第3章 程序的机器级表示
Machine-Level Programming V: Advanced Topics

100076202： 计算机系统导论

V: 高级主题

V: Advanced Topics

任课教师：

宿红毅　张艳　黎有琦　颜珂

Carnegie Mellon University

原作者：

Randal E. Bryant and David R. O'Hallaron

议题

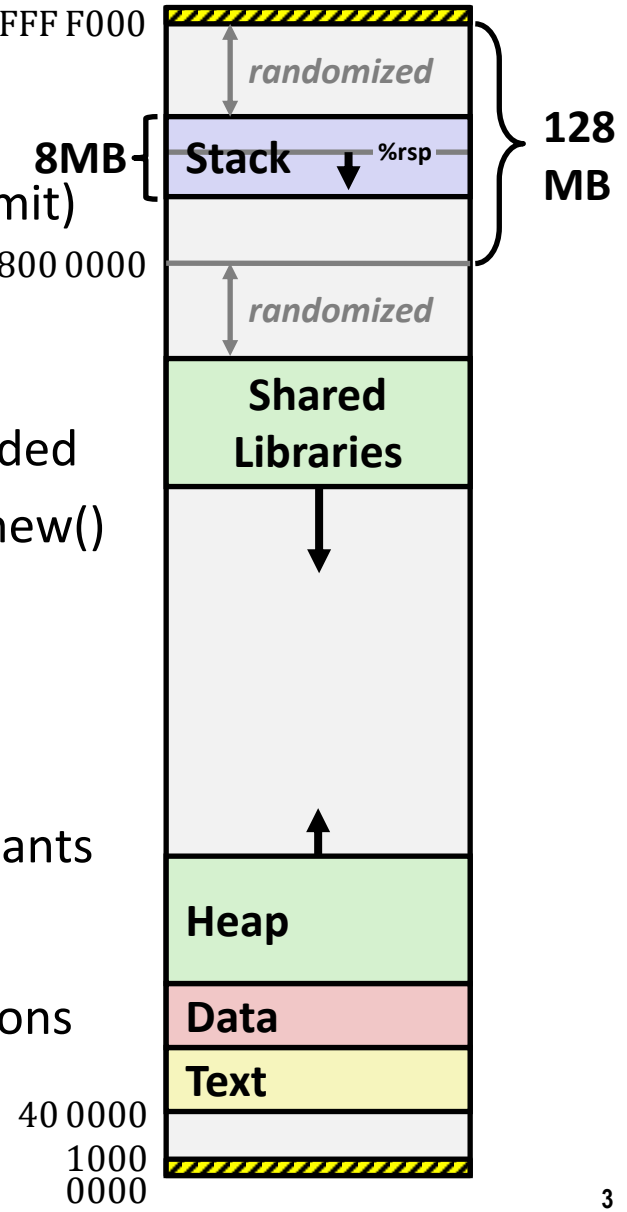
- **内存布局 Memory Layout**
- **缓冲区溢出 Buffer Overflow**
  - 漏洞 Vulnerability
  - 保护 Protection
- **联合 Unions**

x86-64Linux内存布局
x86-64 Linux Memory Layout

没有按照比例画图
not drawn to scale

- **栈 Stack**
  - 运行时栈（8MB限制）Runtime stack (8MB limit)
  - 例如局部变量 E. g., local variables
- **堆 Heap**
  - 需要时动态分配 Dynamically allocated as needed
  - 当调用函数时：When call malloc(), calloc(), new()
- **数据 Data**
  - 静态分配的数据 Statically allocated data
  - 例如全局变量、静态变量、串常量
    - E.g., global vars, `static` vars, string constants
- **文本 /共享库 Text / Shared Libraries**
  - 可执行机器指令 Executable machine instructions
  - 只读的 Read-only

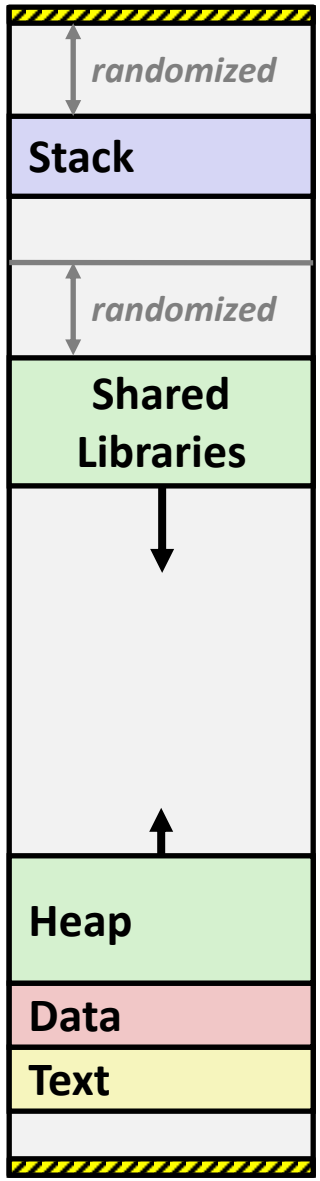
十六进制地址 Hex Address



内存分配示例
Memory Allocation Example

没有按比例画图
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 ... */
}
```



每个变量分布在哪里? Where does everything go?

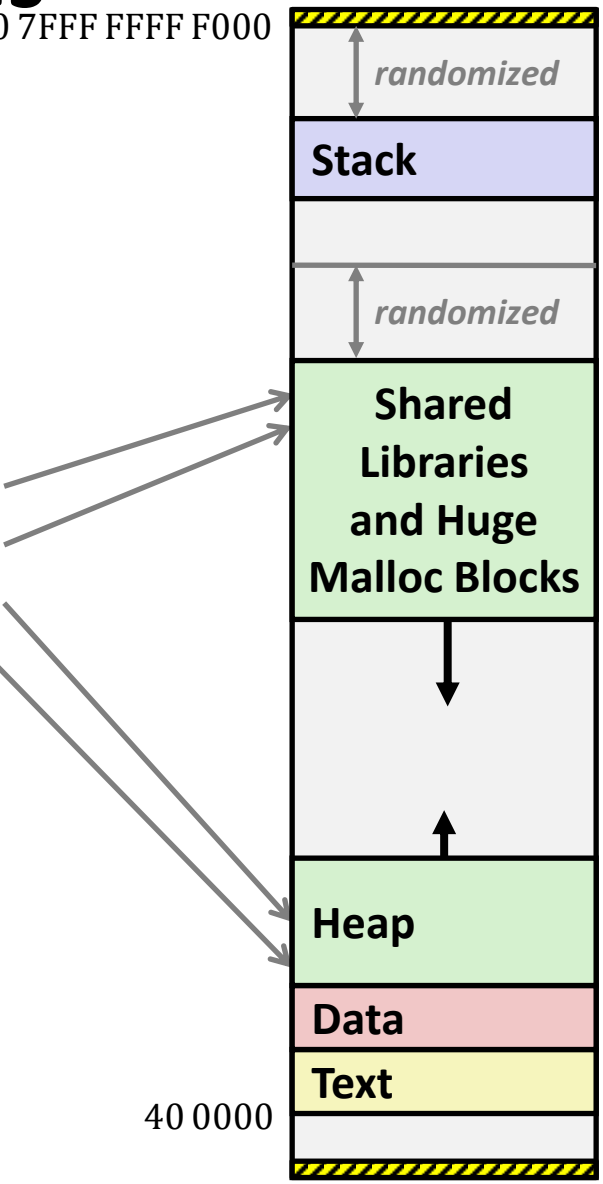
x86-64示例地址
x86-64 Example Addresses

没有按比例画图
not drawn to scale

地址范围 address range ~2^{47}

局部变量	`local`	`0x00007ffe4d3be87c`
动态分配	`phuge1`	`0x00007f7262a1e010`
动态分配	`phuge3`	`0x00007f7162a1d010`
动态分配	`psmall4`	`0x000000008359d120`
动态分配	`psmall2`	`0x000000008359d010`
静态分配	`big_array`	`0x0000000080601060`
静态分配	`huge_array`	`0x0000000000601060`
代码	`main()`	`0x000000000040060c`
代码	`useless()`	`0x0000000000400590`

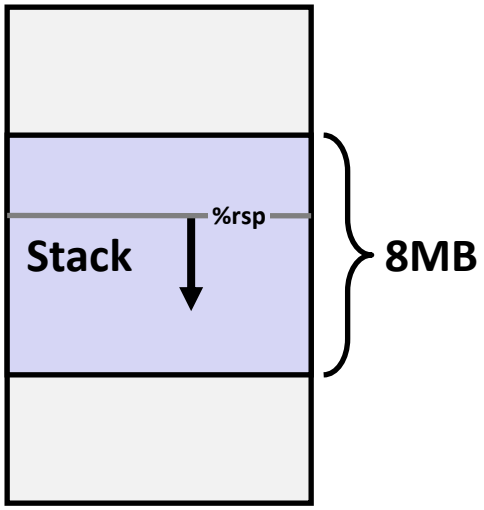
(准确值可能变化)
(Exact values can vary)



失控栈示例 Runaway Stack Example

not drawn to scale

```
int recurse(int x) {
    int a[1<<15];  // 4*2^15 =  128 KiB
    printf("x = %d.  a at %p\n", x, a);
    a[0] = (1<<14)-1;
    a[a[0]] = x-1;
    if (a[a[0]] == 0)
        return -1;
    return recurse(a[a[0]]) - 1;
}
```



- **函数存储局部变量在栈帧中**
  Functions store local data in stack frame
- **递归函数引起深度帧嵌套**
  Recursive functions cause deep nesting of frames
- **当空间用尽时发生什么?**
  What happens when we run out of space?

```
./runaway 67
x = 67.  a at 0x7ffd18aba930
x = 66.  a at 0x7ffd18a9a920
x = 65.  a at 0x7ffd18a7a910
x = 64.  a at 0x7ffd18a5a900
. . .
x = 4.  a at 0x7ffd182da540
x = 3.  a at 0x7ffd182ba530
x = 2.  a at 0x7ffd1829a520
Segmentation fault (core dumped)
```

议题

- 内存布局 Memory Layout
- **缓冲区溢出 Buffer Overflow**
  - 漏洞 Vulnerability
  - 防护 Protection
- 联合 Unions

回忆：内存引用错误示例
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)   ->    3.1400000000
fun(1)   ->    3.1400000000
fun(2)   ->    3.1399998665
fun(3)   ->    2.0000006104
fun(6)   ->    Stack smashing detected  检测到栈击穿
fun(8)   ->    Segmentation fault       段故障
```

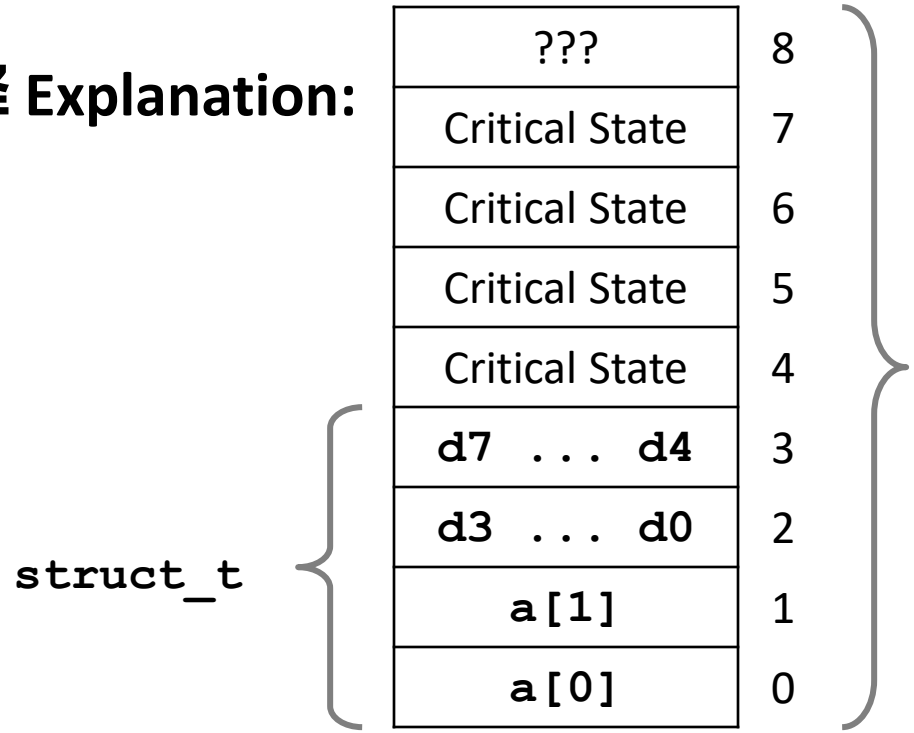
- 结果随着系统而不同 Result is system specific

内存引用错误示例
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(4)   ->    Segmentation fault 段故障
fun(8)   ->    3.1400000000
```

解释 Explanation:



Fun(i)函数访问的位置
Location accessed by
fun(i)

这类问题是一个大麻烦
Such problems are a BIG deal

- **一般称为“缓冲区溢出” Generally called a “buffer overflow”**
  - 当超过为数组分配的内存大小时 when exceeding the memory size allocated for an array
- **为何是个大麻烦？ Why a big deal?**
  - 它是安全漏洞的头号技术原因 It’s the #1 technical cause of security vulnerabilities
    - 从整体看，导致安全漏洞的第一主要原因是**社会工程/用户忽视** #1 overall cause is social engineering / user ignorance
- **最常见的形式 Most common form**
  - 字符串输入没有检查长度 Unchecked lengths on string inputs
  - 特别是对于栈中有界字符数组 Particularly for bounded character arrays on the stack
    - 有时简称栈击穿 sometimes referred to as stack smashing

基于缓冲区溢出进行攻击
Exploits Based on Buffer Overflows

- ***缓冲区溢出错误能够允许远程机器在受害机器上执行任意代码***
  Buffer overflow bugs can allow remote machines to execute arbitrary code on victim machines
- **非常悲惨，在真实程序中很常见 Distressingly common in real** progams
  - 程序员老是犯同样的错误 Programmers keep making the same mistakes ☹
  - 最近的手段使这种攻击更加困难 Recent measures make these attacks much more difficult
- **这种例子有几十年历史了 Examples across the decades**
  - 原始的“互联网蠕虫”（1988年） Original “Internet worm” (1988)
  - “IM大战”（1999年） “IM wars” (1999)
  - 黄昏黑客（游戏Wii-任天堂） （2000年）Twilight hack on Wii (2000s)
  - 。。。而且还有很多 … and many, many more
- **在attacklab中你会学到一些技巧 You will learn some of the tricks in attacklab**
  - 希望能够说服你从不在程序中留下这类漏洞 Hopefully to convince you to never leave such holes in your programs!!

示例：最初的互联网蠕虫（1988年）
Example: the original Internet worm (1988)

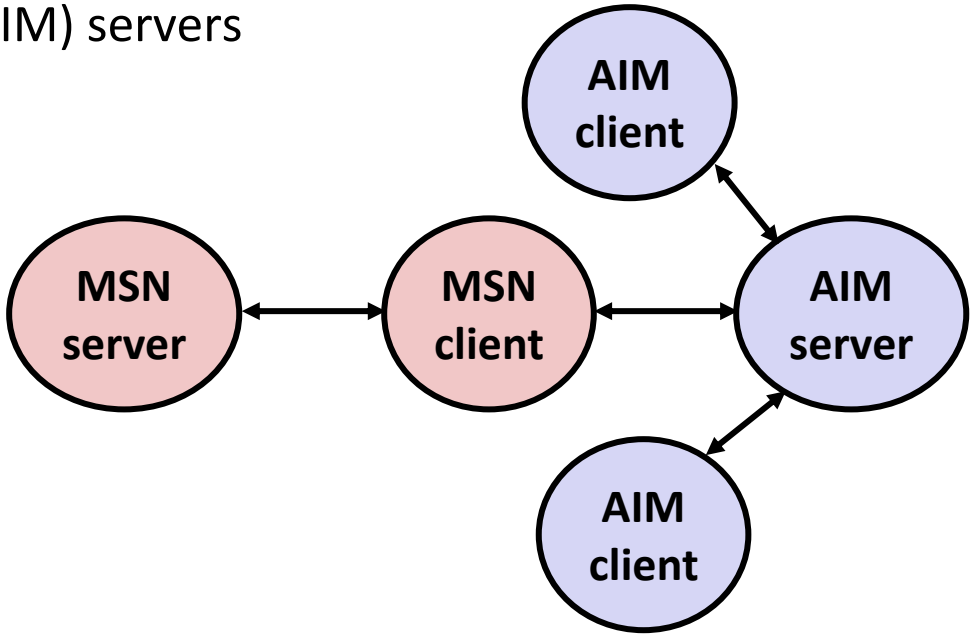
- **攻击几个漏洞进行扩散 Exploited a few vulnerabilities to spread**
  - finger服务器（fingerd）的早期版本使用gets()读取客户发送的参数 Early versions of the finger server (fingerd) used **`gets()`** to read the argument sent by the client:
    - **`finger droh@cs.cmu.edu`**
  - 蠕虫通过发送伪造参数攻击fingerd服务器 Worm attacked fingerd server by sending phony argument:
    - **`finger "exploit-code padding new-return-address"`**
    - 利用漏洞代码：在与攻击者有直接TCP连接的受害计算机上执行根shell。exploit code: executed a root shell on the victim machine with a direct TCP connection to the attacker.

示例：最初的互联网蠕虫（1988年）
Example: the original Internet worm (1988)

- **一旦进入一台机器，就可以扫描其它机器进行攻击 Once on a machine, scanned for other machines to attack**
  - 在几个小时内入侵了超过6000台计算机（占互联网的10%）invaded ~6000 computers in hours (10% of the Internet ☺)
    - 参见1989年6月ACM通讯上的文章 see June 1989 article in *Comm. of the ACM*
  - 蠕虫的年轻作者被起诉 the young author of the worm was prosecuted…
  - 而且组建了CERT。。。目前还挂靠在CMU。 and CERT was formed… still homed at CMU
  - CERT一般指计算机安全应急响应组。

示例2：IM大战 Example 2: IM War

- **1999年7月 July, 1999**
  - 微软发布了MSN Messenger（即时消息通信系统） Microsoft launches MSN Messenger (instant messaging system).
  - Messenger客户可以访问流行的AOL即时消息通信服务（AIM）服务器 Messenger clients can access popular AOL Instant Messaging Service (AIM) servers



IM大战（续） IM War (cont.)

- **1999年8月 August 1999**
  - Messenger客户非常神秘地不再能访问AIM服务器 Mysteriously, Messenger clients can no longer access AIM servers
  - 微软和AOL开始IM大战 Microsoft and AOL begin the IM war:
    - AOL改变服务器来屏蔽Messenger客户 AOL changes server to disallow Messenger clients
    - 微软改变客户端来战胜AOL的改变 Microsoft makes changes to clients to defeat AOL changes
    - 最少13次这类小冲突 At least 13 such skirmishes

IM大战（续） IM War (cont.)

- **1999年8月 August 1999**
  - 实际发生了什么情况？What was really happening?
    - AOL发现了一个缓冲区溢出漏洞在自己的AIM客户端 AOL had discovered a buffer overflow bug in their own AIM clients
    - 他们攻击该漏洞以检测和阻止微软客户端：攻击代码返回一个4字节的签名给服务器（这些字节在AIM客户端的某个位置）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
    - 当微软改变代码以匹配这个签名时，AOL再改变签名的位置 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.

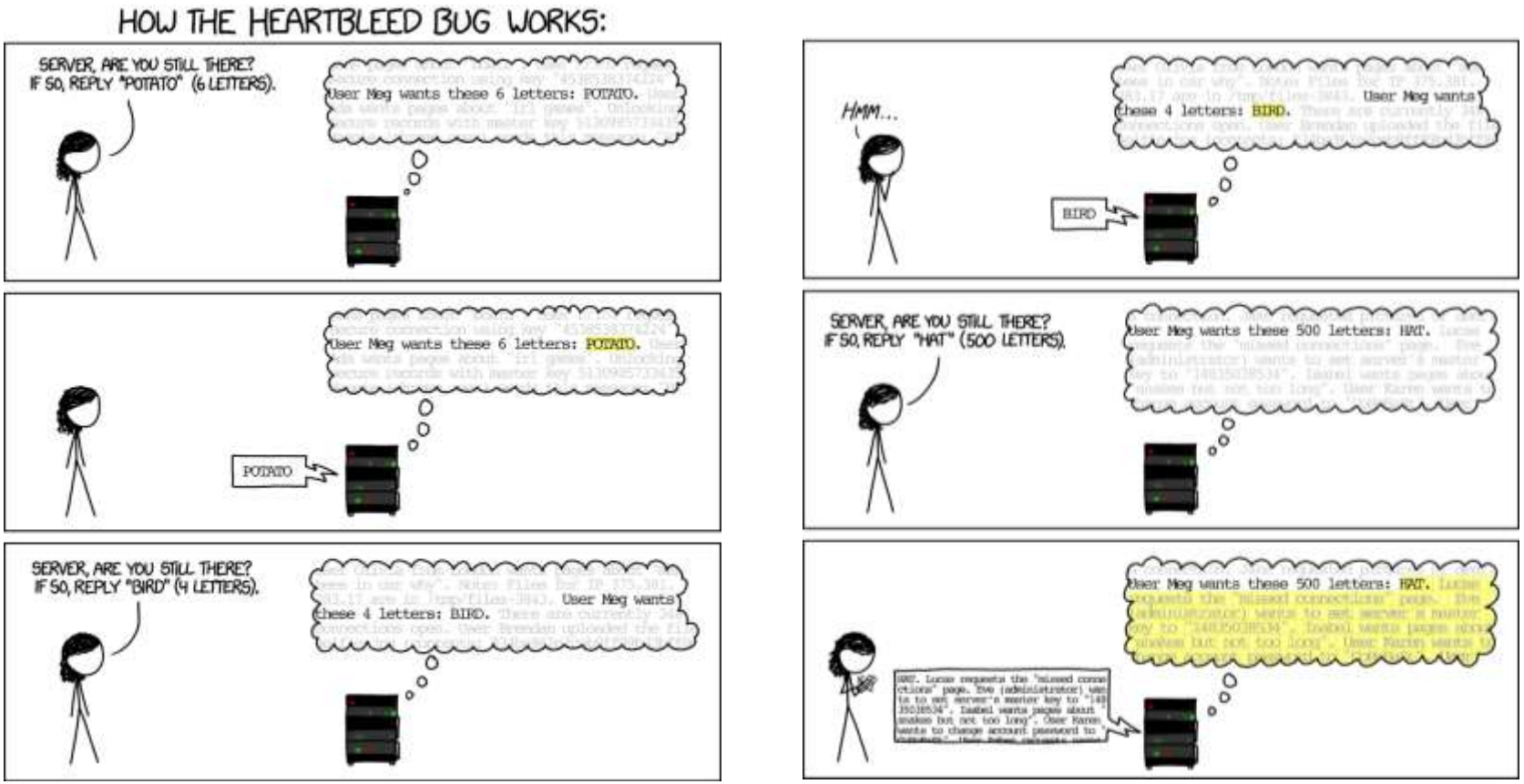
Sincerely,
Phil Bucking
Founder, Bucking Consulting
philbucking@yahoo.com

后来才得知这封邮件原始来自微软内部

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

程序员一直犯这些错误---心血来潮

Programmers keep making these mistakes…



https://xkcd.com/1354/

旁注：木马、蠕虫、病毒，。。。
Aside: Trojans, Worms, Viruses, …

- **三种不同类型的malware（恶意软件） Three different kinds of *malware* (malicious software)**
  - 按照传播方式进行分类 Categorized by how they spread
  - 随着时间的推移分界线变得模糊 Lines have gotten fuzzier over time
- **一个木马诱使人们运行它 A *trojan* tricks people into running it**
  - 取名于特洛伊木马传说 Named after the legend of the Trojan Horse
- **蠕虫会自动传播，无需人为操作 A *worm* spreads automatically, without human action**
  - 需要一种通过网络复制和执行自身的方法 Requires a way to copy and execute itself over the network
- **病毒控制已安装的程序 A *virus* takes control of programs that are already installed**
  - 类似生物病毒 Like a biological virus

字符串库代码 String Library Code

- **Unix函数gets()的实现 Implementation of Unix function `gets()`**

```
/* 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;
}
```

  - 没办法指定读取字符数的限制 No way to specify limit on number of characters to read

- **类似的问题在其它库函数也存在 Similar problems with other library functions**
  - **`strcpy`, `strcat`**: 复制任意长度的串 Copy strings of arbitrary length
  - **`scanf`, `fscanf`, `sscanf`**, 当给定%s转换说明时 when given **`%s`** conversion specification

有漏洞的缓冲区代码
Vulnerable Buffer Code

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

←顺便说一下，多大才足够大 btw, how big is big enough?

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

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

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

缓冲区溢出反汇编
Buffer Overflow Disassembly

echo:

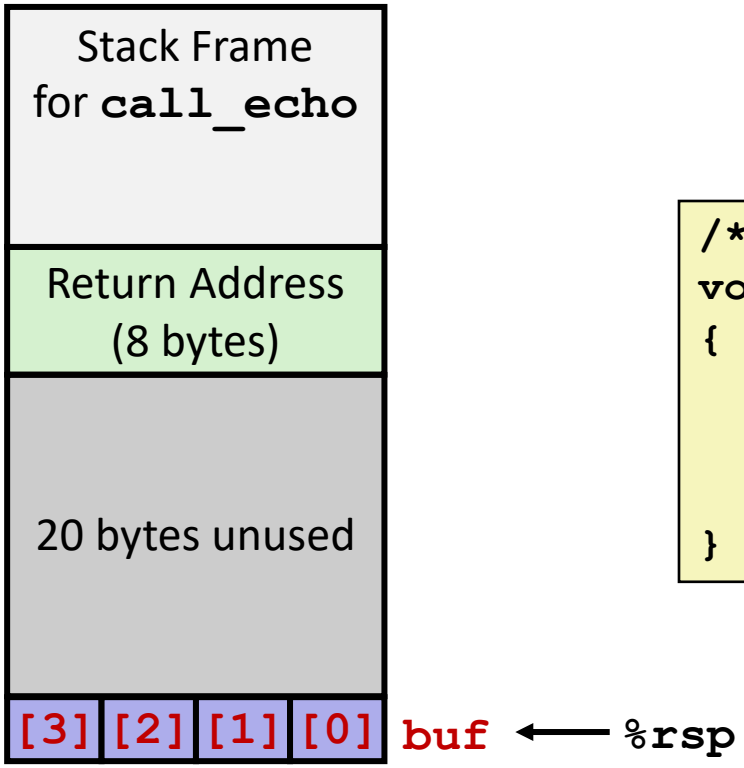
```
0000000004006cf <echo>:
 4006cf:  48 83 ec 18           sub    $0x18,%rsp
 4006d3:  48 89 e7              mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff        callq  400680 <gets>
 4006db:  48 89 e7              mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff        callq  400520 <puts@plt>
 4006e3:  48 83 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
```

缓冲区溢出栈示例
Buffer Overflow Stack Example

调用gets之前 Before call to gets

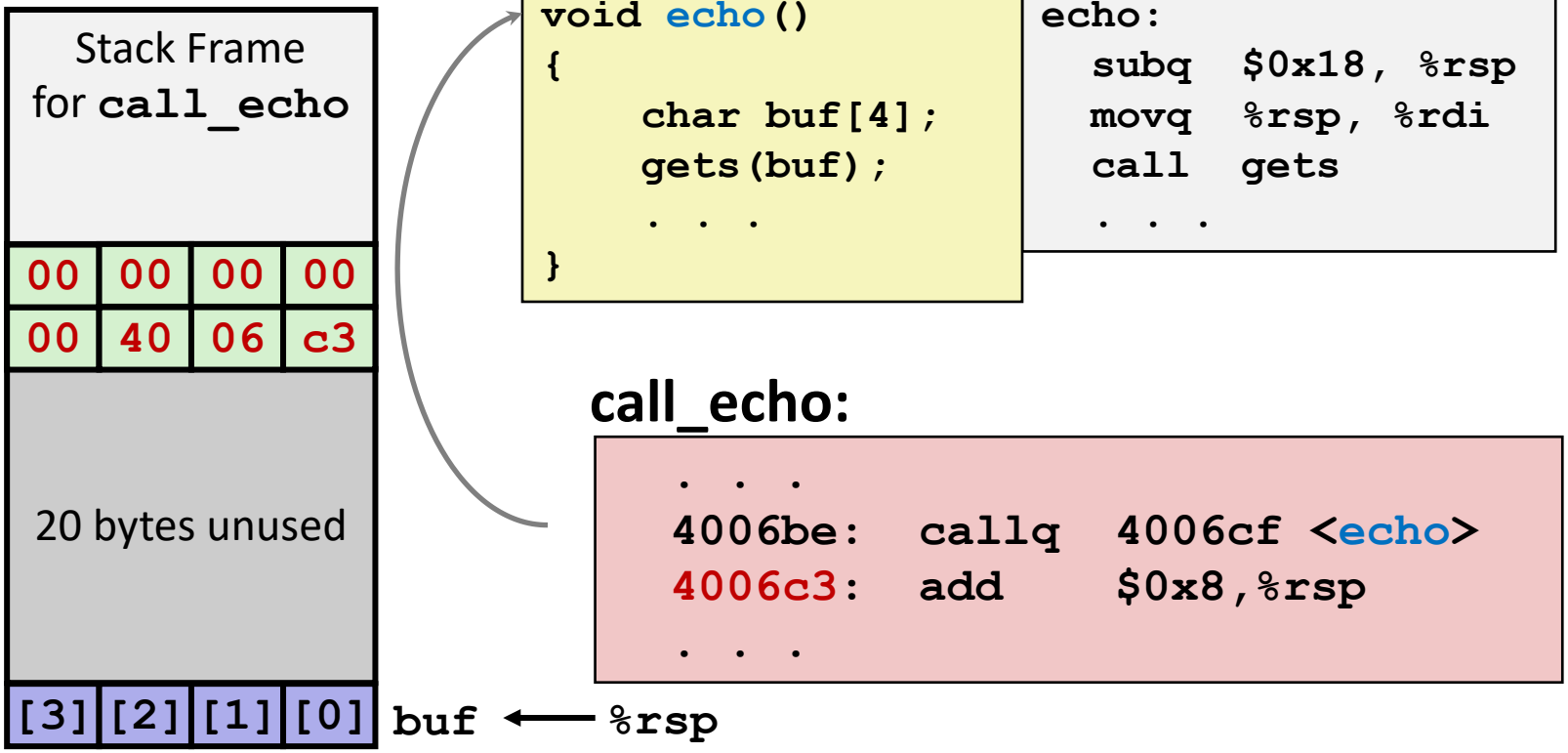



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq $0x18, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

缓冲区溢出栈示例
Buffer Overflow Stack Example

调用gets之前 Before call to gets



缓冲区溢出栈示例 #1

Buffer Overflow Stack Example #1

调用gets后 After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	c3
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   $0x18, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006be:   callq   4006cf <echo>
   4006c3:   add     $0x8,%rsp
   . . .
```

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

"012345678901234567890123\0"

溢出的缓冲区，但是没有破坏状态 Overflowed buffer, but did not corrupt state

缓冲区溢出栈示例 #2
Buffer Overflow Stack Example #2

调用gets后 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   $0x18, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006be:   callq   4006cf <echo>
   4006c3:   add     $0x8,%rsp
   . . .
```

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

程序"返回"到0x400600，然后崩溃 Program "returned" to 0x0400600, and then crashed.

缓冲区溢出栈示例#3

Buffer Overflow Stack Example #3

调用gets后 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:0123456789012345678901 23
0123456789012345678901 23
```

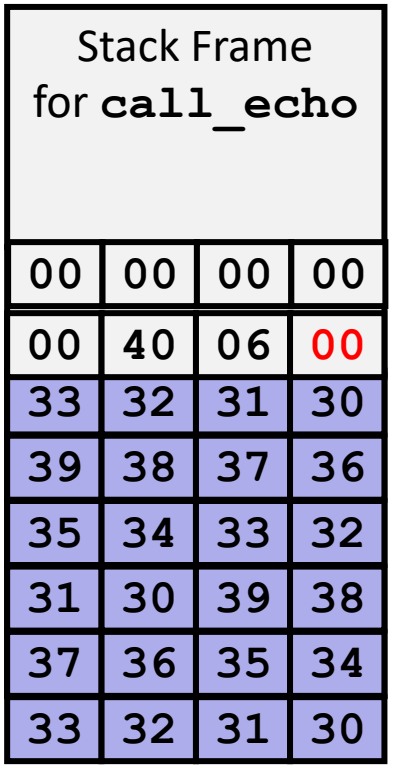
溢出的缓冲区，破坏了返回指针，但程序看起来还能工作

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

缓冲区溢出栈示例#3 解释

Buffer Overflow Stack Example #3 Explained

调用gets后 After call to gets



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”到不相关的代码 “Returns” to unrelated code

发生很多事情，没有修改关键状态 Lots of things happen, without modifying critical state

最终执行retq返回到main函数 Eventually executes `retq` back to `main`

栈击穿攻击 Stack Smashing Attacks

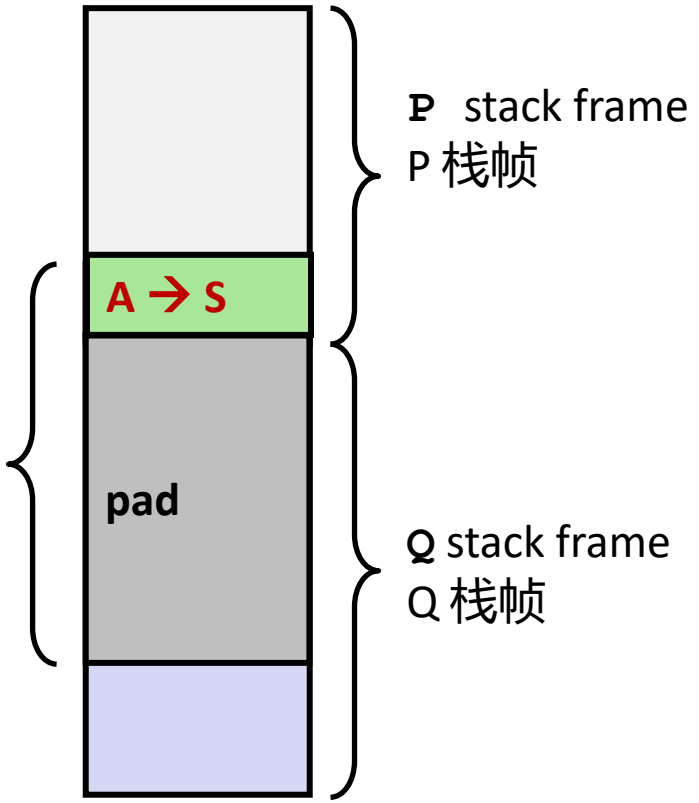
```
void P(){
  Q();
  ...
}
```

返回地址
return
address
A

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

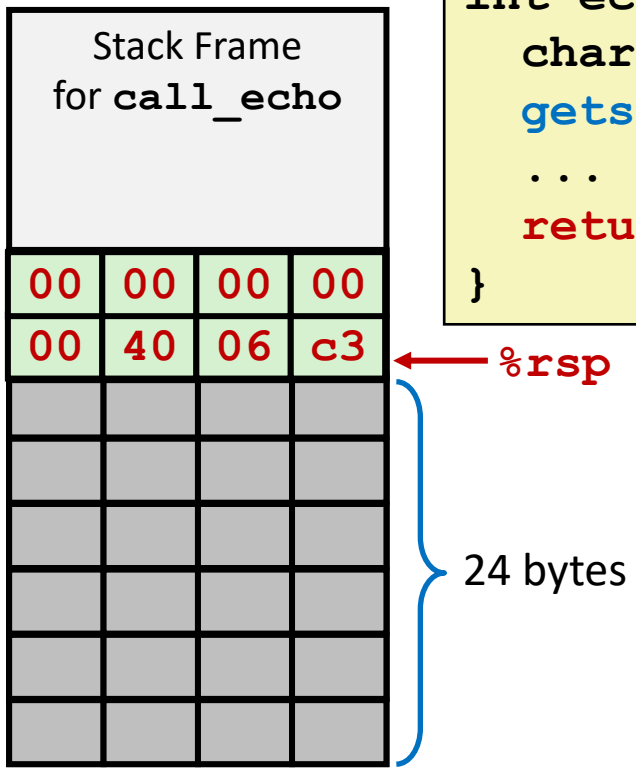
```
void S(){
/* Something
   unexpected */
  ...
}
```

调用gets后的栈 Stack after call to **gets()**



- **用某些其它代码S的地址覆盖正常的返回地址A Overwrite normal return address A with address of some other code S**
- **当Q执行ret时，会跳转到其它代码 When Q executes `ret`, will jump to other code**

制作击穿字符串 Crafting Smashing String




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

目标处的代码 Target Code

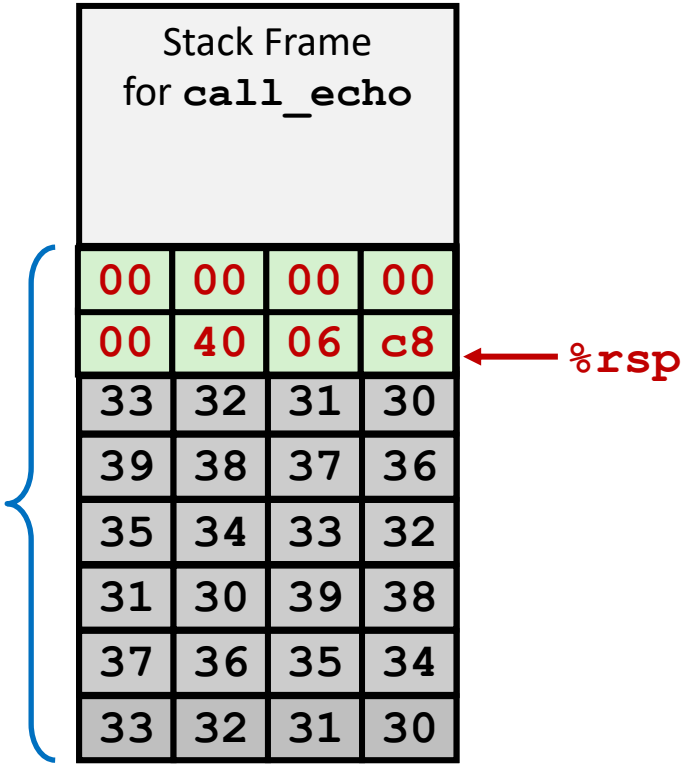
```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

```
00000000004006c8 <smash>:
  4006c8:       48 83 ec 08
```

攻击字符串（十六进制） 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
c8 06 40 00 00 00 00 00
```

击穿字符串的效果 Smashing String Effect



目标处的代码 Target Code

```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

```
00000000004006c8 <smash>:
  4006c8:        48 83 ec 08
```

攻击字符串（十六进制）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
c8 06 40 00 00 00 00 00
```

执行栈击穿 Performing Stack Smash

```
linux> cat smash-hex.txt
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33 c8 06 40 00 00 00 00 00
linux> cat smash-hex.txt | ./hexify | ./bufdemo-nsp
Type a string:012345678901234567890123?@
I've been smashed!
```

- **在文件smash-hex.txt中放入十六进制序列 Put hex sequence in file smash-hex.txt**
- **用hexify程序转换十六进制数字成为字符 Use hexify program to convert hex digits to characters**
  - 其中有些是不可显示的 Some of them are non-printing
- **提供给漏洞攻击程序作为输入 Provide as input to vulnerable program**

```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
c8 06 40 00 00 00 00 00
```

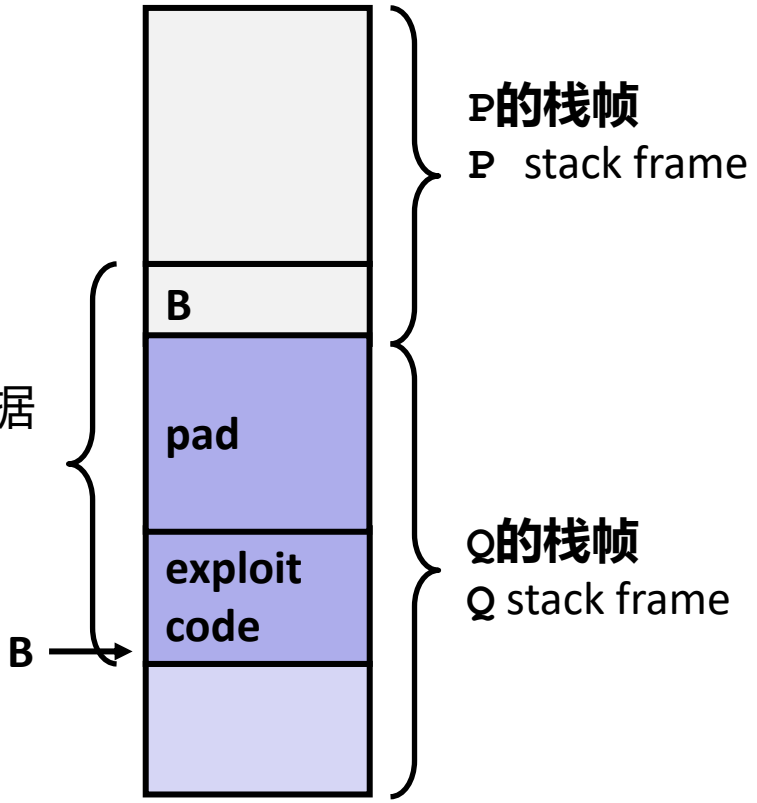
代码注入攻击 Code Injection Attacks

调用gets之后的栈 Stack after call to **gets()**

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

返回地址 return address A

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

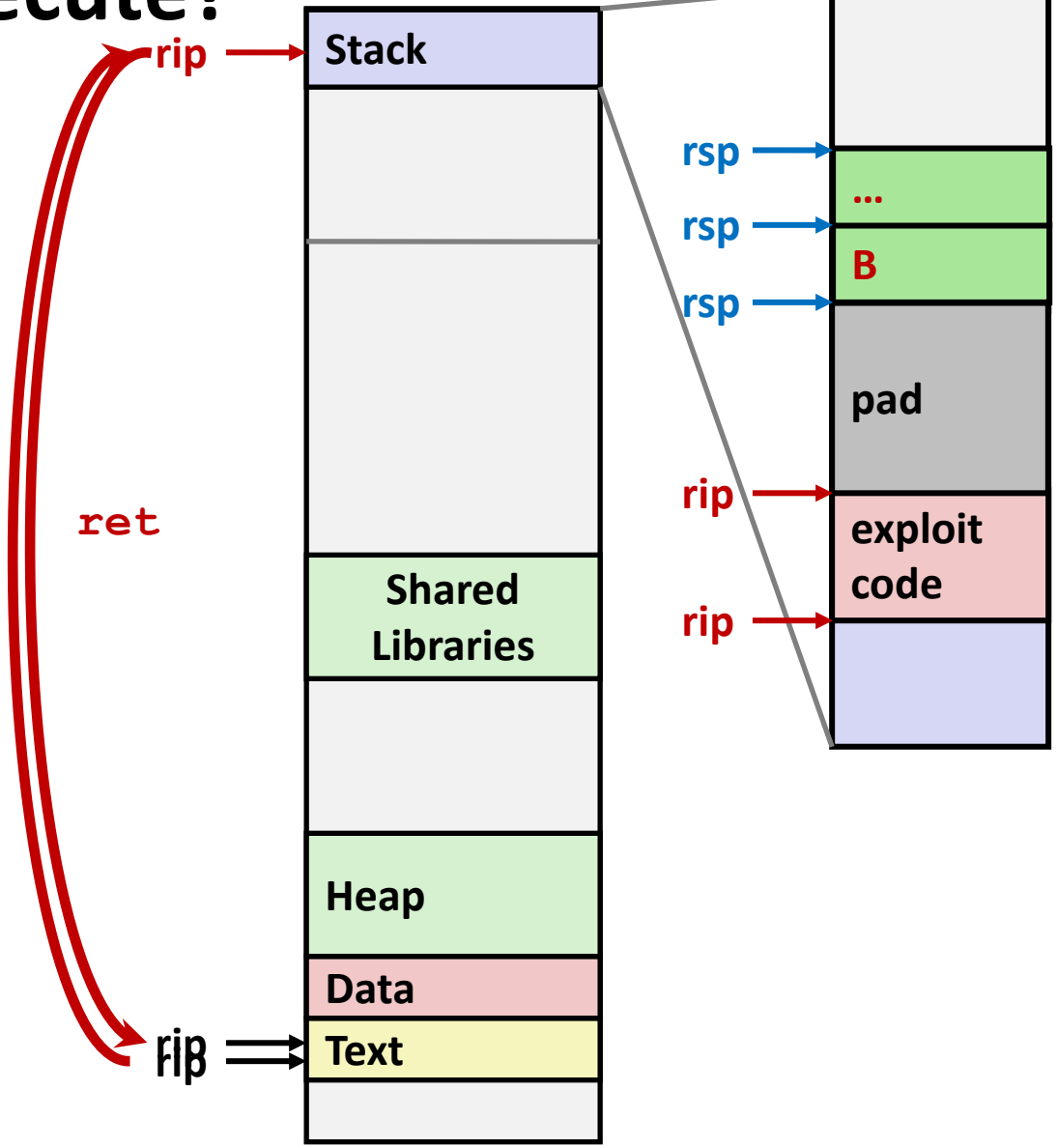


- **输入串包含执行代码的字节表示 Input string contains byte representation of executable code**
- **用缓冲区B的地址覆盖了返回地址A Overwrite return address A with address of buffer B**
- **当Q执行ret指令时，会跳转到攻击代码 When Q executes `ret`, will jump to exploit code**

攻击代码如何执行 How Does The Attack Code Execute?

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

```
int Q() {
  char buf[64];
  gets(buf); // A->B
  ...
  return ...;
}
```



如何防范缓冲区溢出攻击
OK, what to do about buffer overflow attacks

- 避免溢出漏洞 Avoid overflow vulnerabilities
- 采用系统级防护 Employ system-level protections
- 让编译器使用“栈金丝雀” Have compiler use “stack canaries”
- 让我们分别进行讨论 Lets talk about each…

1.在代码中避免溢出漏洞
1. Avoid Overflow Vulnerabilities in Code (!)

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

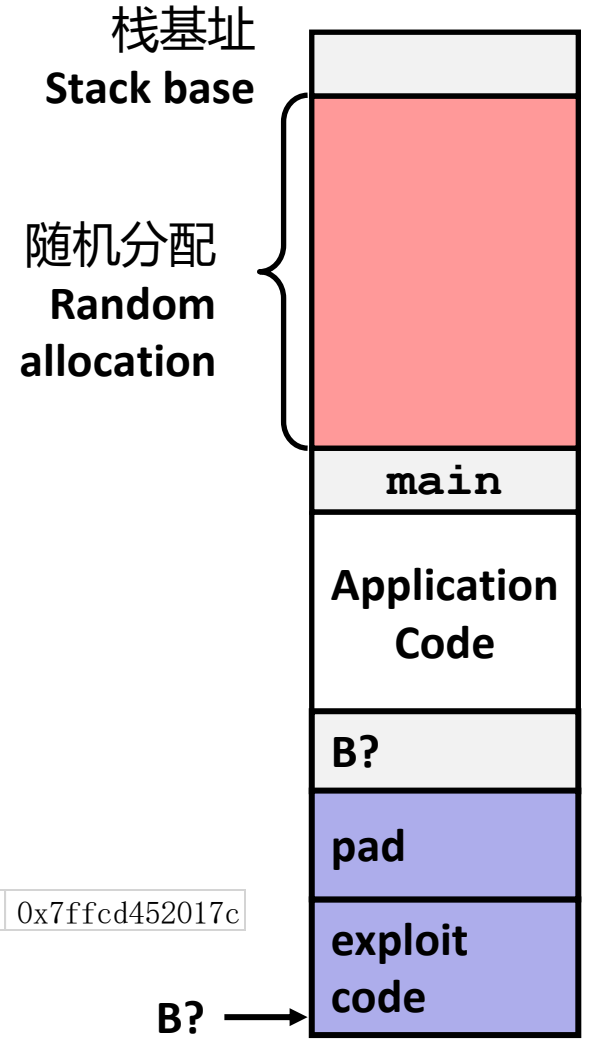
- **例如使用限制字符串的长度的库例程 For example, use library routines that limit string lengths**
  - **fgets** 代替 instead of **gets**
  - **strncpy** 代替 instead of **strcpy**
  - 不使用scanf的%s转换说明 Don't use **scanf** with **%s** conversion specification
    - 使用fgets读字符串 Use **fgets** to read the string
    - 或使用%ns，其中n是合适的整数 Or use **%ns** where **n** is a suitable integer

2.系统级防护会大有帮助
2. System-Level Protections can help

- **栈偏移随机化 Randomized stack offsets**
  - 程序的开始在栈上随机分配一定量空间 At start of program, allocate random amount of space on stack
  - 移动整个程序的栈地址 Shifts stack addresses for entire program
  - 使黑客很难预测插入代码的开始位置 Makes it difficult for hacker to predict beginning of inserted code
  - 例如5次执行内存分配代码E.g.: 5 executions of memory allocation code

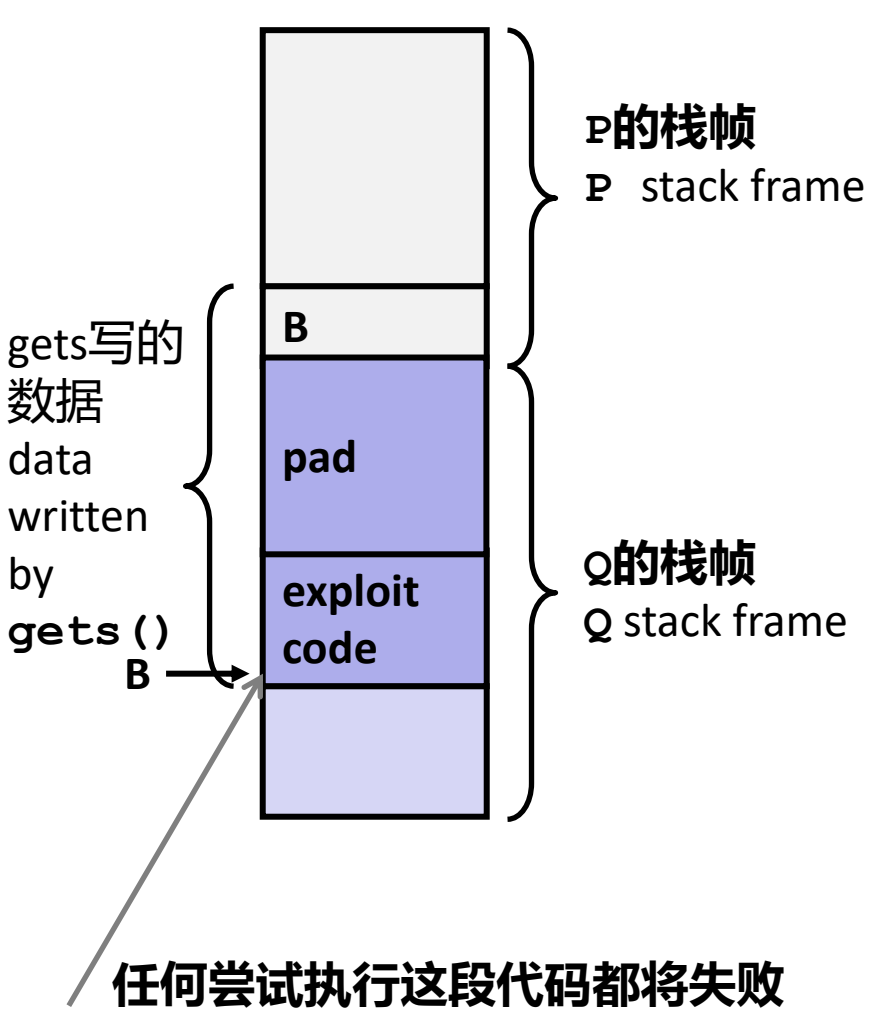
local	0x7ffe4d3be87c	0x7fff75a4f9fc	0x7ffeadb7c80c	0x7ffeaea2fdac	0x7ffcd452017c

  - 每次程序执行栈都重新排放 Stack repositioned each time program executes



2.系统级防护会大有帮助
2. System-Level Protections can help

- **非可执行代码段**
  Nonexecutable code segments
  - 老的x86CPU可以从任何可读地址执行机器代码 Older x86 CPUs would execute machine code from any readable address
  - x86-64增加了一种方式标记内存区域为“不可执行” x86-64 added a way to mark regions of memory as *not executable*
  - 在跳转进任何这类区域时立即崩溃 Immediate crash on jumping into any such region
  - 当前Linux和Windows标记栈为这种方式 Current Linux and Windows mark the stack this way



3.栈金丝雀会有帮助
3. Stack Canaries can help

- **思想 Idea**
  - 在栈中缓冲区以外放置特殊值（金丝雀） Place special value (“canary”) on stack just beyond buffer
  - 在退出函数之前检查金丝雀是否破坏 Check for corruption before exiting function
- **GCC实现 GCC Implementation**
  - `-fstack-protector`
  - 现在是默认的（早期是关闭的）Now the default (disabled earlier)

```
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
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
```

保护缓冲区反汇编
Protected Buffer Disassembly

echo:

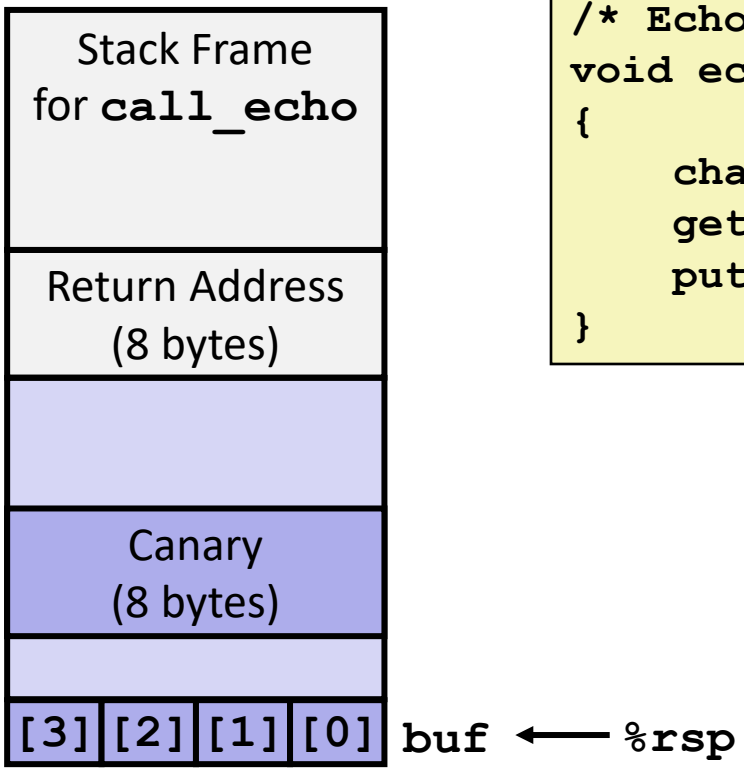
```
40072f:   sub    $0x18,%rsp
400733:   mov    %fs:0x28,%rax
40073c:   mov    %rax,0x8(%rsp)
400741:   xor    %eax,%eax
400743:   mov    %rsp,%rdi
400746:   callq  4006e0 <gets>
40074b:   mov    %rsp,%rdi
40074e:   callq  400570 <puts@p
400753:   mov    0x8(%rsp),%rax
400758:   xor    %fs:0x28,%rax
400761:   je     400768 <echo+0
400763:   callq  400580 <__stac
400768:   add    $0x18,%rsp
40076c:   retq
```

旁注：Aside: `%fs:0x28`

- **重用一项老功能（“分段地址”）即额外的指针寄存器**
  Reusing an old feature (“segmented addresses”) for an extra pointer register
- **看成是0x28(%fs) Think of this as 0x28(%fs)**
- **%fs在程序开始初始化，%fs:0x28容纳一个随机数 %fs set up at program start, %fs:0x28 holds a random number**

设置金丝雀 Setting Up Canary

调用gets之前 Before call to gets

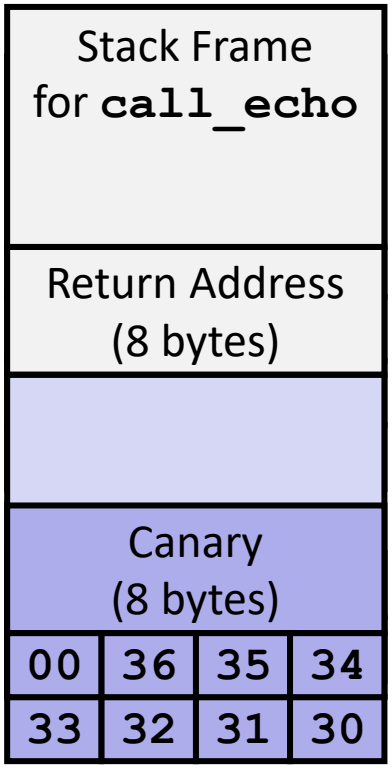



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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

检查金丝雀 Checking Canary

调用gets之后 After call to gets




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

Input: *0123456*

有些系统：
金丝雀的最低有效字节为0x00，允许输入01234567
Some systems:
LSB of canary is 0x00
Allows input 01234567

```
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:        . . .
```

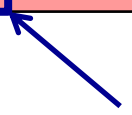
面向返回的编程攻击
Return-Oriented Programming Attacks

- **挑战（对于黑客） Challenge (for hackers)**
  - 随机化分配栈使预测缓冲区位置变得很难 Stack randomization makes it hard to predict buffer location
  - 标记栈为非可执行区域使插入二进制代码变得很难 Marking stack nonexecutable makes it hard to insert binary code
- **替代策略 Alternative Strategy**
  - 使用现存的代码 Use existing code
    - 程序或C语言库的一部分 Part of the program or the C library
  - 将代码片段串联在一起取得总体期望的效果 String together fragments to achieve overall desired outcome
  - *不能*战胜栈金丝雀 *Does not overcome stack canaries*
- **从小工具构造程序 Construct program from *gadgets***
  - 指令序列以ret结束 Sequence of instructions ending in **`ret`**
    - Ret的编码为单一字节0xc3 Encoded by single byte **`0xc3`**
  - 从运行指令到运行固定的指令位置 Code positions fixed from run to run
  - 代码是可执行的 Code is executable

小工具示例1 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
```



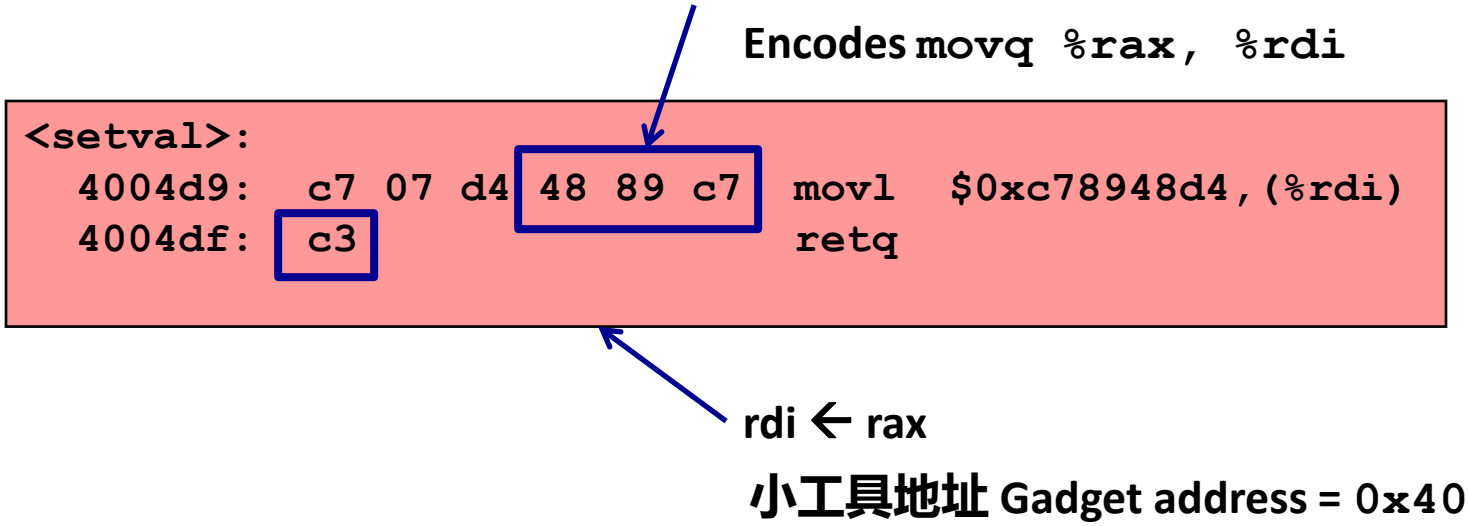
rax ← rdi + rdx

小工具地址 Gadget address = 0x4004d4

- 使用现存函数的尾部 Use tail end of existing functions

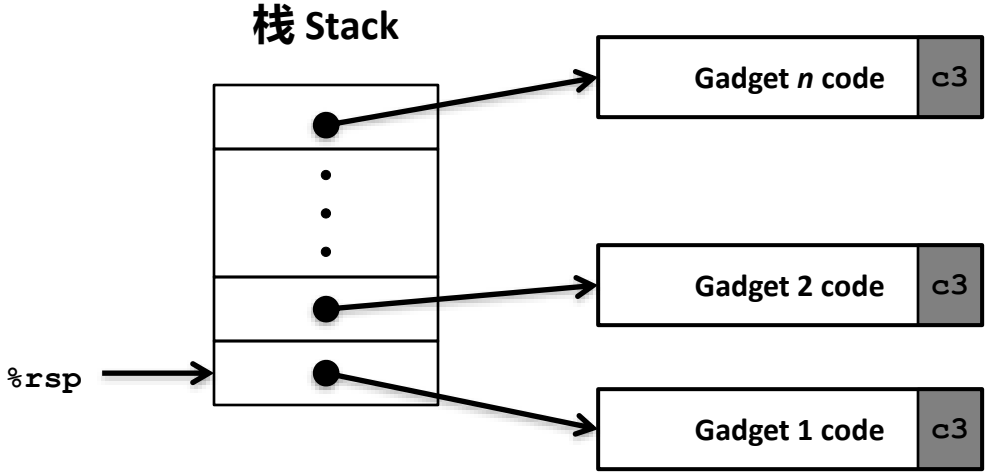
小工具示例#2 Gadget Example #2

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



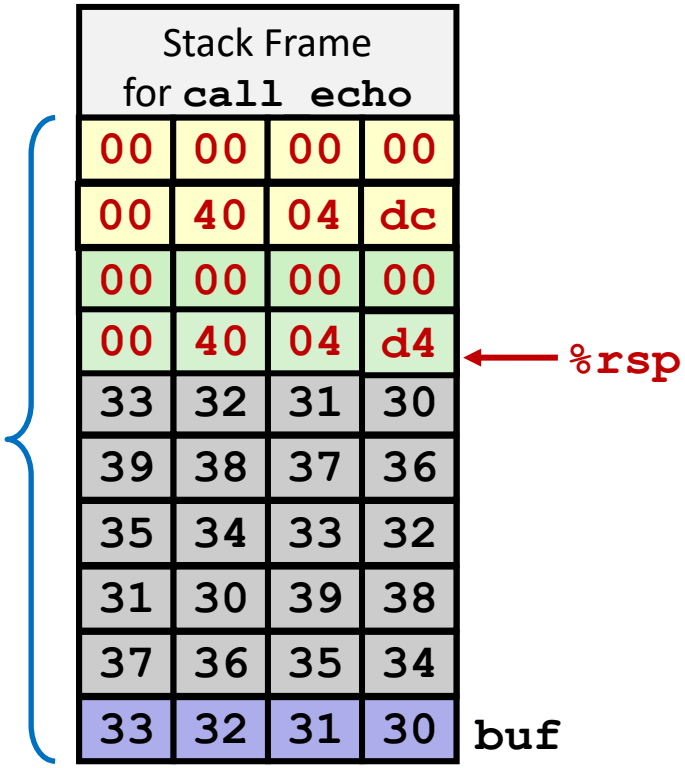
- 改变字节代码的目的 Repurpose byte codes

面向返回的编程ROP执行
ROP Execution



- **用ret指令触发 Trigger with `ret` instruction**
  - 将开始执行小工具1 Will start executing Gadget 1
- **每个小工具的最后ret指令将开始下一个小工具 Final `ret` in each gadget will start next one**
  - ret：从栈中弹出地址并跳转到该地址

制作ROP攻击字符串
Crafting an ROP Attack String



- **小工具#1 Gadget #1**
  - `0x4004d4` rax ← rdi + rdx
- **小工具#2 Gadget #2**
  - `0x4004dc` rdi ← rax
- **组合效果 Combination**

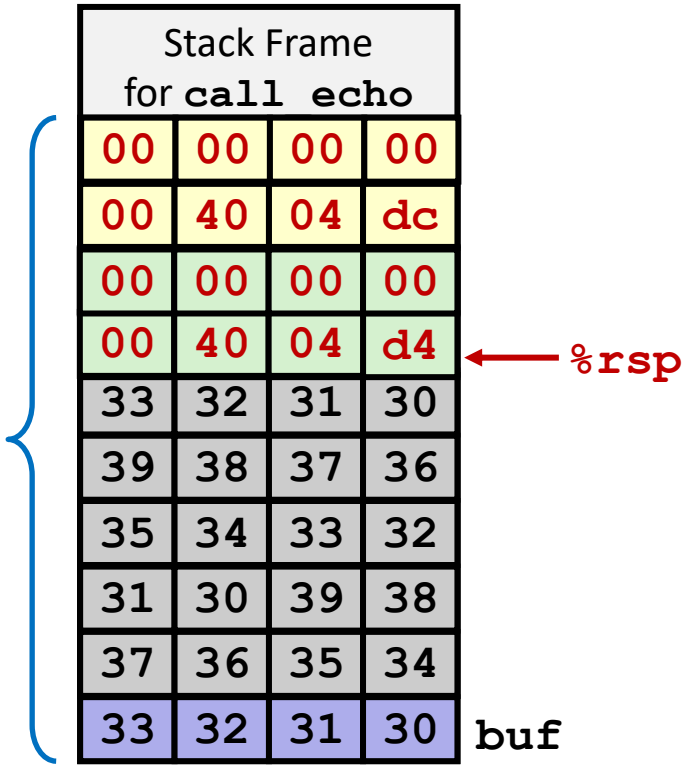
  rdi ← rdi + rdx

攻击字符串（十六进制） 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
```

多个小工具将向上破坏栈 Multiple gadgets will corrupt stack upwards

Echo返回时会发生什么情况?
What Happens When `echo` Returns?



1. Echo执行ret指令 Echo executes `ret`
   - 启动小工具#1 Starts Gadget #1
2. 小工具#1执行ret指令 Gadget #1 executes `ret`
   - 启动小工具#2 Starts Gadget #2
3. 小工具#2执行ret指令 Gadget #2 executes `ret`
   - 在某个地方结束。。。 Goes off somewhere ...

多个小工具将向上破坏栈 Multiple gadgets will corrupt stack upwards

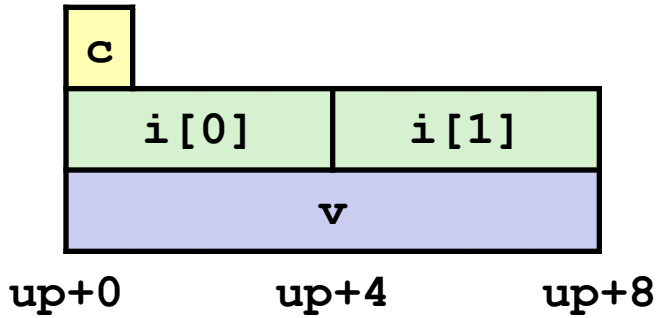
议题

- 内存布局 Memory Layout
- 缓冲区溢出 Buffer Overflow
  - 漏洞 Vulnerability
  - 防护 Protection
- **联合 Unions**

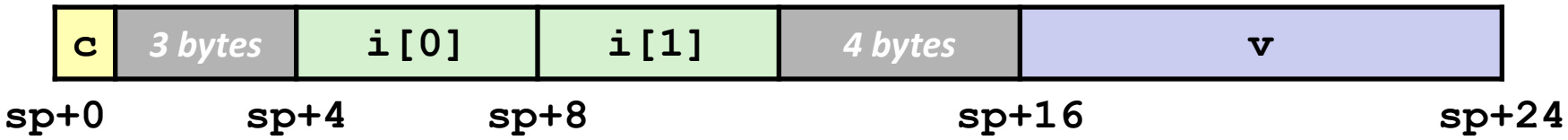
联合分配空间方式 Union Allocation

- 按照最大元素分配空间 Allocate according to largest element
- 一次仅能使用一个字段 Can only use one field at a time

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

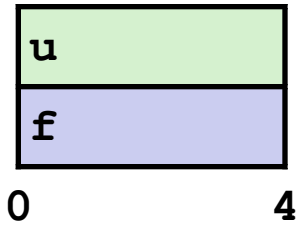



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



使用联合访问比特位模式
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;
}
```

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

与强制转换u成float型相同吗?

Same as `(float) u`?

与强制转换f成无符号数整数相同吗?

Same as `(unsigned) f`?

字节顺序修正 Byte Ordering Revisited

- **概念 Idea**
  - 不同字长的数据存储在内存的不同连续字节地址 Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - 哪个字节是最高（低）有效字节？ Which byte is most (least) significant?
  - 当在机器之间交换二进制数据时会引起问题 Can cause problems when exchanging binary data between machines
- **大端法 Big Endian**
  - 最高字节有最低地址 Most significant byte has lowest address
  - Sparc, *Internet*
- **小端法 Little Endian**
  - 最低字节有最低地址 Least significant byte has lowest address
  - Intel x86, ARM Android and IOS
- **双端法 Bi Endian**
  - 可以进行配置选择一种方法 Can be configured either way
  - ARM

字节顺序示例 Byte Ordering Example

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

在short/int/long中字节如何存储?

How are the bytes inside short/int/long stored?

内存地址增加 Memory addresses growing →

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

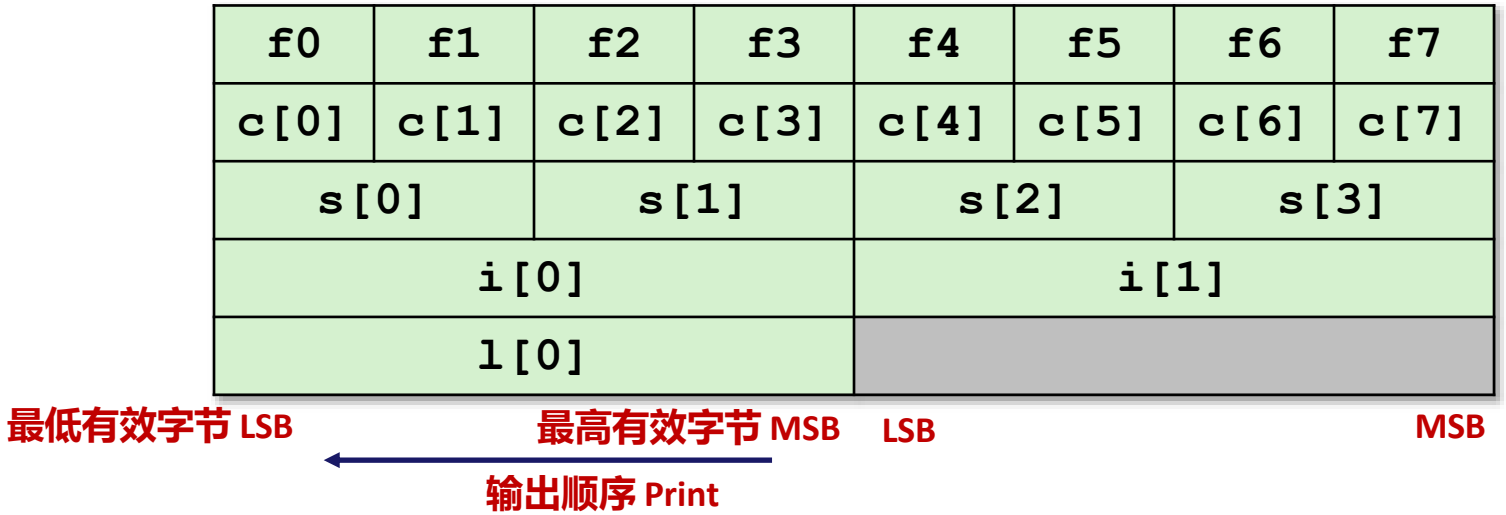
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

字节顺序示例（续）
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]);
```

IA32上字节顺序 Byte Ordering on IA32

小端法 Little Endian

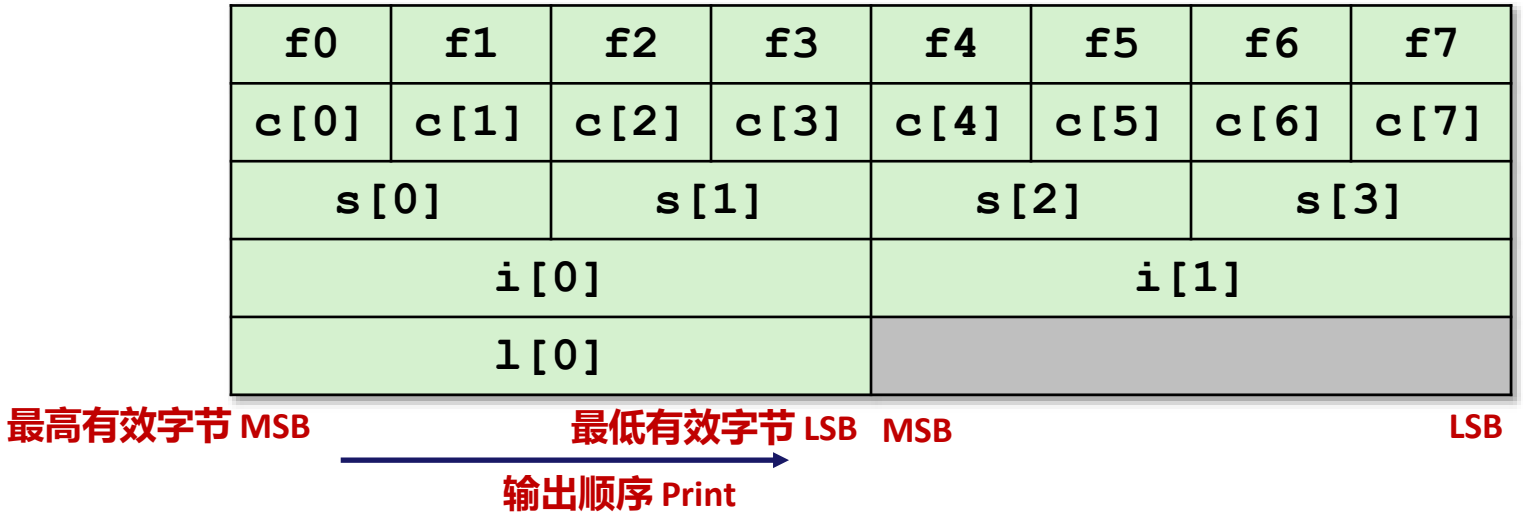


输出 Output:

```
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]
```

Sun机器上字节顺序 Byte Ordering on Sun

大端法 Big Endian

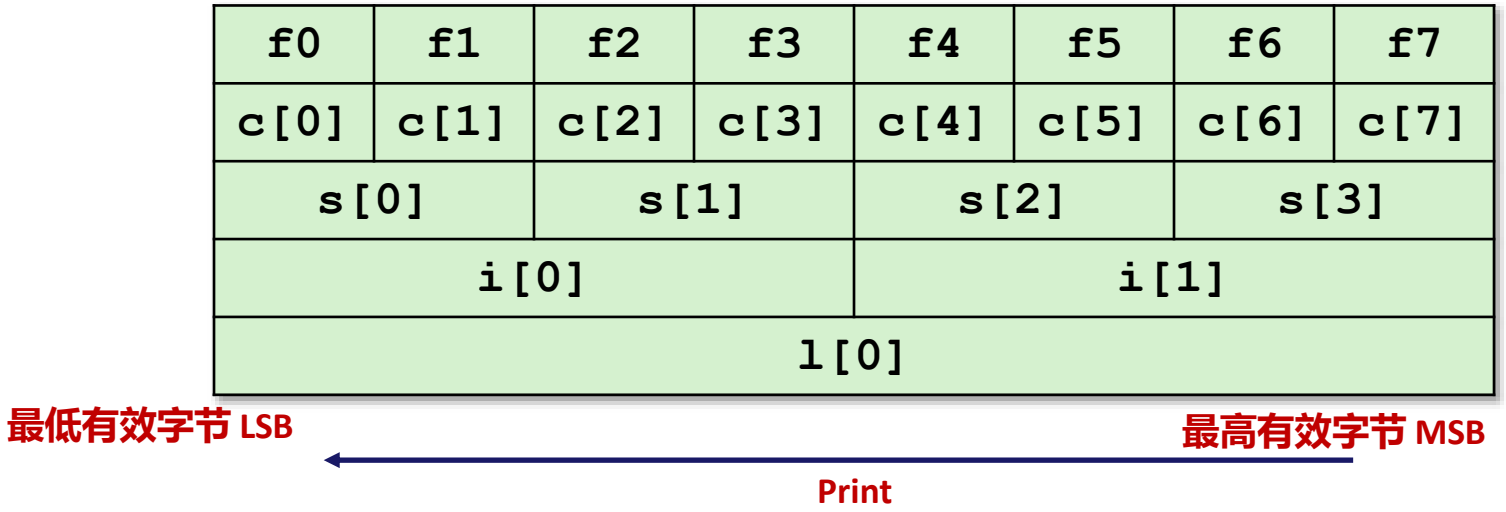


输出 Output on Sun:

```
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]
```

x86-64机器上字节顺序
Byte Ordering on x86-64

小端法 Little Endian



输出 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]
```

C语言中的组合类型小结
Summary of Compound Types in C

- **数组 Arrays**
  - 连续分配内存 Contiguous allocation of memory
  - 对齐到满足每个元素的对齐需求 Aligned to satisfy every element's alignment requirement
  - 指向第一个元素 Pointer to first element
  - 不进行边界检查 No bounds checking
- **结构 Structures**
  - 按照声明的顺序分配字节 Allocate bytes in order declared
  - 在中间进行填充和在结尾满足对齐需求 Pad in middle and at end to satisfy alignment
- **联合 Unions**
  - 覆盖声明 Overlay declarations
  - 回避类型系统的方法 Way to circumvent type system

小结 Summary

- **内存布局 Memory Layout**
- **缓冲区溢出 Buffer Overflow**
  - 漏洞 Vulnerability
  - 防护 Protection
  - 代码注入攻击 Code Injection Attack
  - 面向返回的编程 Return Oriented Programming
- **联合 Unions**