



2022秋季

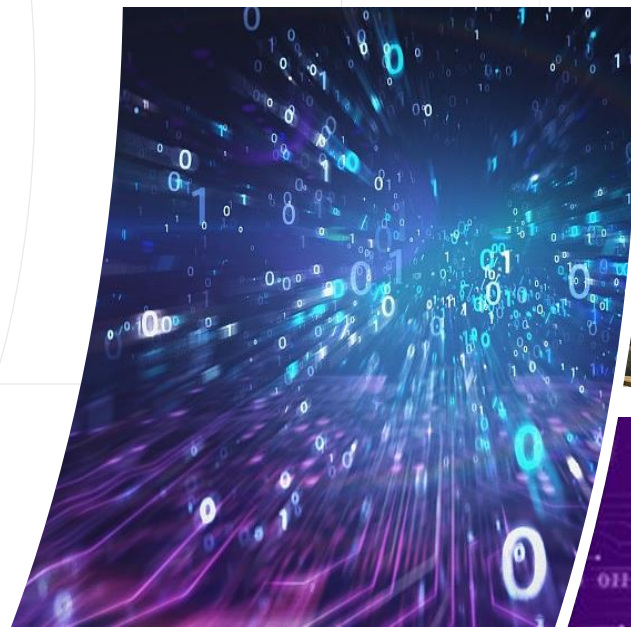
计算机系统概论

Introduction to Computer Systems

80X86汇编语言与 C语言-6

⊗ 韩文弢

✉ hanwentao@tsinghua.edu.cn



计算机体系结构

C程序在硬件层面的表示

-
- 数据/代码的内存布局
 - 栈、堆等各类数据段以及代码段的layout (第六讲)
 - 缓冲区溢出等 (第六讲)

```
int array[4] = {1, 2, 3, 4};
static int brray[4] = {1, 2, 3, 4};

static int intra_sum (int x[4], int y)
{
    return x[y-1];
}

int main()
{
    int val = intra_sum(array, 3) + inter_sum(brray, 3);
    return val;
}
```

main.c

编译

```
intra_sum:
    movslq    %esi, %rsi
    movl      -4(%rdi,%rsi,4), %eax
    ret

main:
    pushq     %rbx
    movl      $3, %esi
    movl      $array, %edi
    call      intra_sum
    movl      %eax, %ebx // val -> ebx
    movl      $3, %esi
    movl      $brray, %edi
    movl      $0, %eax
    call      inter_sum
    addl      %ebx, %eax // val -> eax
    popq      %rbx
    ret

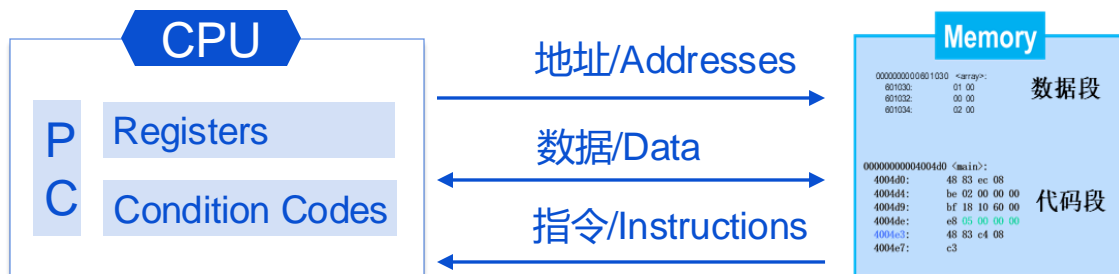
brray:
    .long     1
    .long     2
    .long     3
    .long     4

array:
    .long     1
    .long     2
    .long     3
    .long     4
```

汇编指令

main.o

链接



运行

```
0000000004004de <main>:
4004de: 53                push    %rbx
4004df: be 03 00 00 00    mov     $0x3,%esi
4004e4: bf 40 10 60 00    mov     $0x601040,%edi
4004e9: e8 e8 ff ff ff    callq   4004d6 <intra_sum>
4004ee: 89 c3             mov     %eax,%ebx
4004f0: be 03 00 00 00    mov     $0x3,%esi
4004f5: bf 30 10 60 00    mov     $0x601030,%edi
4004fa: b8 00 00 00 00    mov     $0x0,%eax
4004ff: e8 04 00 00 00    callq   400508 <inter_sum>
400504: 01 d8             add     %ebx,%eax
400506: 5b               pop     %rbx
400507: c3               retq
```

仅给出部分内容

内存地址

程序/数据在硬件层面的表示与运行



目录

CONTENTS

01
内存布局 (memory layout) ———>>

02
缓冲区溢出 (buffer overflow) >>

Linux进程的内存布局 (x86-64)

» Stack

- 运行栈 (默认大小为8MB)
- 例如, 局部变量

» Heap

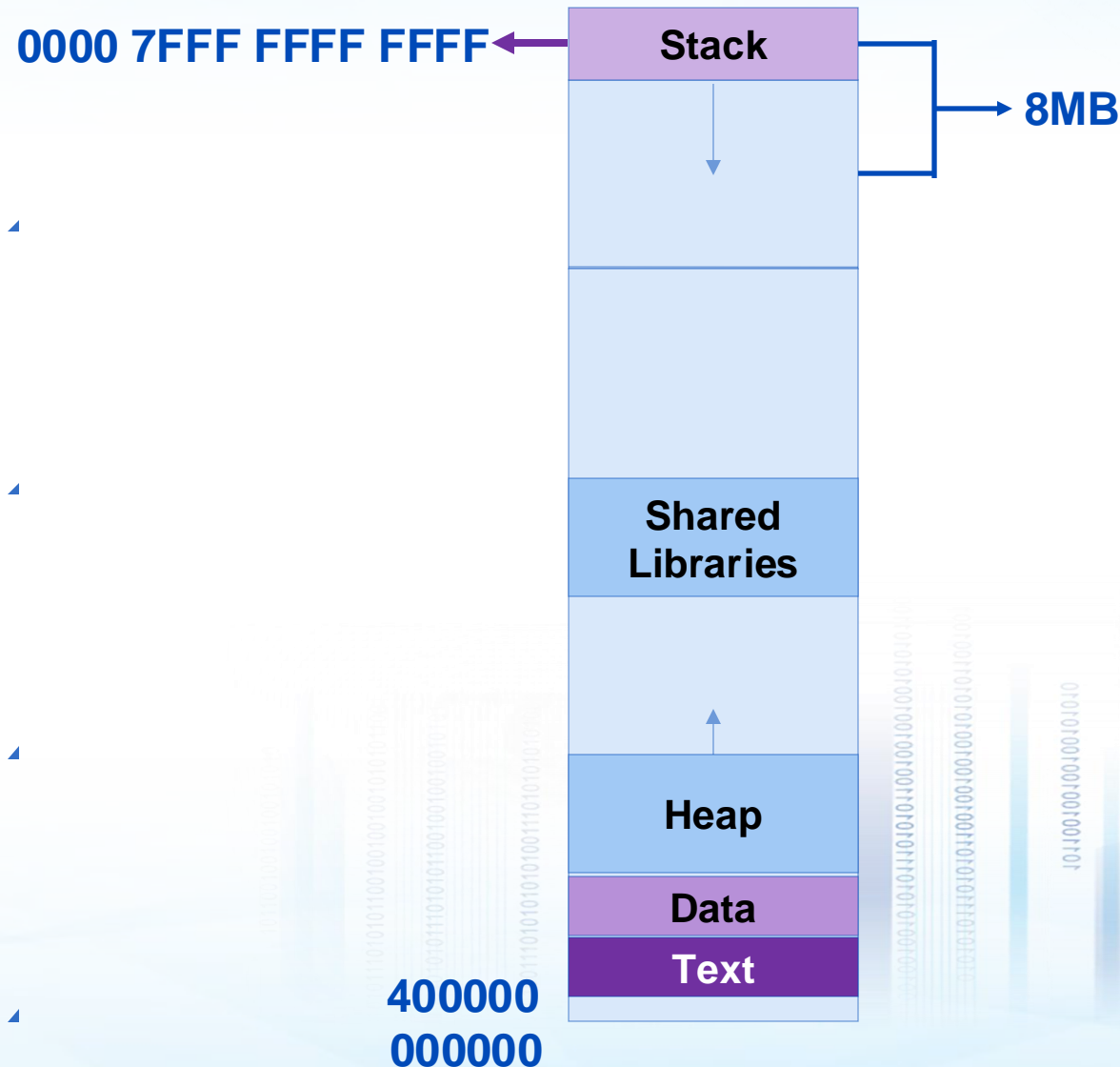
- 按需动态分配
- 通过 malloc(), calloc(), new 等分配使用

» Data

- 静态分配的数据空间
- 比如, 全局变量, 静态变量, 字符串常量

» Text / Shared Libraries

- 指令 (代码)
- Text只读



内存分配示例

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

int global = 0;

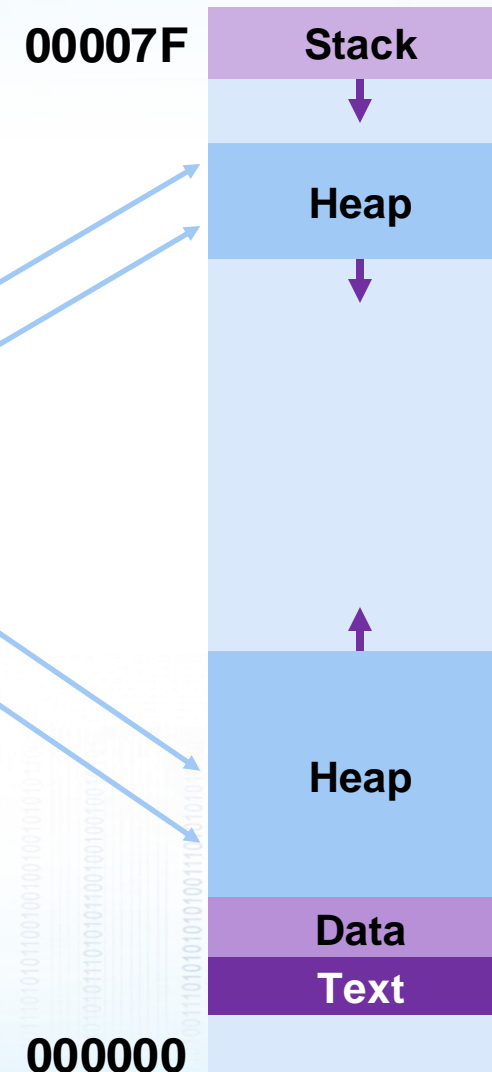
int useless() { return 0; }

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 ... */
}
```



示例地址

| 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 |





目录

CONTENTS

01
内存布局 (memory layout) ———>>

02
缓冲区溢出 (buffer overflow) >>

» Unix函数gets()的实现

- 无法指定要读取的字符数限制

» 其他字符串库函数也存在类似的问题

- strcpy, strcat: 复制任意长度的字符串
-
- 通过%s转义符使用scanf, fscanf, sscanf等

```
/* 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;
}
```


易受攻击的缓冲区相关代码

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

←多大算足够大?

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

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

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

易受攻击的缓冲区相关代码

echo:

0000000004006cf <echo>:

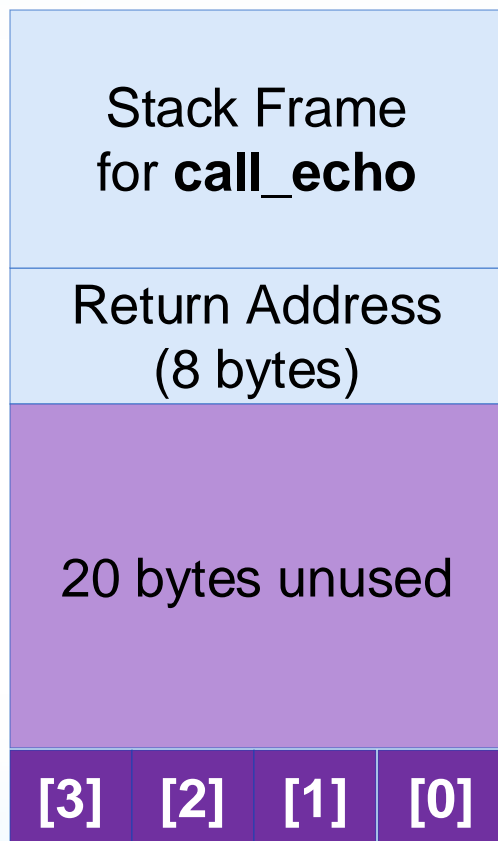
| | | | |
|---------|----------------|-------|-------------------|
| 4006cf: | 48 83 ec 18 | sub | \$0x18,%rsp |
| 4006d3: | 48 89 e7 | mov | %rsp,%rdi |
| 4006d6: | e8 a5 ff ff ff | callq | 400680 <gets> |
| 4006db: | 48 89 e7 | mov | %rsp,%rdi |
| 4006de: | e8 3d fe ff ff | callq | 400520 <puts@plt>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

缓冲区溢出时的栈

调用*gets*之前



buf ← %rsp

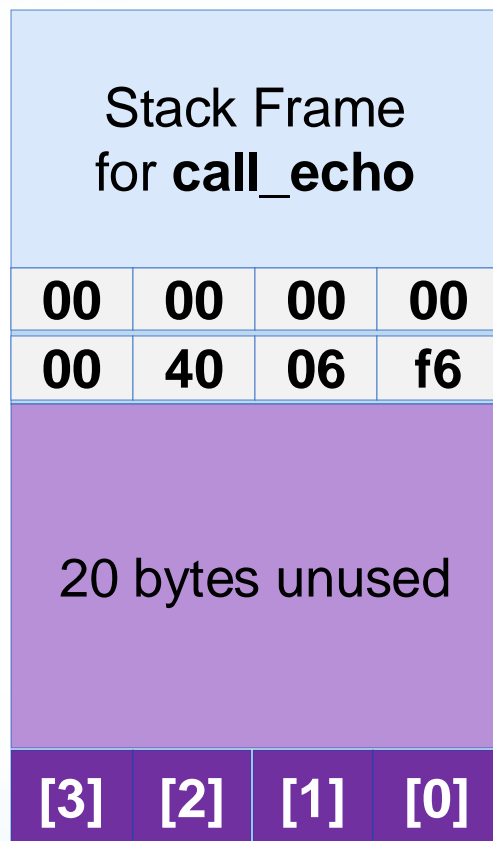
```
/* Echo Line */  
void echo()  
{  
    char buf[4]; /* Way too small! */  
    gets(buf);  
    puts(buf);  
}
```

echo:

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

缓冲区溢出时的栈

调用*gets*之前



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
- . . .

缓冲区溢出时的栈

调用*gets*之后

| 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 | 29 | 28 |
| 27 | 26 | 25 | 24 |
| 23 | 22 | 21 | 20 |

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
```

缓冲区溢出，但是程序运行状态没有被破坏

缓冲区溢出时的栈

调用*gets*之后

| 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 | 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: 0123456789012345678901234
Segmentation Fault
```

缓冲区溢出，且破坏了返回地址

缓冲区溢出时的栈

调用`gets`之后

| 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);
    ...
}
```

`call_echo:`

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

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

缓冲区溢出，且破坏了返回地址，但是程序看起来“正常”运行

缓冲区溢出时的栈

调用*gets*之后

| 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
```

- “返回”到无关指令
- 可能发生了很多事，但是程序看起来运行正常
- 最后通过retq返回主函数

代码注入攻击

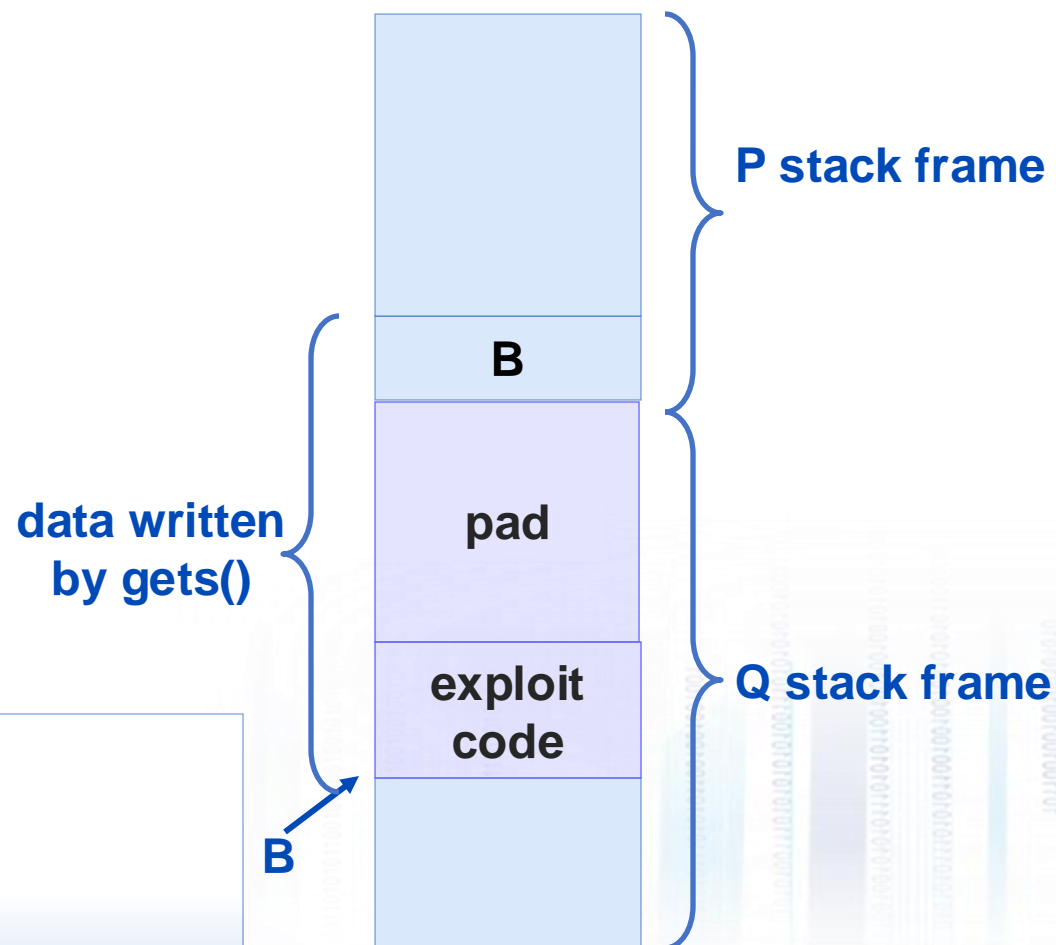
```
void P(){  
    Q();  
    ...  
}
```

return address A

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

- 输入字符串包含**可执行代码的字节表示**
- 用缓冲区B的地址覆盖函数返回地址A
- 当程序执行ret时，将跳转到“注入代码”

调用gets之后



代码注入攻击——最简单的原理示例

```
1 void test() {  
2     int val;  
3     val = getbuf();  
4     printf("No exploit.  Getbuf returned 0x%x\n", val);  
5 }
```

```
1 unsigned getbuf()  
2 {  
3     char buf[BUFFER_SIZE];  
4     Gets(buf);  
5     return 1;  
6 }
```

```
1 void touch1() {  
2     vlevel = 1;  
3     printf("Touch1!: You called touch1()\n");  
4     validate(1);  
5     exit(0);  
6 }
```

- 目标：函数test调用getbuf后，直接运行函数touch1，而不返回到函数test

(虚拟机下实例演示)

预防手段

» 预防手段 1

- ▶ 避免溢出漏洞

» 预防手段 2

- ▶ 采用系统级防护手段

» 预防手段 3

- ▶ 通过编译器采用“金丝雀”方法



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函数

- 使用fgets而不是gets
- 使用strncpy替代strcpy
- 不使用带有%s转义符的scanf
 - ▶ 使用fgets读取字符串
 - ▶ 或者使用%ns转义符，n指定长度

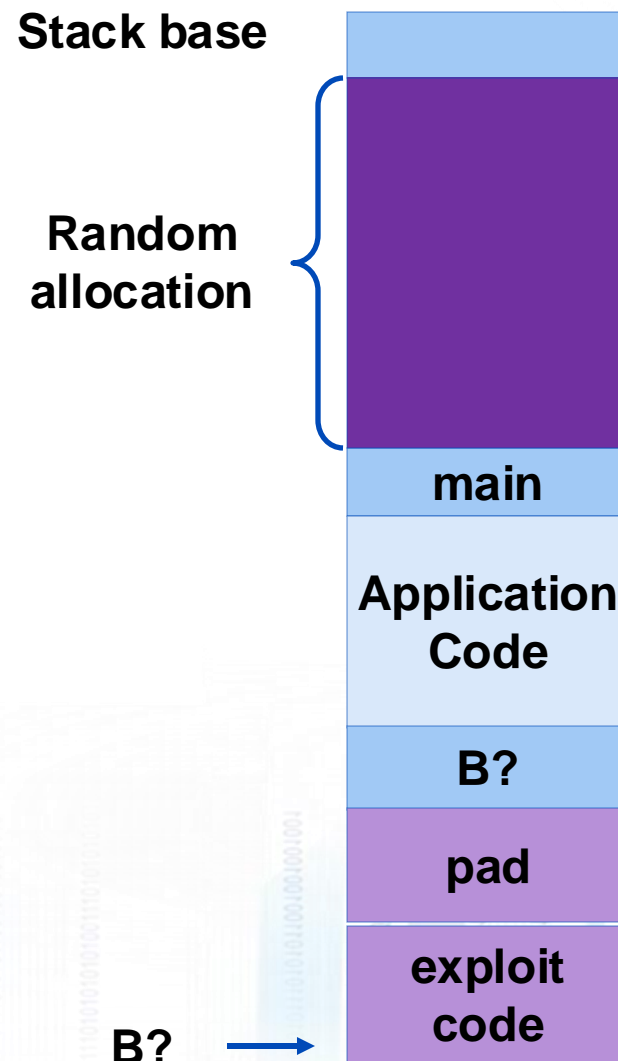
2.采用系统级防护手段

» 随机的stack base地址

- ◆ 在程序开始时，在栈上分配随机数量的空间
- ◆ 整个程序的栈被移动了
- ◆ 使得黑客很难预测插入代码的起始地址
- ◆ 比如，5次执行每一次的局部变量地址都不一样

| | | | | | |
|-------|----------------|----------------|----------------|----------------|----------------|
| local | 0x7ffe4d3be87c | 0x7fff75a4f9fc | 0x7ffeadb7c80c | 0x7ffeaea2fdac | 0x7ffcd452017c |
|-------|----------------|----------------|----------------|----------------|----------------|

- ▶ 因为，每次程序执行时栈都重新定位

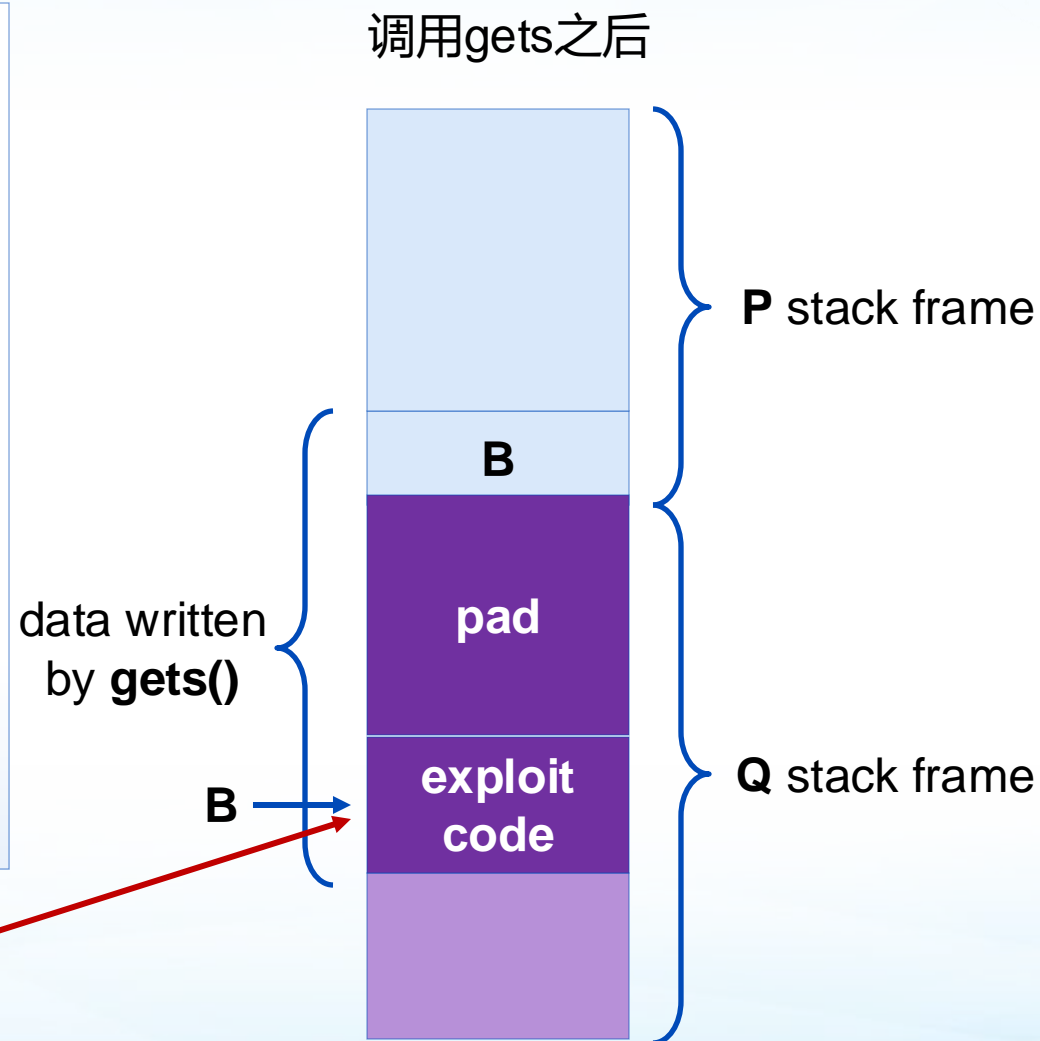


2.采用系统级防护手段

》 (处理器支持的) 不可执行段

- 在传统的x86中，只能将内存区域标记为“只读”或“可写”。
 - 这意味着可以执行任何可读的区域
- X86-64添加了显式的“可执行”权限
- 栈被标记为不可执行

任何执行此代码的尝试都将失败



3. “金丝雀” 方法

» 原理

- ◆ 将特殊值(“canary”)放在缓冲区之上的栈内
- ◆ 在退出函数之前检查该值是否被破坏

» GCC 的实现

- ◆ -fstack-protector
- ◆ 默认是打开的

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

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

3. “金丝雀” 方法

Linux系统下 fs:0x28存储一个特殊值用于“守卫”栈；fs:0x28这个地址在glibc中被定义为stack_chk_guard，相关的代码可能是这样的::

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
```

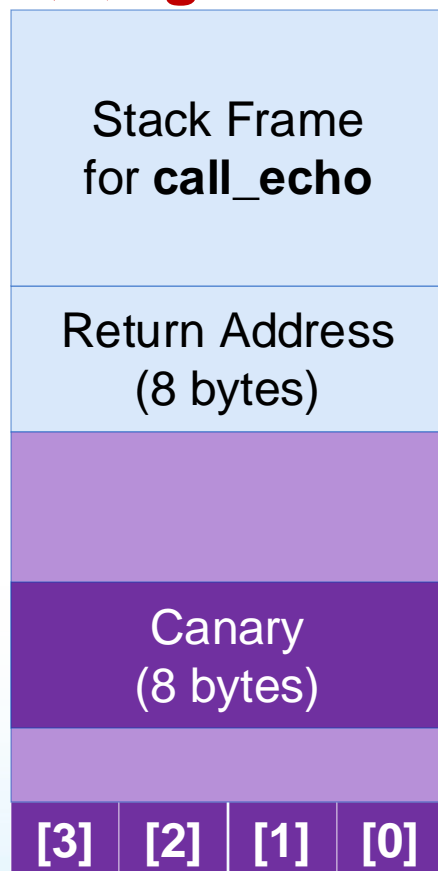
```
unsigned long __stack_chk_guard;
void __stack_chk_guard_setup(void)
{
    __stack_chk_guard = 0xBAAAAAAD; // provide some magic numbers
}

void __stack_chk_fail(void)
{
    /* Error message */
} // will be called when guard variable is corrupted
```


3. “金丝雀” 方法

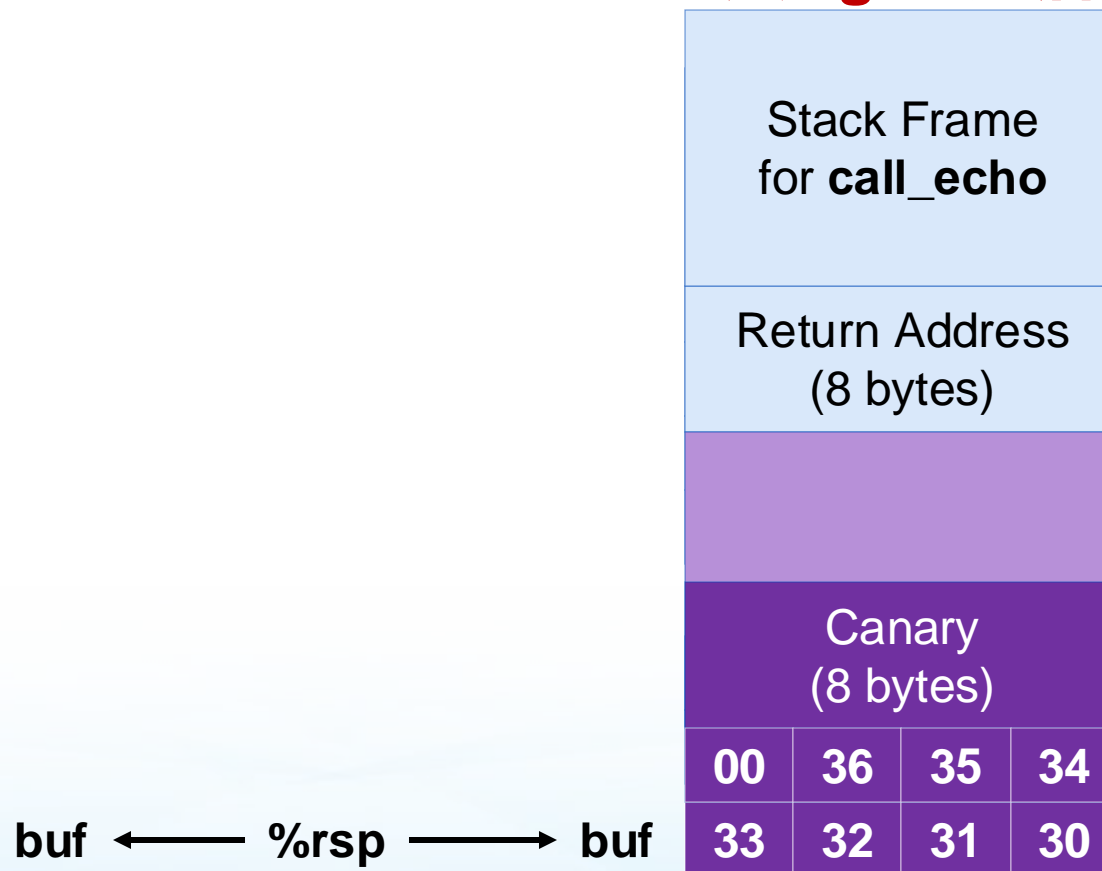
建立 “金丝雀”

调用gets之前



检查 “金丝雀”

调用gets之后



Return-Oriented Programming (ROP) 攻击

» 挑战 (对黑客而言)

- 栈的随机初始化位置使得很难预测缓冲区地址
- 将栈标记为不可执行使得很难插入二进制代码

» 替代策略

- 复用已有代码
 - ▶ 比如, 某些共享库代码
- 将现有代码的若干片段串在一起, 达到总体期望的结果
- 但是无法应对 “金丝雀”

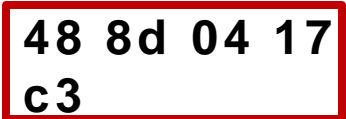
» 用gadgets (代码片段) 构造程序

- Gadget: 由指令ret结尾的代码片段
 - ▶ ret的编码即为0xc3
- 每次运行时代码位置都是固定的
- 可执行

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

使用现有函数的尾部

Gadget Example #2

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

这一段数字表示movq %rax, %rdi

<setval>:

4004d9: c7 07 d4 **48 89 c7** movl \$0xc78948d4, (%rdi)

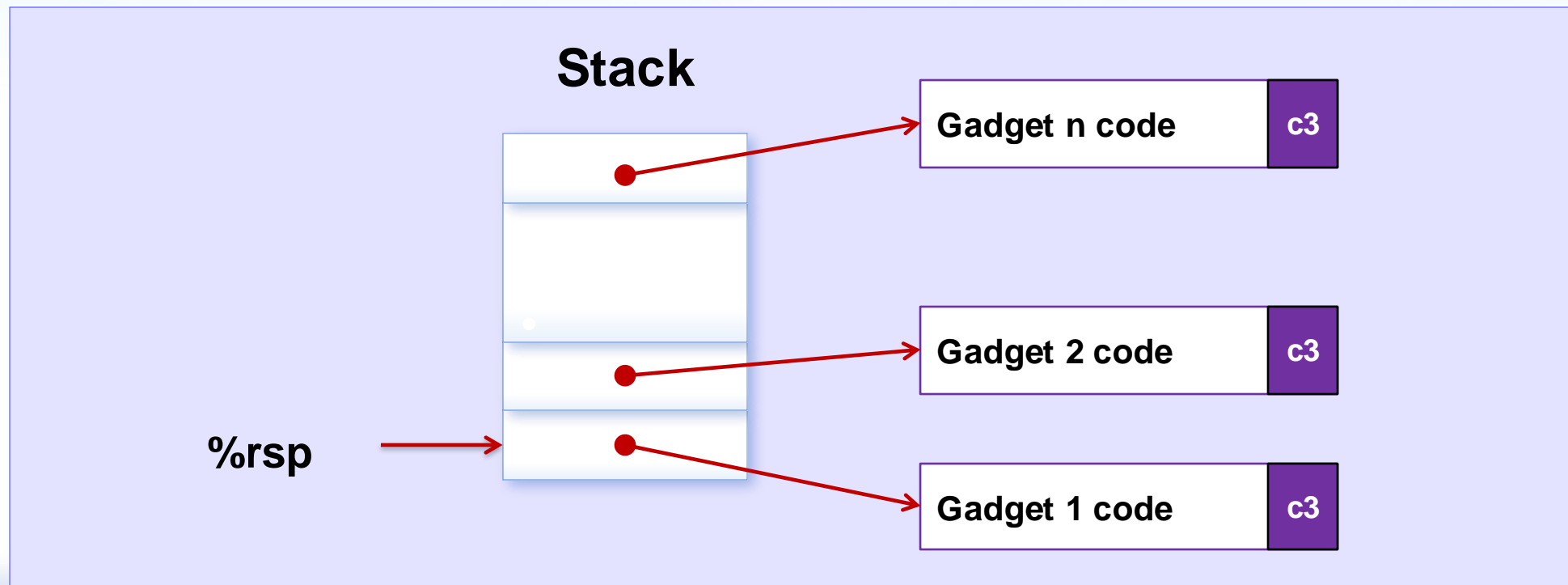
4004df: **c3** retq

rdi ← rax

Gadget address = 0x4004dc

字节码被重新解释

ROP的执行流程



- ◆ 由ret指令触发
 - ▶ 开始执行Gadget 1
- ◆ 每个Gadget中的最终ret指令触发下一个Gadget的执行