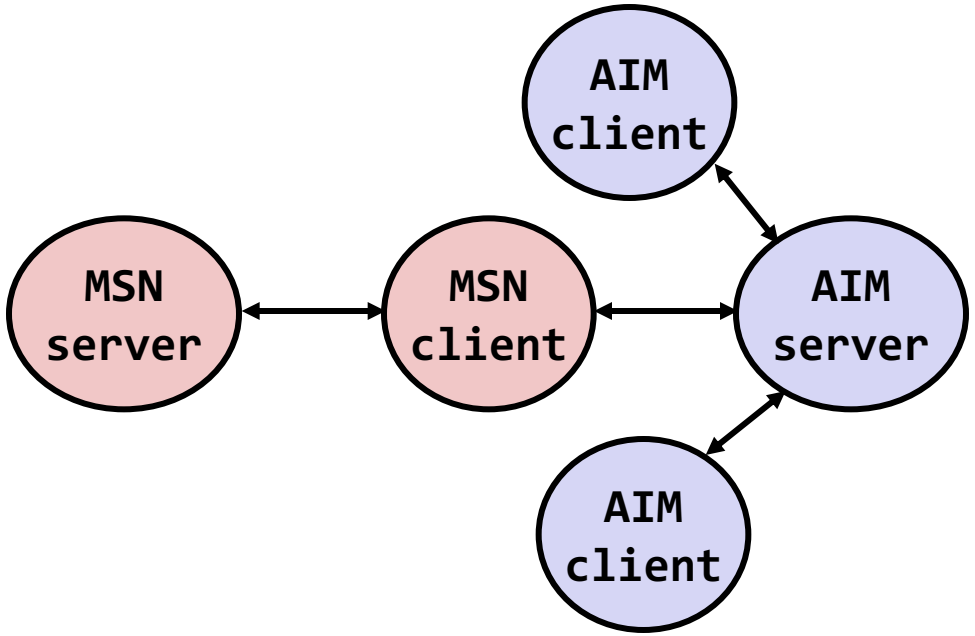
- *远程机器利用缓冲区溢出漏洞可以在受害机器上执行任意代码*
- 令人苦恼的是，这在实际程序中很常见
  - 程序员总是在范相同的错误☹
  - 最近一些措施让这些攻击变得更加困难
- 近几十年的例子
  - Original “Internet worm” (1988)
  - “IM wars” (1999)
  - Twilight hack on Wii (2000s)
  - …还有很多很多
- 你将在**attacklab**中学习部分技巧
  - 希望能帮助你不在自己的程序中留下这样的漏洞！！

示例1：最初的网络蠕虫(1988)

- 利用一些漏洞进行传播
  - 早期版本的手指服务器(fingerd)使用gets()方法读取客户端发送的参数：
    - **finger yiligong@whu.edu.cn**
  - 蠕虫通过发送伪造的参数进行攻击：
    - **finger *“exploit-code padding new-return-address”***
    - 攻击代码：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*
  - 蠕虫的年轻作者被起诉

示例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（续）

- **August 1999**
  - 神奇的是，MSN服务器的用户不再能使用AIM的服务器了
  - 微软和AOL开始了IM war：
    - AOL修改了服务器以拒绝MSN的用户
    - Microsoft makes changes to clients to defeat AOL changes微软为了还击对用户做了修改
    - At least 13 such skirmishes
  - 事实上是怎么回事呢？
    - AOL发现了自己AIM用户的缓冲区溢出漏洞
    - 他们利用这个漏洞探测和隔离微软：攻击代码会返回一个4字节的签名（在AIM用户的某些位置）给服务器
    - 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

后来确定这封邮件来自微软内部!

另：蠕虫和病毒

- 蠕虫是这样的一个程序：
  - 可以自己运行
  - 可以将自己的完整版传播到其它计算机中

- 病毒是这样的代码：
  - 将自己加在别的程序中
  - 并不能独立运行

- 两者(通常)都是为了在计算机之间传播和造成破坏而设计的

应对缓冲区溢出措施

- 避免溢出漏洞
- 使用系统级保护
- 编译器使用金丝雀
- 我们一一道来……

1.避免代码中的溢出漏洞

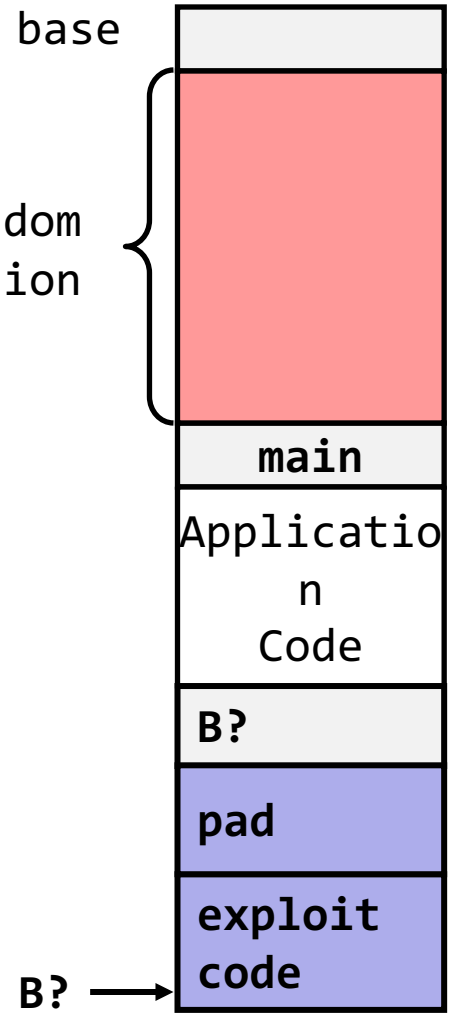
```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- 例如，使用限制字符串长度的库例程
  - 使用**fgets**代替**gets**
  - 使用**strncpy**代替**strcpy**
  - 不要使用**scanf**的**%s** conversion specification
    - 使用**fgets**来读取字符串
    - 或使用**%ns**，其中**n**是一个合适的整型

2.系统级的保护

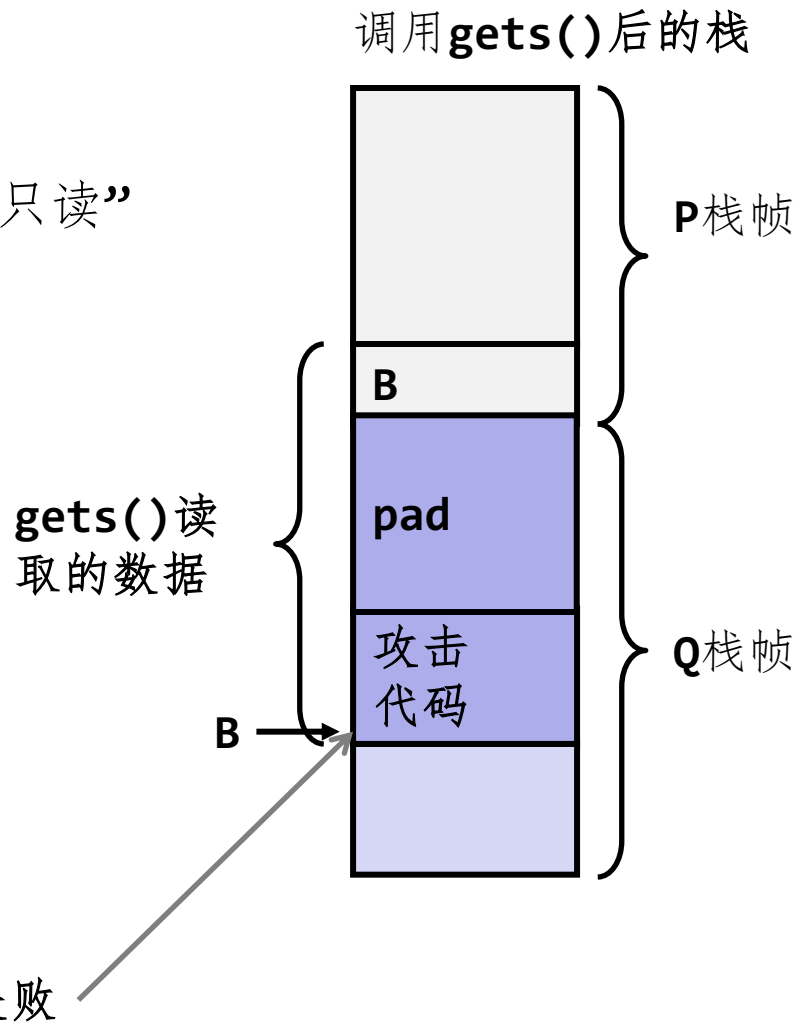
- 栈随机化
  - 程序开始时，在栈上分配一段随机大小的空间
  - 整个程序的地址发生位移
  - 让黑客更加难以确定插入代码的地址
  - E.g.: 5 executions of memory allocation code5个内存分配代码的执行
    - 每次执行程序，栈的位置都发生改变

local 0x7ffe4d3be87c 0x7fff75a4f9fc 0x7ffeadb7c80c 0x7ffeaea2fdac 0x7ffcd452017c



2.系统级保护

- 不可执行代码段
  - 传统的x86体系中，可以将内存区域标记为“只读”或“可写”
    - 可以执行任何可读的代码
  - 增加了显式“执行”权限
  - 栈被标记为不可执行



3.栈金丝雀

- 想法
  - 在缓冲区附近放置一个特殊值（“金丝雀值”）
  - 退出函数前检查是否遭到破坏
- GCC实现
  - **-fstack-protector**
  - Now the default (disabled earlier)

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

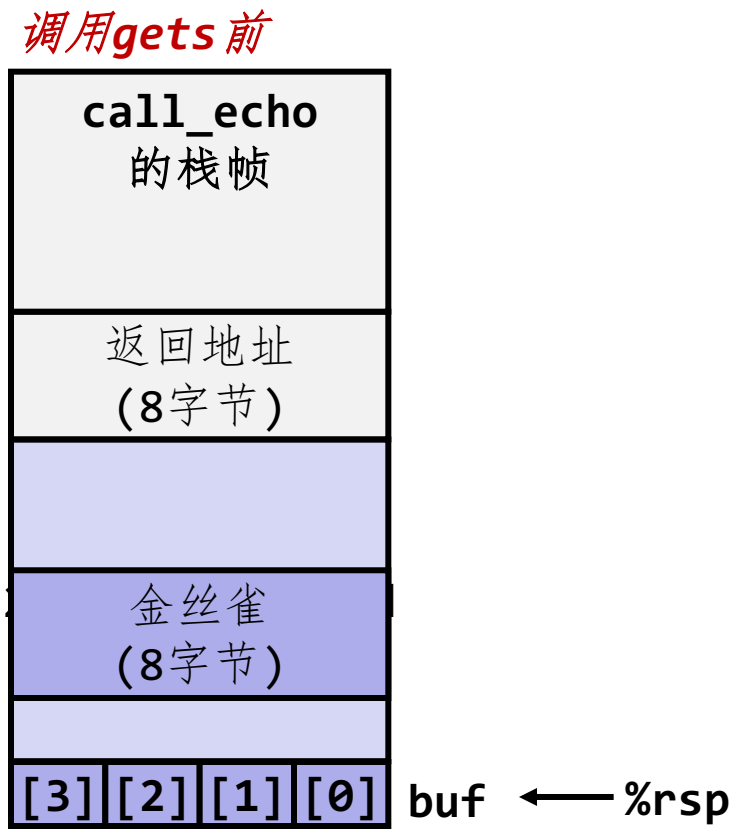
```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```

受保护缓冲区的汇编代码

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
```

设置金丝雀

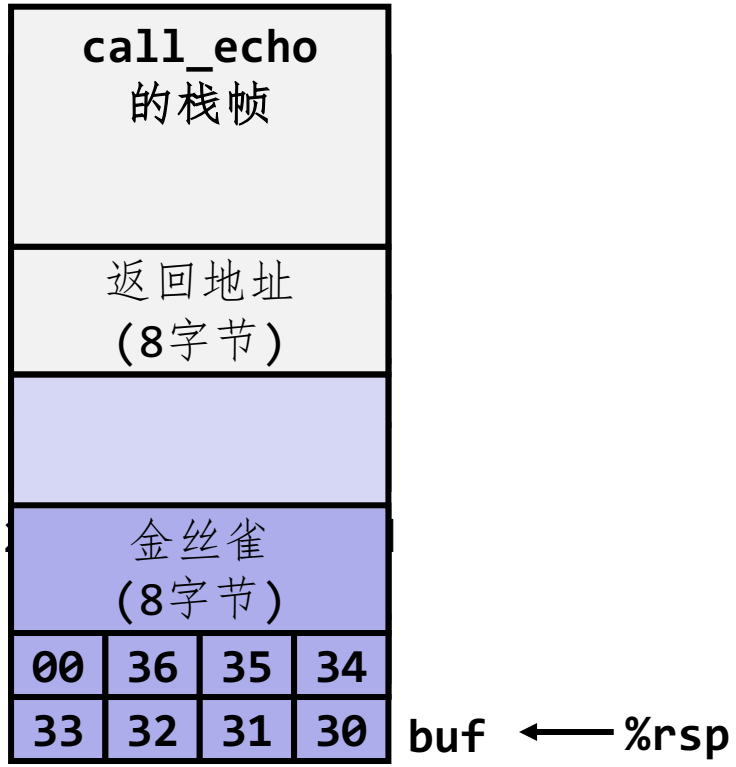



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:0x28, %rax  # Get canary
    movq    %rax, 8(%rsp) # Place on stack
    xorl    %eax, %eax    # Erase canary
    . . .
```

检查金丝雀

调用gets后




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

输入*0123456*

```
echo:
    . . .
    movq    8(%rsp), %rax     # Retrieve from stack
    xorq    %fs:0x28, %rax    # Compare to canary
    je      .L6               # If same, OK
    call    __stack_chk_fail  # FAIL
.L6:. . .
```

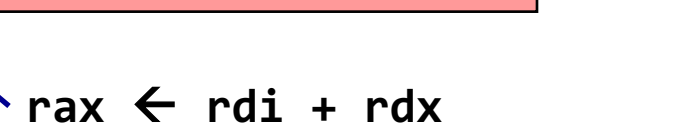
Return-Oriented Programming Attacks

- 挑战（对黑客而言）
  - 栈随机化使预测缓冲区的位置变得困难
  - Marking stack nonexecutable makes it hard to insert binary code将栈标记为不可执行使插入二进制代码变得困难
- 可选择的策略
  - 使用已经存在的代码
    - 例如，stdlib的库代码
  - 字符串的片段组合在一起以达到希望的结果
  - *并不能“骗”过金丝雀*
- *从gadgets中构建程序*
  - 以**ret结尾的指令序列**
    - 由单字节**0xc3编码**
  - Code positions fixed from run to run
  - 代码是可执行的

Gadget示例#1

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

```
0000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```



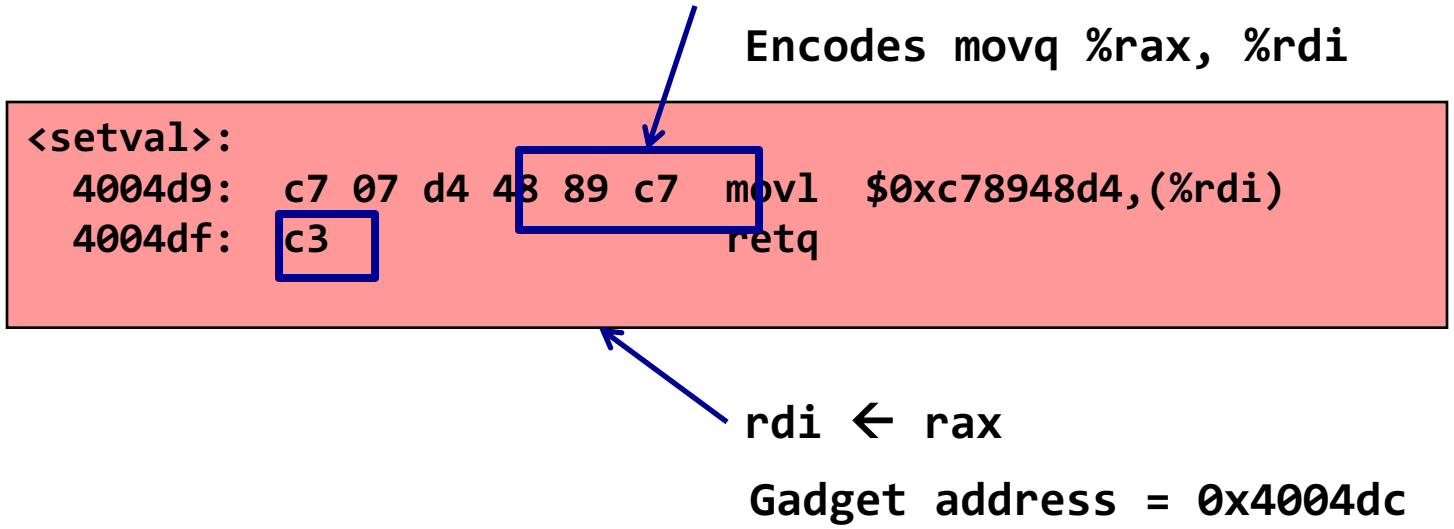
rax ← rdi + rdx

Gadget address = 0x4004d4

- 使用已经存在函数的尾部

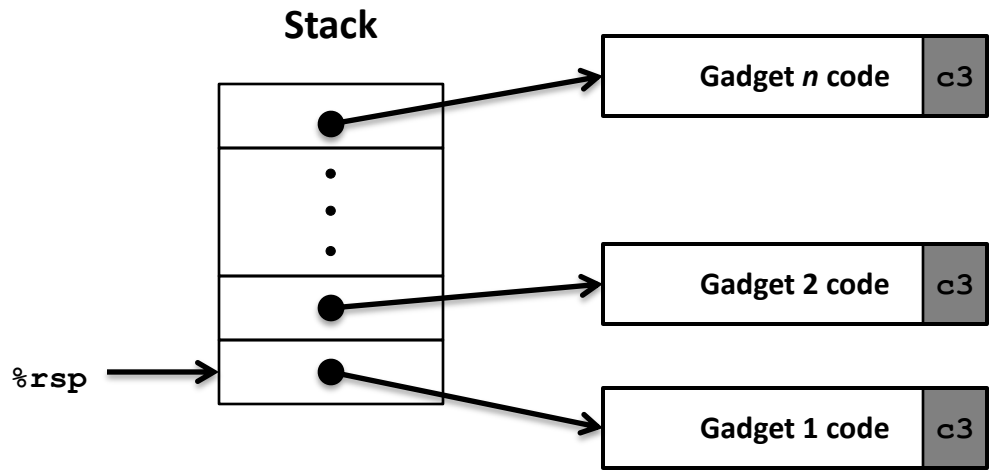
Gadget示例#2

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



- 将字节码移作他用

ROP Execution（怕术语啥用错，这页就不翻了）



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

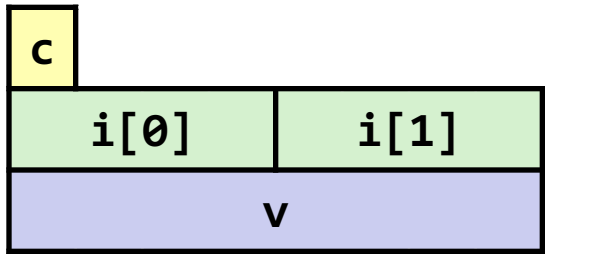
本节内容

- 内存布局
- 缓冲区溢出
  - 漏洞
  - 保护
- 联合

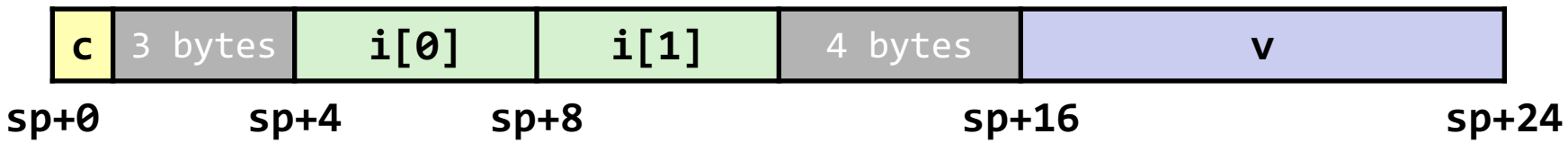
联合的内存分配

- 根据最大的元素分配
- 一次只能使用一个字段

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

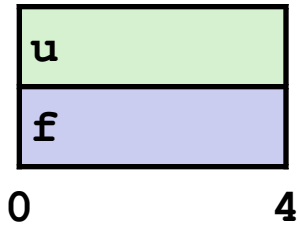



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



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 ?

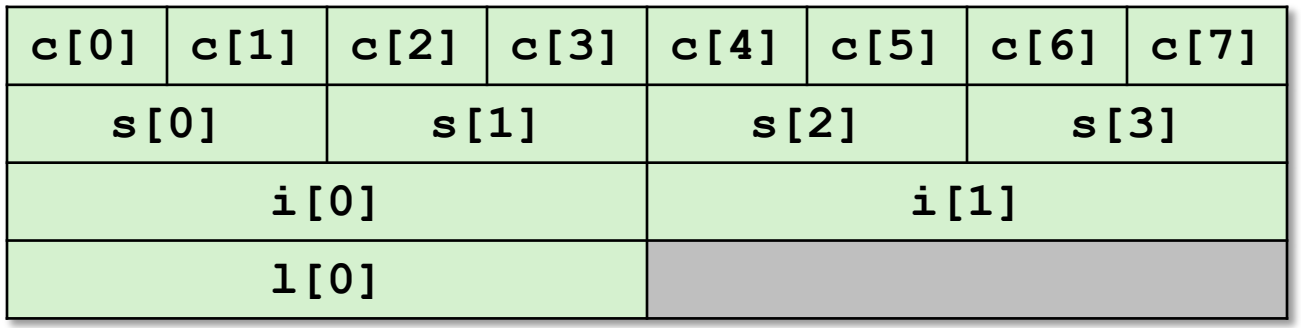
字节顺序再探

- 想法
  - 单字、双字、四字在内存中以2/4/8连续字节存储
  - 哪个字节是最高字节呢？
  - 当机器交换二进制数据是会产生问题
- 大端
  - 最高字节在最低地址处
  - Sparc
- 小端
  - 最低字节在最低地址处
  - Intel x86, ARM Android和IOS
- Bi Endian
  - Can be configured either way
  - ARM

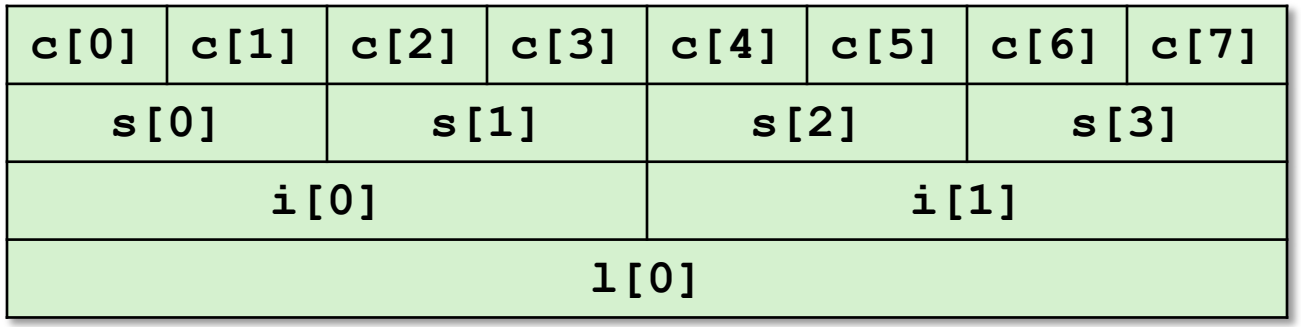
字节顺序示例

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

32-bit



64-bit

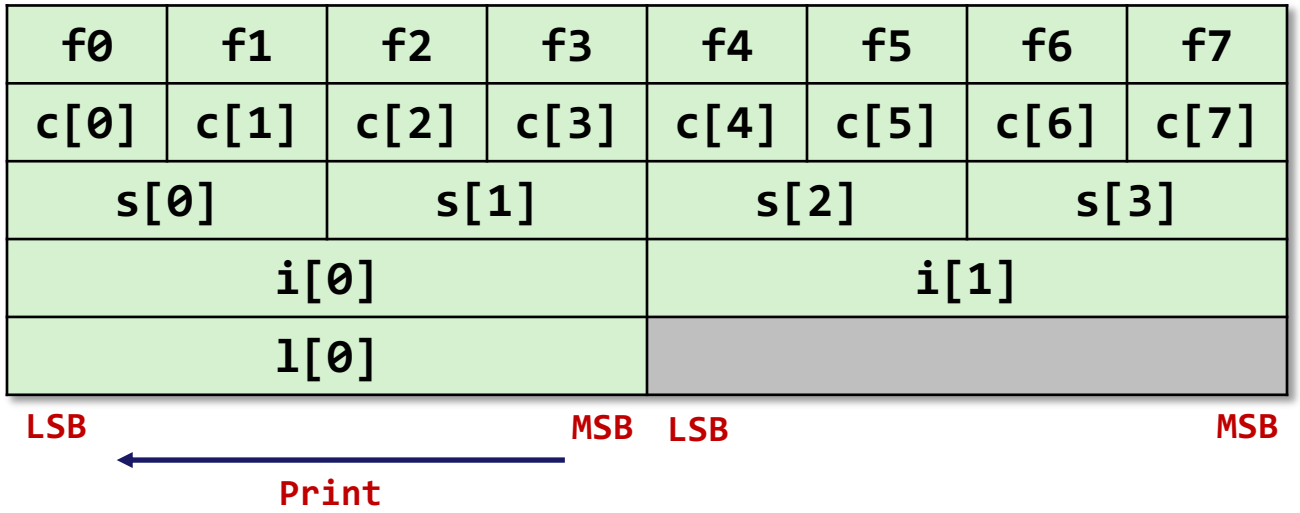


字节顺序示例（续）

```
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]);
```

IA32的字节顺序

小端

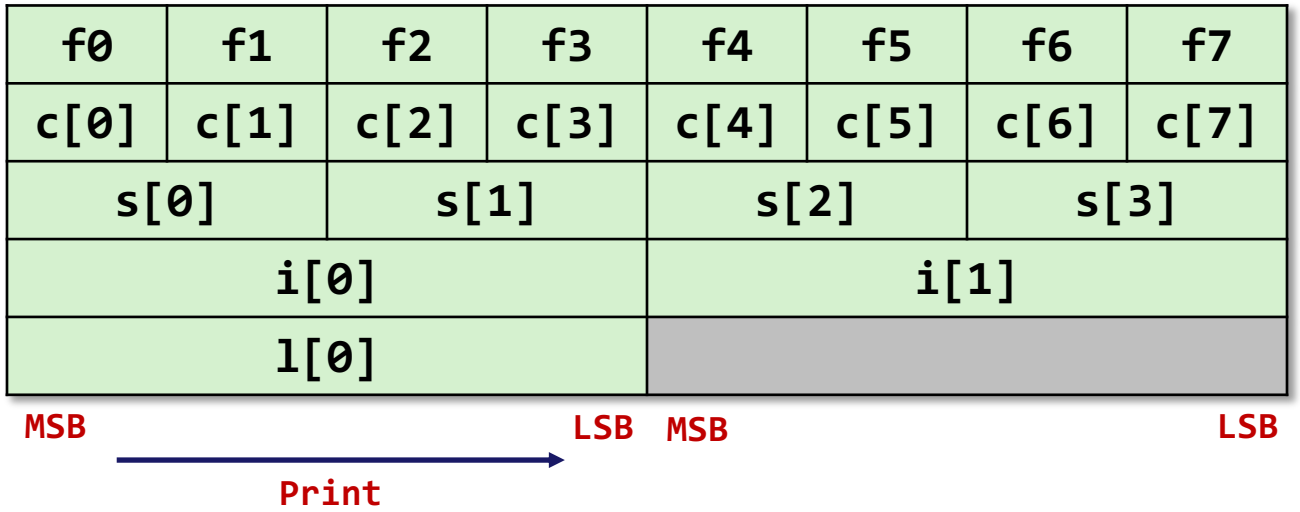


输出:

```
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]
```

Sun的字节顺序

大端

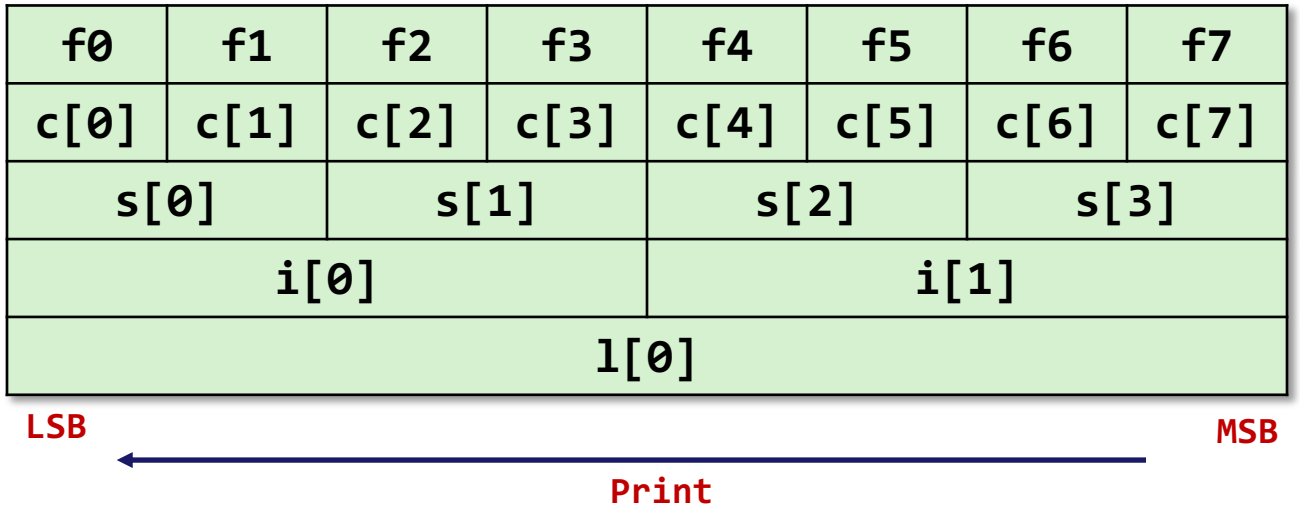


输出：

```
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]
```

x86-64的字节顺序

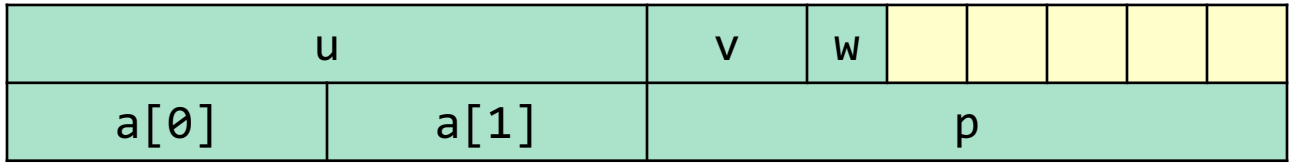
小端



输出：

```
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]
```

练习题3.43



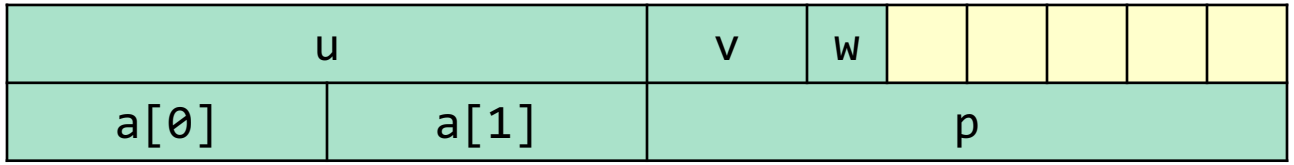

```
typedef union {
    struct {
        long                u;
        short               v;
        char                w;
    } t1;
    struct {
        int                 a[2];
        char                *p;
    } t2;
} u_type;

void get(u_type *up, type *dest) {
    *dest = expr;
}
```

expr	*type*	Code
up->t1.u		
up->t1.v		
&up->t1.w		
up->t2.a		
up->t2.a[up->t1.u]		
*up->t2.p		

Instruction		Effect	Description
MOV	*S*, *D*	*D* ← *S*	Move
movb			Move byte
movw			Move word
movl			Move double word
movq			Move quad word
movabsq	*I*, *R*	*R* ← *I*	Move absolute quad word

Figure 3.4 Simple data movement instructions.

练习题3.43



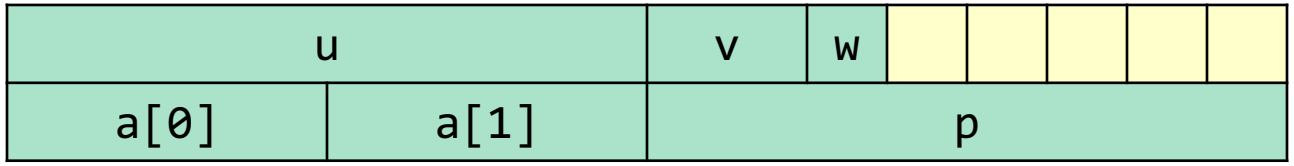

```
typedef union {
    struct {
        long                u;
        short               v;
        char                w;
    } t1;
    struct {
        int                 a[2];
        char                *p;
    } t2;
} u_type;

void get(u_type *up, type *dest) {
    *dest = expr;
}
```

expr	*type*	Code
up->t1.u	long	
up->t1.v	short	
&up->t1.w	char *	
up->t2.a	int *	
up->t2.a[up->t1.u]	int	
*up->t2.p	char	

Instruction		Effect	Description
MOV	*S*, *D*	*D* ← *S*	Move
movb			Move byte
movw			Move word
movl			Move double word
movq			Move quad word
movabsq	*I*, *R*	*R* ← *I*	Move absolute quad word

Figure 3.4 Simple data movement instructions.

练习题3.43




```
typedef union {
    struct {
        long                u;
        short               v;
        char                w;
    } t1;
    struct {
        int                 a[2];
        char                *p;
    } t2;
} u_type;

void get(u_type *up, type *dest) {
    *dest = expr;
}
```

expr	*type*	Code
up->t1.u	long	movq (%rdi), %rax movq %rax, (%rsi)
up->t1.v	short	movw 8(%rdi), %ax movw %ax, (%rsi)
&up->t1.w	char *	lea 10(%rdi), %rax movq %rax, (%rsi)
up->t2.a	int *	movq %rdi, (%rsi)

up->t2.a[up->t1.u]	int	movq (%rdi), %rax
		movl (%rdi, %rax, 4), %eax
		movl %eax, (%rsi)

*up->t2.p	char	movb 8(%rdi), %al movb %al, (%rsi)

Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
movb			Move byte
movw			Move word
movl			Move double word
movq			Move quad word
movabsq	I, R	$R \leftarrow I$	Move absolute quad word

Figure 3.4 Simple data movement instructions.

总结：C语言中的组合

- 数组
  - 连续分配的内存
  - 满足每个元素的对齐要求
  - 指针指向第一个元素
  - 没有边界检查
- 结构
  - Allocate bytes in order declared按声明的顺序分配字节
  - 在中间和结尾填充以满足对齐要求
- 联合
  - Overlay declarations
  - Way to circumvent type system