程序的机器级表示V：高级主题

章节要求

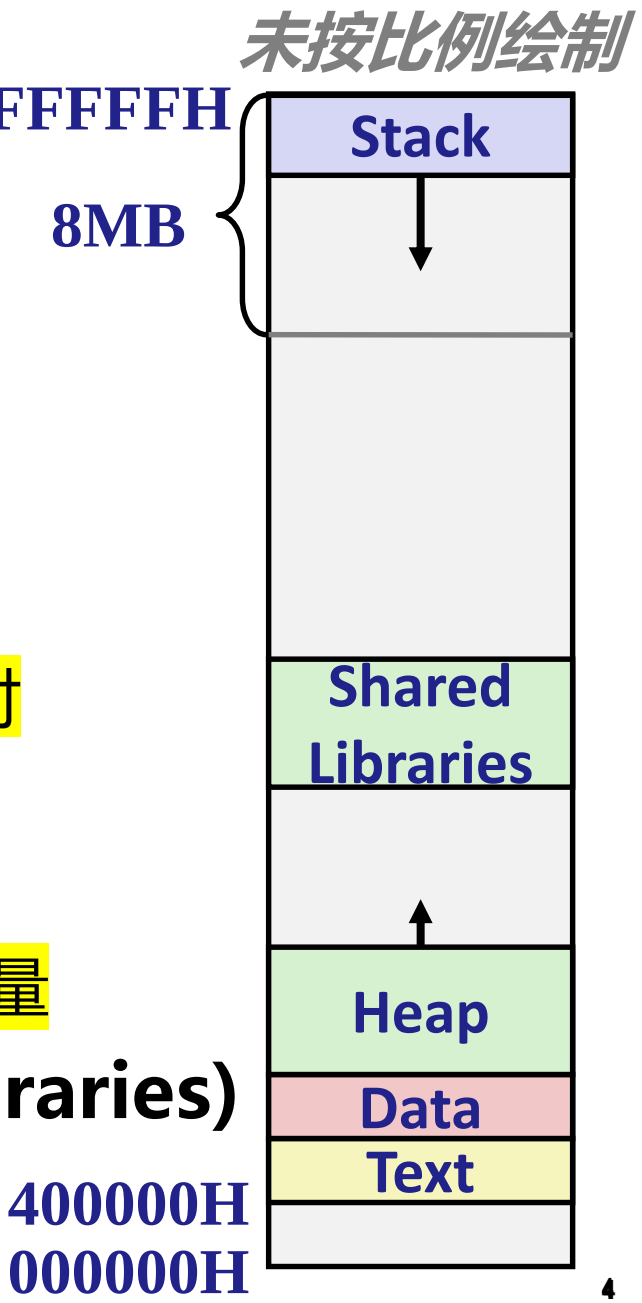
- 明确内存布局细节
- 掌握缓冲区溢出原理 （概念题）
- 知道如何预防缓冲区溢出（简答题）
- 明确一段代码在大端序小端序下的输出
- 经典例题

主要内容

- 内存布局
- 缓冲区溢出
  - 安全隐患
  - 防护
- 联合

x86-64 Linux 内存布局

- **栈(Stack)**
  - 运行时栈 (8MB limit)
  - 涉及局部变量
- **堆(Heap)**
  - 按需动态分配
  - 时机:调用malloc(), calloc(), new()时
- **数据(Data)**
  - 静态分配的内存中保存的数据
  - 全局变量、static变量、字符串常量
- **代码/共享库(Text / Shared Libraries)**
  - 只读的可执行的机器指令

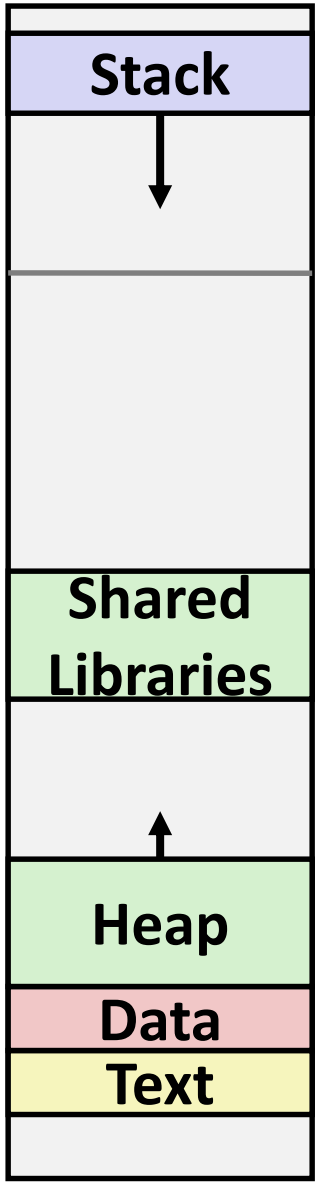


内存分配示例

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

程序中各个部分都在哪里?

未按比例绘制

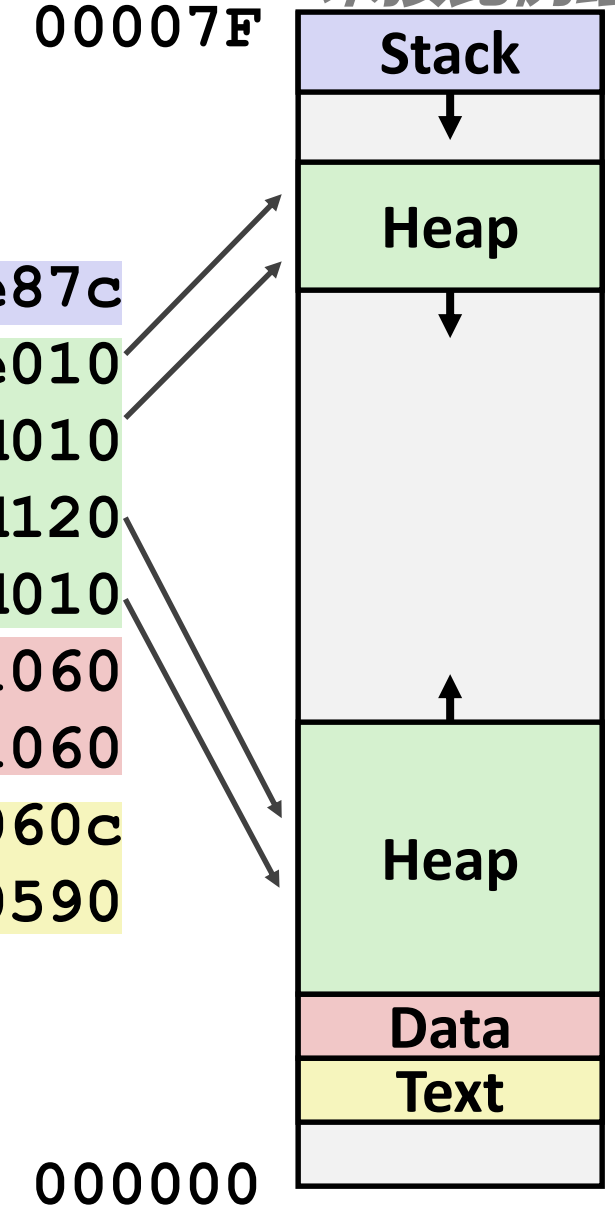


x86-64 例子的地址

未按比例绘制

地址范围 ~2^{47}

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x0000000008359d120
p2	0x0000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590



主要内容

- 内存布局
- **缓冲区溢出**
  - 安全隐患
  - 防护
- 联合

内存引用的Bug示例

```
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(1) → 3.14

fun(2) → 3.1399998664856

fun(3) → 2.00000061035156

fun(4) → 3.14

fun(6) → Segmentation fault

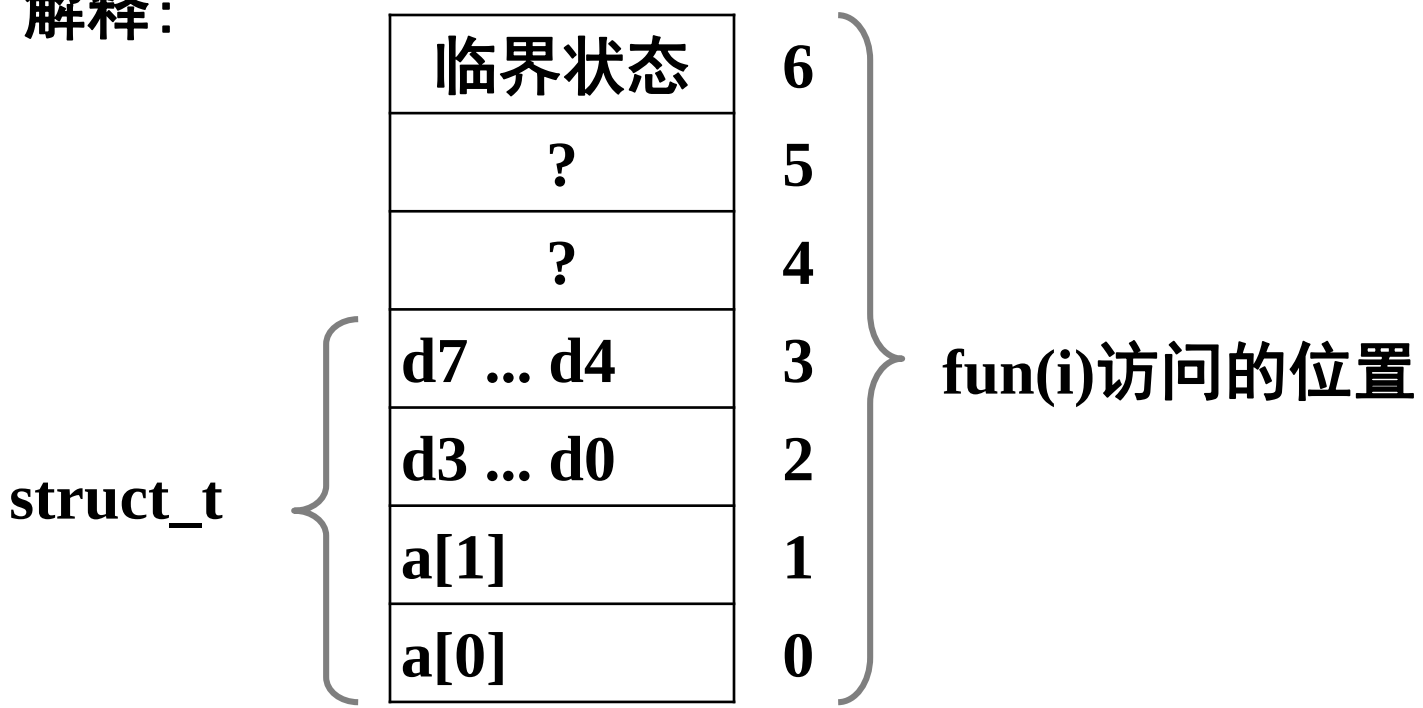
运行结果与系统有关

内存引用的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：



这是个 大 问题。

- **一般称为“缓冲区溢出”**
  - 当超出数组分配的内存大小（范围）
- **为何是大问题?**
  - 破坏性修改
  - 返回地址覆盖
- **更一般的形式**
  - 字符串输入不检查长度
  - 特别是堆栈上的有界字符数组
    - 有时称为堆栈粉碎(stack smashing)

字符串库的代码

- 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;
}
```

  - 无法设定读入字符串的长度限制
- **其他库函数也有类似问题**
  - `strcpy`, `strcat`: 任意长度字符串的拷贝
  - `scanf`, `fscanf`, `sscanf`, 使用 `%s` 转换符时

存在安全隐患的缓冲区代码

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

问题：分配多大才足够?

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

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

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

缓冲区溢出的反汇编

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
```

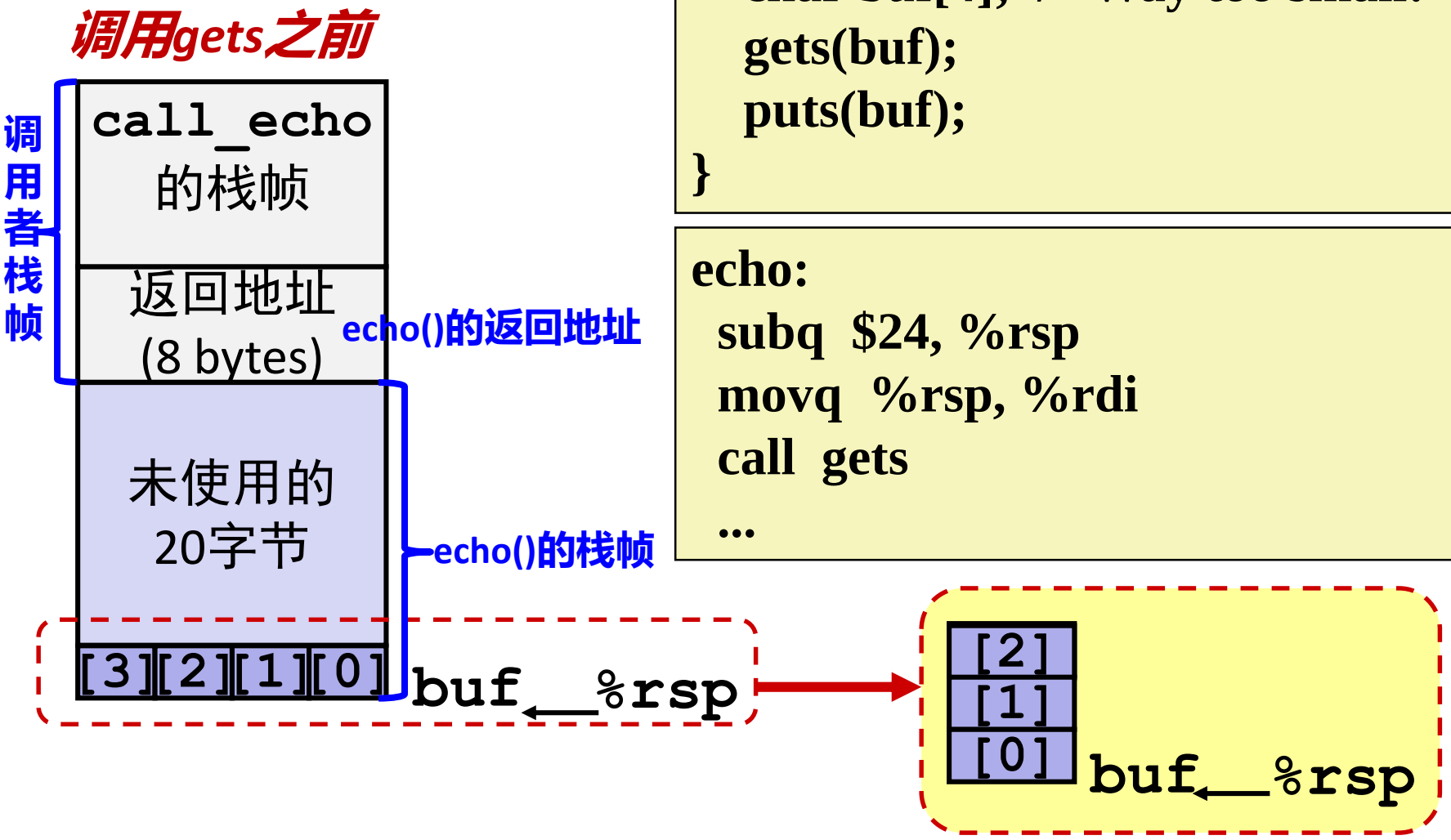
call_echo:

```
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
```

缓冲区溢出的栈示例

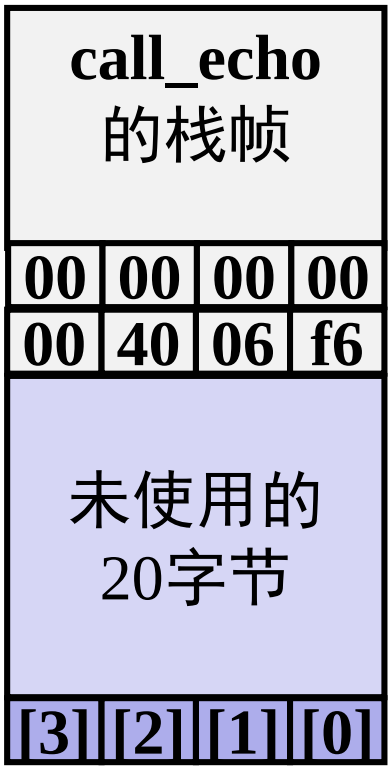
```
/* Echo Line */
void echo(){
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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



缓冲区溢出的栈示例

调用gets之前




```
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之后

call_echo 的栈帧

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
  . . .
```

缓冲区溢出，但没有破坏状态

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

缓冲区溢出的栈示例 #2

调用gets之后

call_echo 的栈帧

00	00	00	00
00	40	00	34
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:

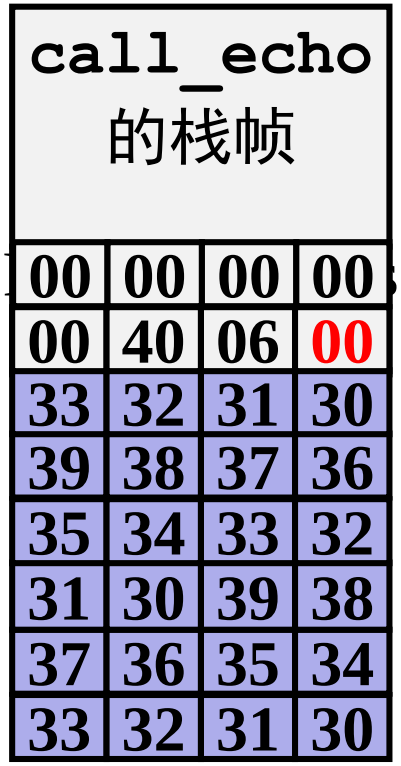
```
  ...
  4006f1:    callq  4006cf <echo>
  4006f6:    add    $0x8,%rsp
  ...
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678
901234
Segmentation Fault
```

溢出的缓冲区，返回地址被破坏

缓冲区溢出的栈示例 #3

调用gets之后




```
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
  . . .
```

溢出的缓冲区,破坏了返回地址，但程序看起来能工作

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901230123
45678901234567890123
```

缓冲区溢出的栈示例 #3 ——解读

调用gets之后

call_echo 的栈帧			
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
```

返回到无关的代码

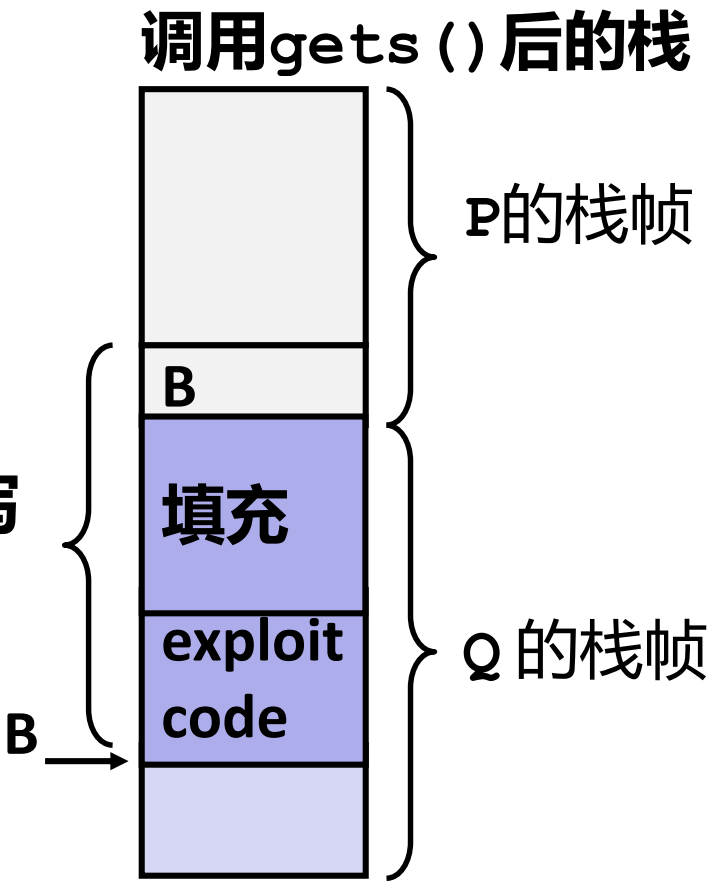
大多数情况不会修改临界状态，最终执行retq 返回主程序

代码注入攻击(Code Injection Attacks)

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

返回地址A

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



- 输入字符串包含可执行代码的字节序列!
- 将返回地址 A用缓冲区B的地址替换
- 当Q执行ret后，将跳转到B处，执行漏洞利用程序(exploit code)

基于缓冲区溢出的漏洞利用程序

- ***缓冲区溢出错误允许远程机器在受害者机器上执行任意代码。***
- **在程序中常见，令人不安**
  - 程序员持续犯相同的错误
  - 最近的措施使这些攻击更加困难。
- **经典案例**
  - 原始"互联网蠕虫"(Internet worm),1988
  - 即时通讯战争"IM wars",1999
  - Twilight hack on Wii, 2000s(不改动硬件，直接在Wii上运行自制程序)
- **在相应的实验中会学到一些技巧**
  - 希望能说服你永远不要在程序中留下这样的漏洞！！

PS：蠕虫和病毒

- **蠕虫(Worm):程序**
  - 可以自行运行
  - 可以将自己的完整版本传播到其他计算机上
- **病毒(Virus): 代码**
  - 将自己添加到别的程序中
  - 不独立运行

- **两者通常都能在计算机之间传播并造成破坏。**

针对缓冲区溢出攻击，怎么做?

- 避免溢出漏洞
- 使用系统级的防护
- 编译器使用“栈金丝雀”(stack canaries)

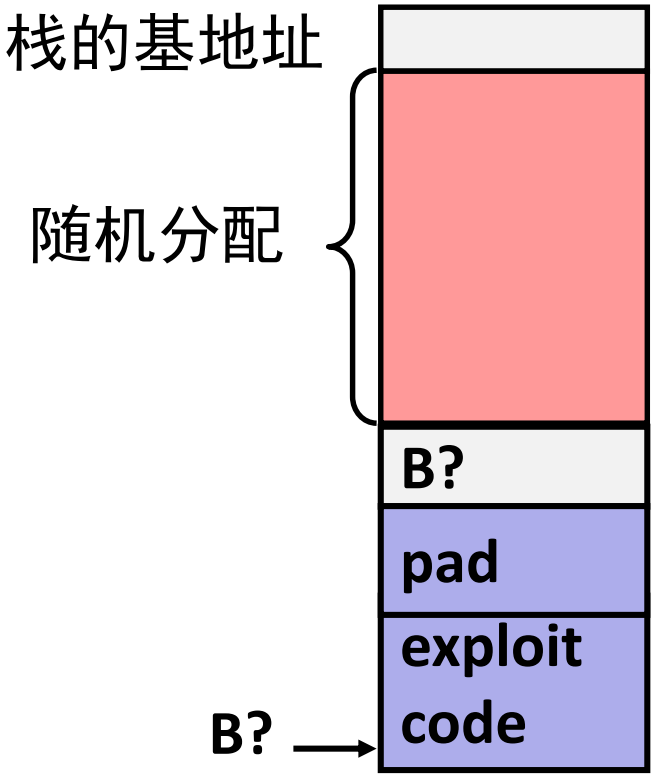
1. 代码中避免溢出漏洞(!)

```
/* 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
  - fgets 代替gets
  - strncpy 代替strcpy
  - 在scanf函数中别用%s
    - 用fgets读入字符串
    - 或用 %ns代替%s，其中n是一个合适的整数

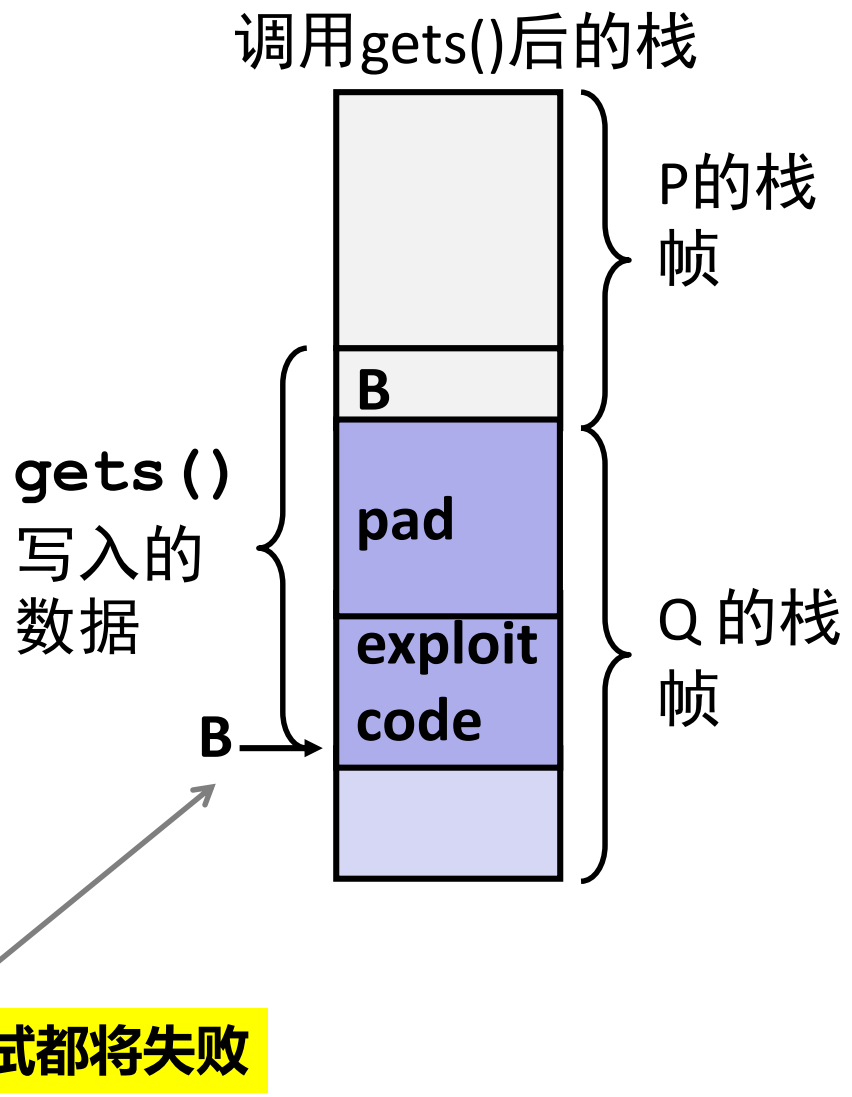
2. 系统级防护

- **随机的栈偏移**
  - 程序启动后，在栈中分配随机数量的空间
  - 将移动整个程序使用的栈空间地址
  - 黑客很难预测插入代码的起始地址
  - 例如：执行5次内存申请代码
    - 每次程序执行，栈都重新定位



2. 系统级防护

- **非可执行代码段**
  - 在传统的x86中，可以标记存储区为“只读”或“可写的”
    - 可以执行任何可读的操作
  - x86-64添加显式“执行”权限
  - 将stack标记为不可执行



3. 栈金丝雀(Stack Canaries)

- **想法**
  - 在栈中buffer之后的位置放置特殊的值——金丝雀("canary")
  - 退出函数之前，检查是否被破坏
- **用GCC 实现**
  - `-fstack-protector`
  - 该选项现在是默认开启的(早期默认关闭)

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

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

保护缓冲区反汇编

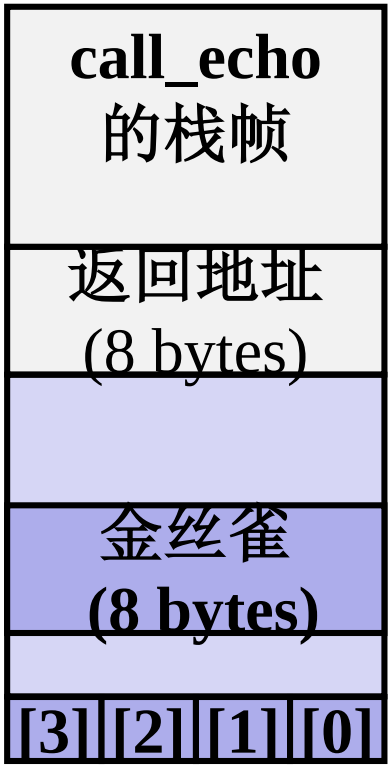
echo:

```
40072f:    sub    $0x18,%rsp
400733:    mov    %fs:0x28,%rax
40073c:    mov    %rax,0x8(%rsp)  #放置（段寻址）
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      #检测
400761:    je     400768 <echo+0x39>
400763:    callq  400580 <__stack_chk_fail@plt>
400768:    add    $0x18,%rsp
40076c:    retq
```

设立金丝雀(Canary)

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

调用gets之前

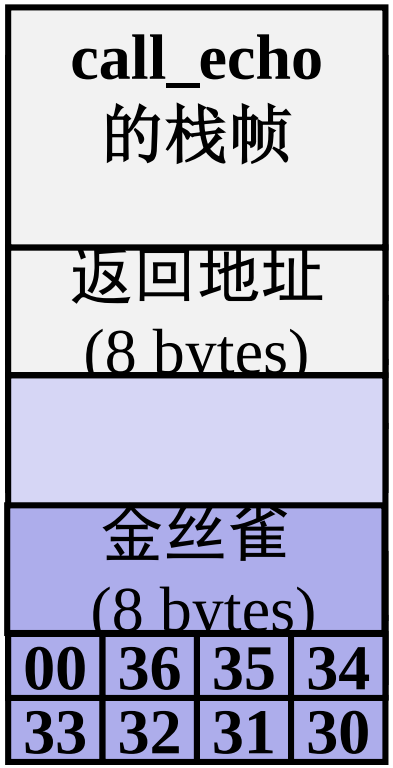



```
echo:
    . . .
    movq    %fs:40, %rax  # Get canary
    movq    %rax, 8(%rsp) # Place on stack
    xorl    %eax, %eax    # Erase canary
    . . .
```

核对金丝雀

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

调用gets后



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:  …
```

Input: *0123456*

面向返回的编程攻击

- **挑战(对黑客)**
  - 栈随机化使缓冲区位置难以预测。
  - 标记栈为不可执行，很难插入二进制代码
- **替代策略**
  - 使用已有代码
    - 例如：stdlib的库代码
  - 将片段串在一起以获得总体期望的结果。
  - 无需克服栈金丝雀

金丝雀不是无懈可击的：https://www.coder.work/article/6184263

- **从小工具构建攻击程序**
  - 以`ret`结尾的指令序列
    - 单字节编码为`0xc3`
  - 每次运行，代码的位置固定
  - 代码可执行

nop sled 空操作雪橇

- 在实际攻击代码之前，插入很长一段nop指令
- nop除了对程序计数器加一没有任何效果
- 只要猜对这段序列中某一个指令的地址，指令就会沿着序列划过，最终执行攻击代码。
- 这样，以枚举的方式就可以破解栈随机化。

小工具例子 #1 (假设栈中返回地址被修改成0x4004d4)

```
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

小工具地址 = 0x4004d4

- 使用现有功能的尾部

小工具例子 #2 (假设栈中返回地址被修改成0x4004dc)

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

`movq %rax, %rdi`的编码

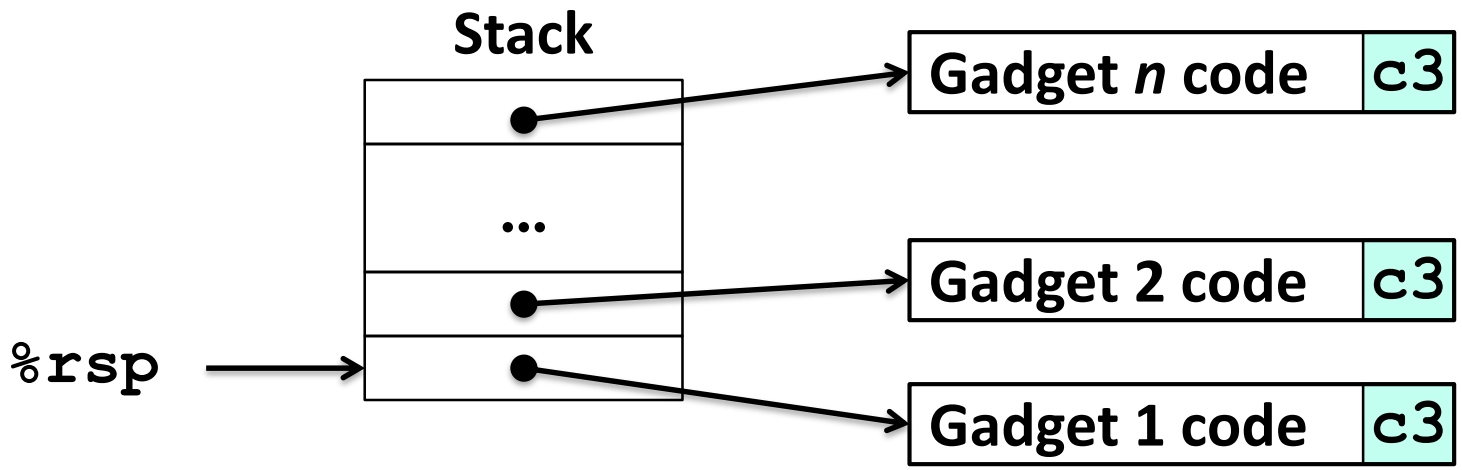
```
<setval>:
  4004d9:  c7 07 d4 48 89 c7  movl  $0xc78948d4,(%rdi)
  4004df:  c3                 retq
```

rdi ← rax

小工具地址 = 0x4004dc

- 改变字节码的用途

面向返回编程(ROP)的 执行



- `ret` 指令触发
  - 将开始运行 Gadget 1
- **每个小工具最终的 `ret`将启动下一个小工具**
- **通过小工具序列的运行，达到攻击目的。**

主要内容

- 内存布局
- 缓冲区溢出
  - 安全隐患
  - 防护
- **联合**

联合union

- 一个联合的总大小等于最大的字段大小。
- 可以规避c语言的类型系统，但也会产生很多讨厌的错误

例如：

一个二叉树，只有叶子节点有两个数据，内部节点没有数据。

```
union node_u{
        struct{
                union node_u *left;
                union node_u *right;
        }internal;
        double data[2];
}
```

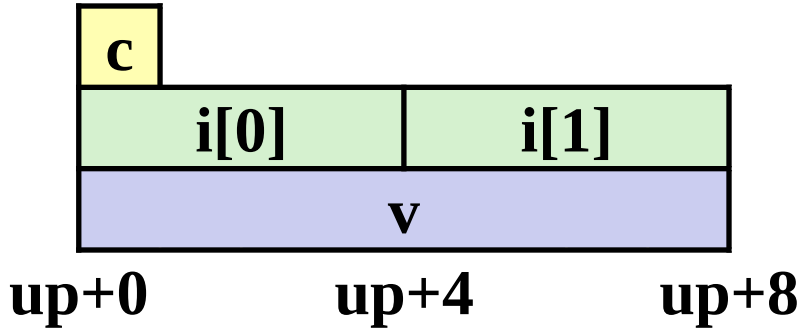
大小只有16字节。

如果n是指向节点的指针：n->internal.left和&data[0]指向同一块存储空间。

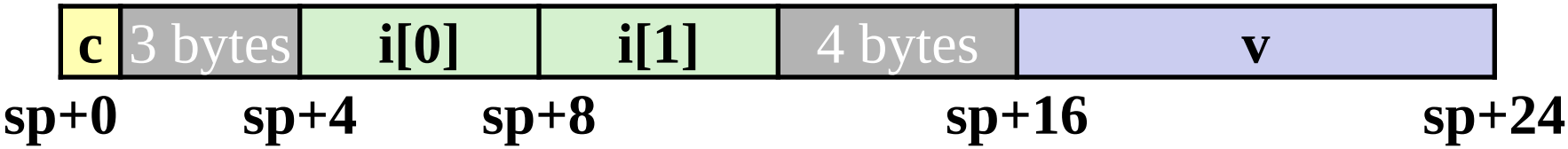
联合的内存分配

- 依据最大成员申请内存
- 同时只能使用一个成员

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

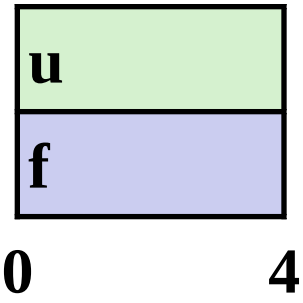



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



使用联合获取位模式（重新解读）

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




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

是否和(float)u 相同?

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

是否和(unsigned)f 相同?

字节序

- **想法**
  - short/long/quad words 在内存中用连续的2/4/8 字节存储
  - 哪个字节是最高/低位?
  - 在不同机器之间交换顺序，会有问题。
- **大端序(Big Endian)**
  - 最高有效位在低地址，如sun 的工作站Sparc
- **小端序(Little Endian)**
  - 最低有效位在低地址，如Intel x86、ARM Android、IOS
- **双端序(Bi Endian)**
  - 可配置成大/小端序，如ARM

字节序的例子

```
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]							

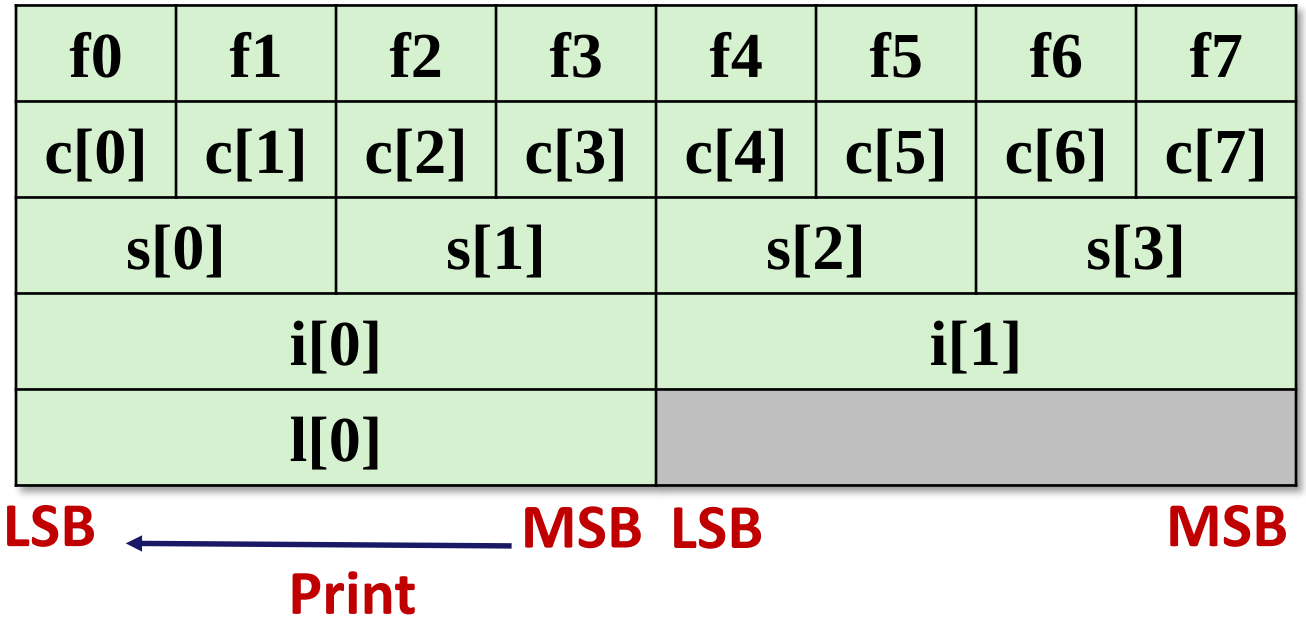
64-bit

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]							

```
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的字节序

小端序



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的字节序

大端序

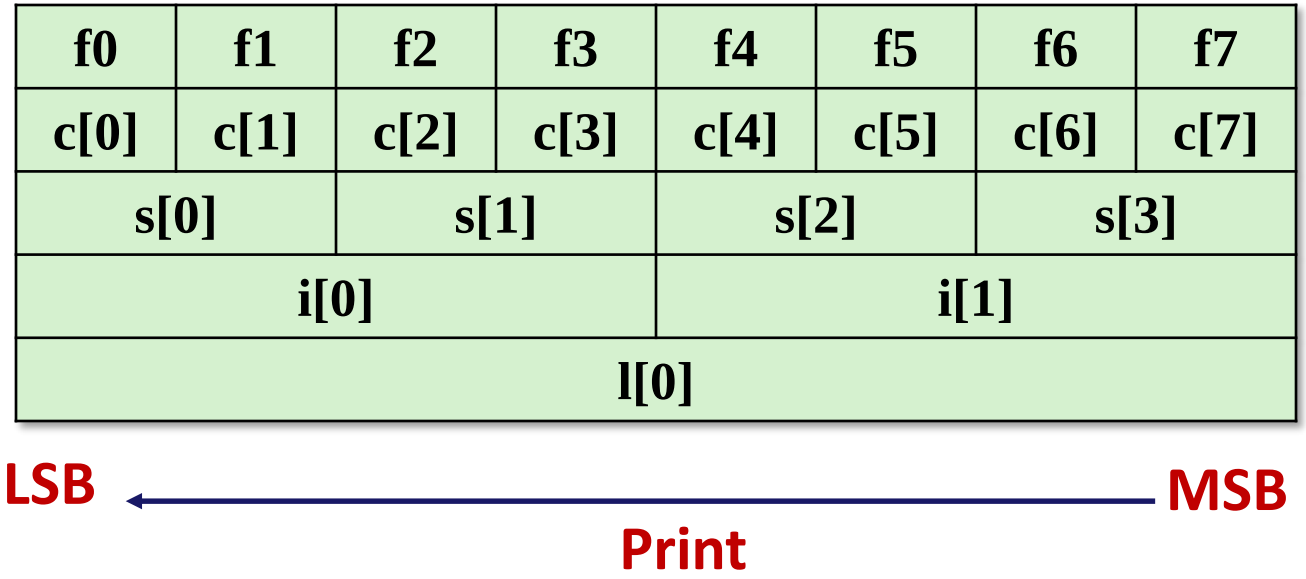
f0	f1	f2	f3	f4	f5	f6	f7
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

Sun机器的输出:

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的字节序

小端序



x86-64机器的输出

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]

经典例题

1.简述缓冲区溢出攻击的原理以及防范方法（5分）。

攻击原理（3个采分点）：

（1）程序输入缓冲区写入特定的数据，例如在gets读入字符串时

（2）使位于栈中的缓冲区数据溢出，用特定的内容覆盖栈中的内容，例如函数返回地址等

（3）使得程序在读入字符串，结束函数gets从栈中读取返回地址时，错误地返回到特定的位置，执行特定的代码，达到攻击的目的。

防范方法(2个采分点)：

（1）代码中避免溢出漏洞：例如使用限制字符串长度的库函数。

（2）随机栈偏移：程序启动后，在栈中分配随机数量的空间，将移动整个程序使用的栈空间地址。

（3）限制可执行代码的区域

（4）进行栈破坏检查——金丝雀

补充解释：对抗缓冲区溢出攻击4种方法

1）使用限制字符串长度的库函数。

2）栈随机化

栈随机化的思想使得栈的位置在程序每次运行时都有变化。程序开始时，在栈上分配一段0~n字节之间的随机大小的空间。分配的范围n必须足够大，才能获得足够多的栈地址变化，但是又要足够小，不至于浪费程序太多空间。

Linux系统中，栈随机化已经变成了标准行为。是更大的一类技术中的一种，这类技术称为地址空间布局随机化。每次运行时程序的不同部分，包括程序代码、库代码、栈、全局变量和堆数据，都会被加载到内存的不同区域。

3）限制可执行代码区域

消除攻击者向系统中插入可执行代码的能力，限制哪些内存区域能够存放可执行代码，只有保存编译器产生的代码的那部分才需要是可执行的，其他部分可以被限制为只允许读和写。

4）栈破坏检测

最近的GCC版本在产生的代码中加入了一种栈保护者机制，来检测缓冲区越界。其思想是在栈帧中任何局部缓冲区与栈状态之间存储一个特殊的值，也称哨兵值，是在程序每次运行时随机产生的。在恢复寄存器状态和从函数返回前，程序检查这个金丝雀值是否被该函数的某个操作或者该函数调用的某个函数的某个操作改变了。

经典例题

2. 简述C编译过程对非寄存器实现的int全局变量与非静态int局部变量处理的区别。包括存储区域、赋初值、生命周期、指令中寻址方式等。

	int全局变量	int局部变量
存储区域	数据段	堆栈段
赋初值	编译时 int x=1;	程序执行时，执行数据传送类指令如 MOVL $1234, 8(RSP)
生命周期	程序整个执行过程中都存在	进入子程序后在堆栈中存在(如执行subq $8, %rsp)子程序返回前清除消失
指令中寻址方式	其地址是个常数, 寻址如 movl 0x806808C, %eax	通过rsp/rbp的寄存器相对寻址方式。如类似 (%rsp) 或8(%rsp)或-8(%rbp)等

经典例题

3.当调用malloc这样的C标准库函数时，（)可以在运行时动态的扩展和收缩。

A. 堆　　B. 栈　　C. 共享库　　D. 内核虚拟存储器

答案：A 考点：malloc如何动态分配内存地

4.当函数调用时，（)可以在程序运行时动态地扩展和收缩。

A.程序代码和数据区 B. 栈　　C. 共享库　　D. 内核虚拟存储器

答案：B 考点：栈在程序运行时的状态

C语言复合类型总结

- **数组**
  - 连续分配内存
  - 对齐：满足每个元素对齐要求
  - 数组名是首个元素的指针常量
  - 没有越界检查!
- **结构体**
  - 各成员按结构体定义中的顺序分配内容
  - 在中间、末尾填充字节，以满足对齐要求
- **联合**
  - 覆盖的声明
  - 规避类型系统对编程束缚的方法

Enjoy!