# 计算机系统基础

## 程序的机器级表示(6)

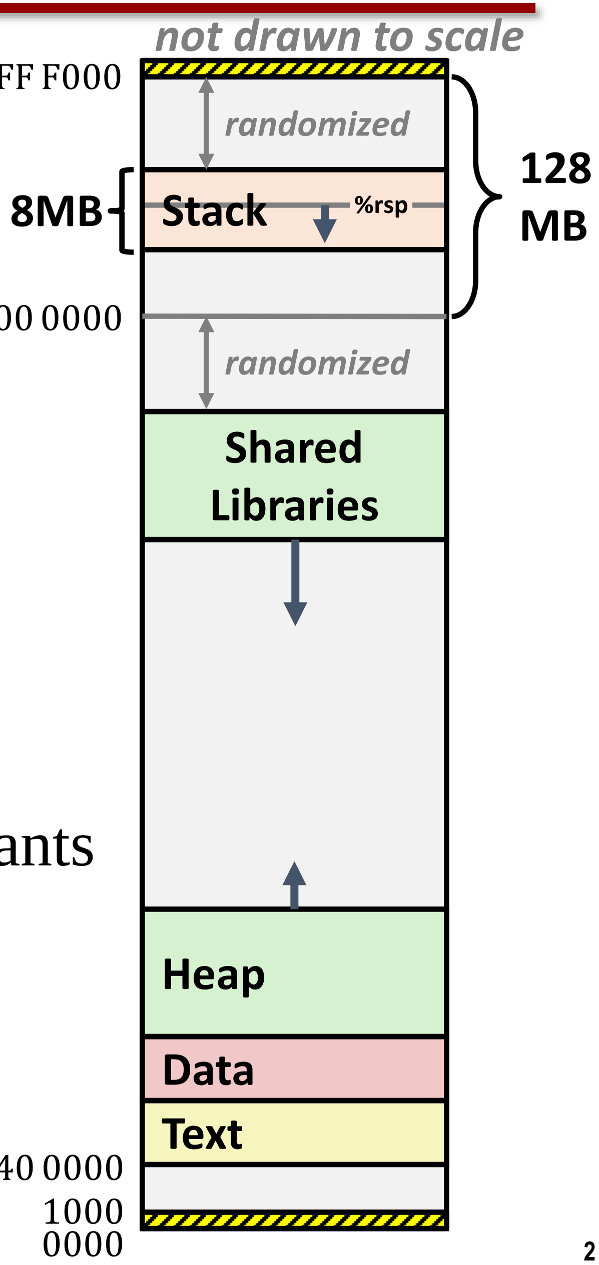
王晶

jwang@ruc.edu.cn，信息楼124

2024年11月

# x86-64 Linux 存储空间

- 栈
  - 运行时栈 (8MB limit)
  - e.g., 局部变量
- 堆
  - 按需动态分配
  - `malloc(), calloc(), new()`
- 数据
  - 静态分配的数据
  - E.g., global vars, `static` vars, string constants
- 代码 / 共享库
  - 可执行的机器代码
  - 只读



# 内存分配的例子

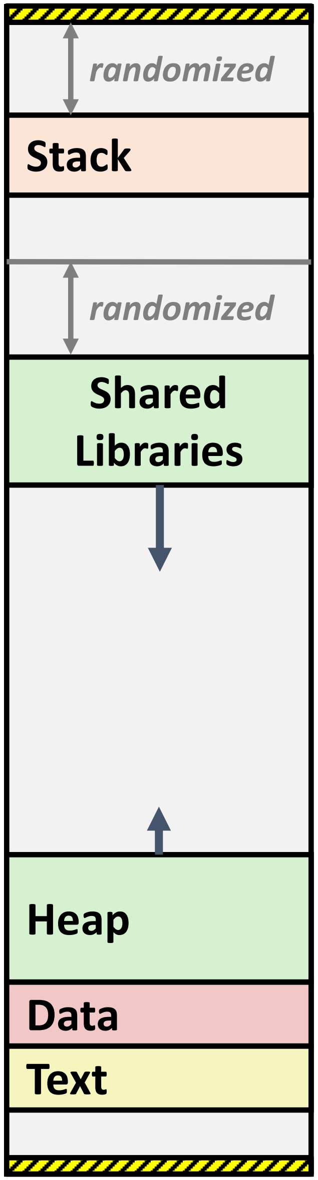
```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *phuge1, *psmall2, *phuge3, *psmall4;
    int local = 0;
    phuge1 = malloc(1L << 28);  /* 256 MB */
    psmall2 = malloc(1L << 8);  /* 256  B */
    phuge3 = malloc(1L << 32);  /*   4 GB */
    psmall4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```

0000 7FFF FFFF F000



40 0000

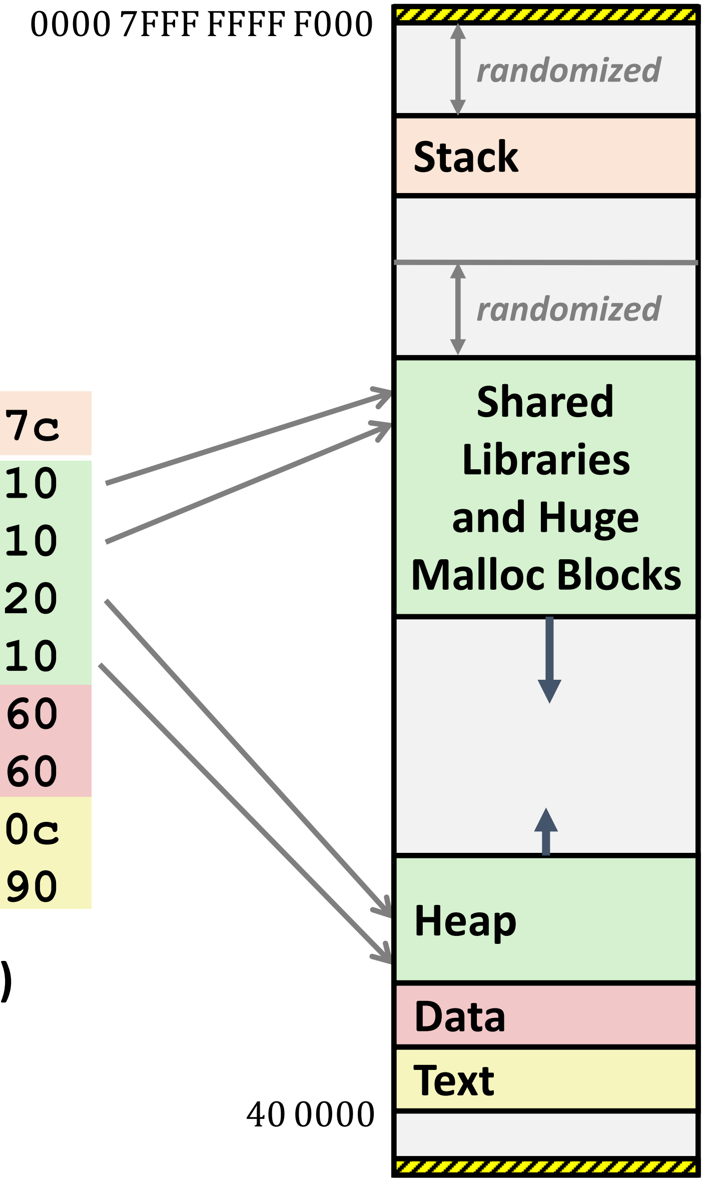
*Where does everything go?*

# x86-64 下的地址

*address range ~$2^{47}$*

| | |
|---|---|
| `local` | `0x00007ffe4d3be87c` |
| `phuge1` | `0x00007f7262a1e010` |
| `phuge3` | `0x00007f7162a1d010` |
| `psmall4` | `0x000000008359d120` |
| `psmall2` | `0x000000008359d010` |
| `big_array` | `0x0000000080601060` |
| `huge_array` | `0x0000000000601060` |
| `main()` | `0x000000000040060c` |
| `useless()` | `0x0000000000400590` |

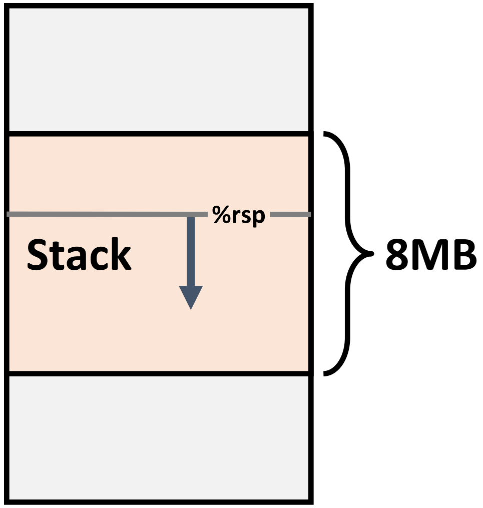
**(Exact values can vary)**



# 栈溢出的例子

*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;
}
```



- 函数将本地数据存储在栈帧中
- 递归函数导致栈帧的深度嵌套
- 当空间用完时会发生什么？

```
./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)
```

# 回顾内存访问的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)   ->    3.1400000000
fun(1)   ->    3.1400000000
fun(2)   ->    3.1399998665
fun(3)   ->    2.0000006104
fun(6)   ->    Stack smashing detected
fun(8)   ->    Segmentation fault
```

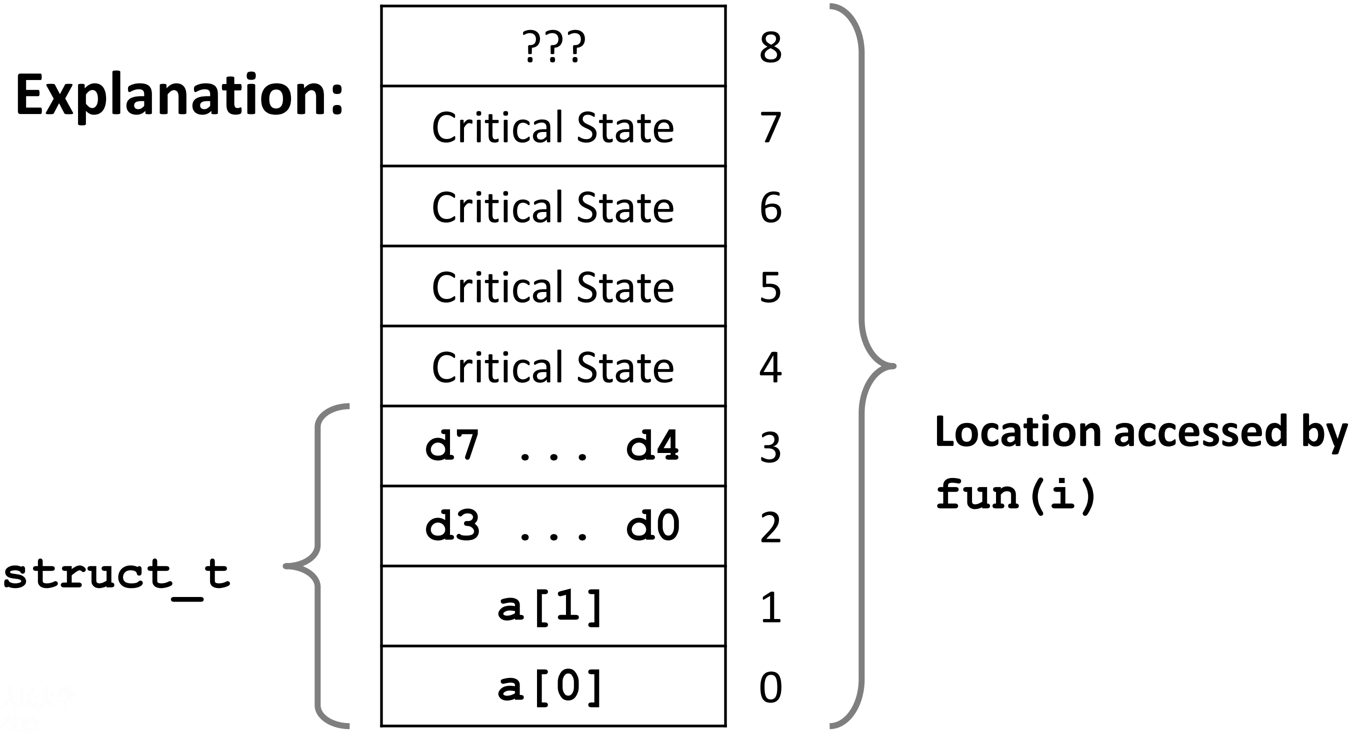
- Result is system specific

# 内存访问的bug

```
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
fun(8)   ->    3.1400000000
```

**Explanation:**



# 这是一个大问题

- 通常称为“缓冲区溢出”
  - 当超过为数组分配的内存大小时
- 为什么是大事？
  - 这是安全漏洞的 #1 技术原因
- 最常见的形式
  - 字符串输入上未检查的长度
  - 特别是对于栈上的有界字符数组

# 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;
}
```

  - 无法指定要读取的字符数限制
- 其他库函数也有类似问题
  - **`strcpy`**, **`strcat`**: Copy strings of arbitrary length
  - **`scanf`**, **`fscanf`**, **`sscanf`**, when given **`%s`** conversion specificatio

# 易受攻击的缓冲区代码

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

← btw, how big
is big enough?

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

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

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

# 缓冲区溢出的反汇编

## echo:

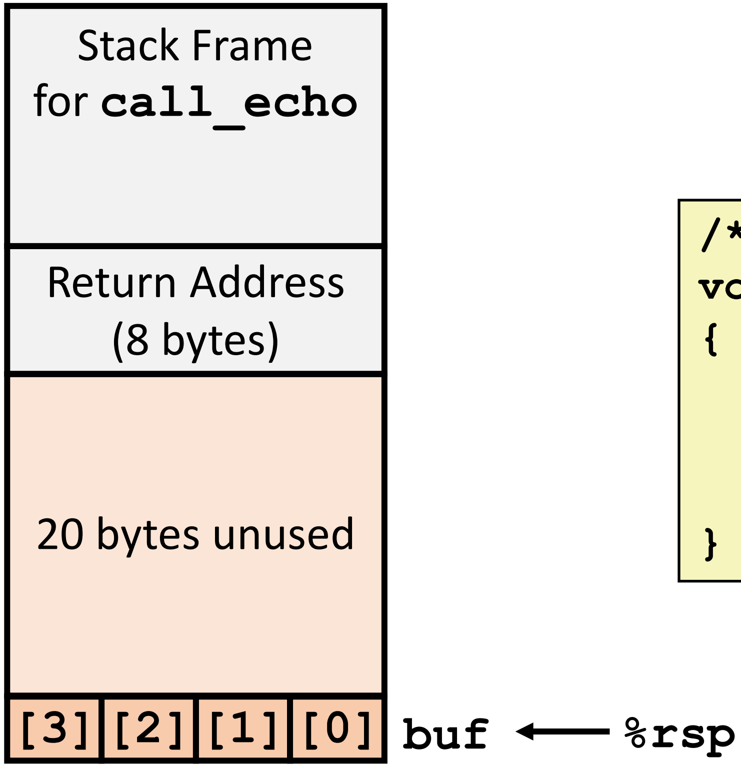
```
0000000004006cf <echo>:
 4006cf:   48 83 ec 18                sub    $0x18,%rsp
 4006d3:   48 89 e7                   mov    %rsp,%rdi
 4006d6:   e8 a5 ff ff ff             callq  400680 <gets>
 4006db:   48 89 e7                   mov    %rsp,%rdi
 4006de:   e8 3d fe ff ff             callq  400520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

*Before call to gets*

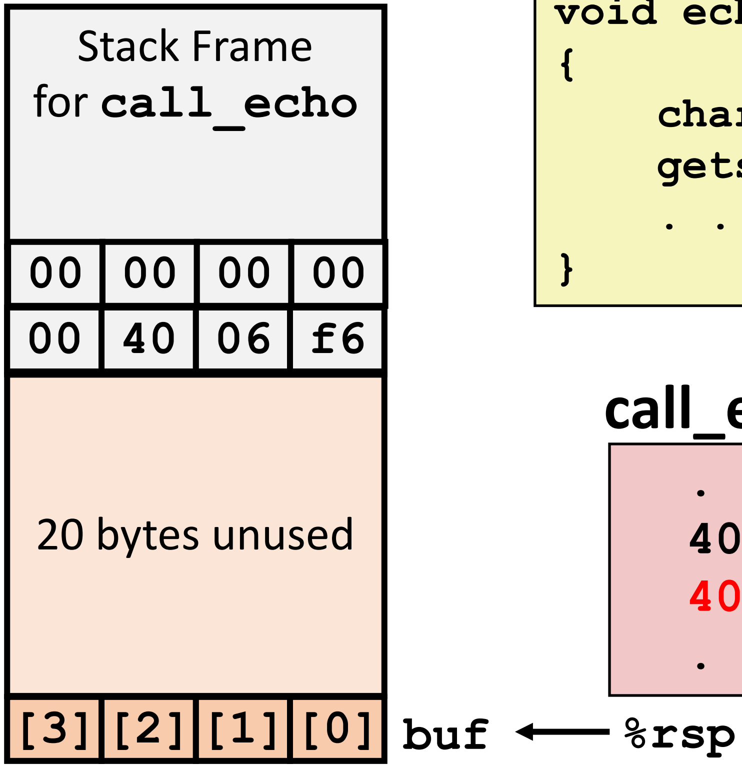


```
/* Echo Line */
void echo()
{
    char buf[4];   /* Way too small! */
    gets(buf);
    puts(buf);
}
```

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

# 缓冲区溢出的栈

*Before call to gets*



Stack Frame for `call_echo`

| | | | |
|---|---|---|---|
| 00 | 00 | 00 | 00 |
| 00 | 40 | 06 | f6 |

20 bytes unused

| | | | |
|---|---|---|---|
| [3] | [2] | [1] | [0] |

buf ← %rsp

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

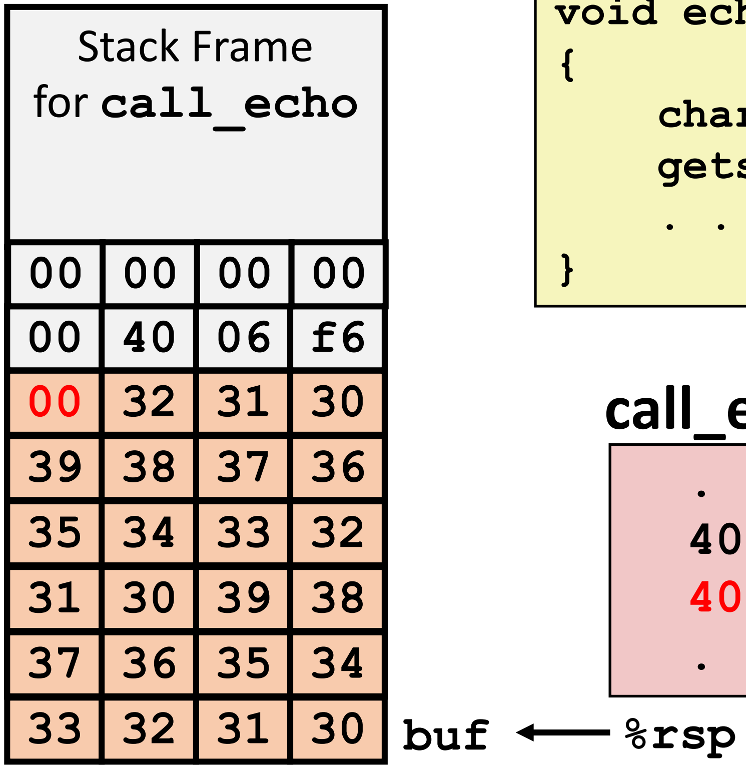
```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

**call_echo:**

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

# 缓冲区溢出的栈

*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 |

```
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
  . . .
```

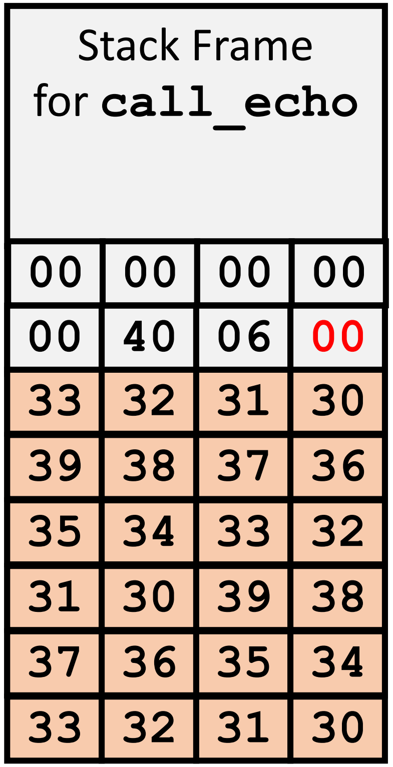
```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

```
"012345678901234567890123\0"
```

**Overflowed buffer, but did not corrupt state**

# 缓冲区溢出的栈

*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
Segmentation fault
```

**Program "returned" to 0x0400600, and then crashed.**

# Stack Smashing攻击

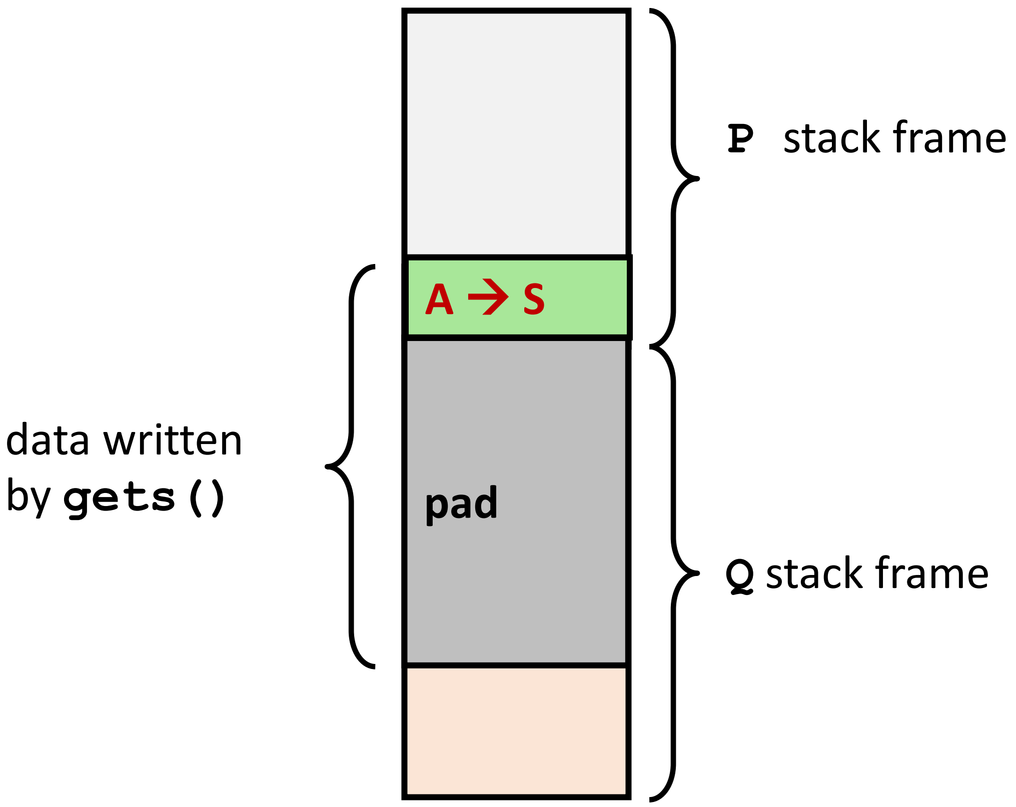
```
void P(){
   Q();
   ...
}
```

return address A

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

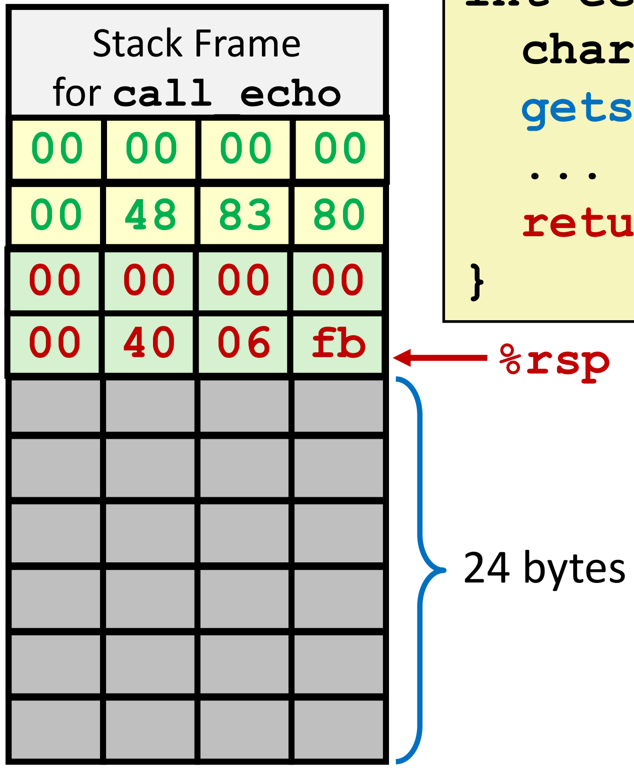
```
void S(){
/* Something
   unexpected */
   ...
}
```

Stack after call to gets()



- Overwrite normal return address A with address of some other code S
- When Q executes ret, will jump to other code

# Crafting Smashing String



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

***Target Code***

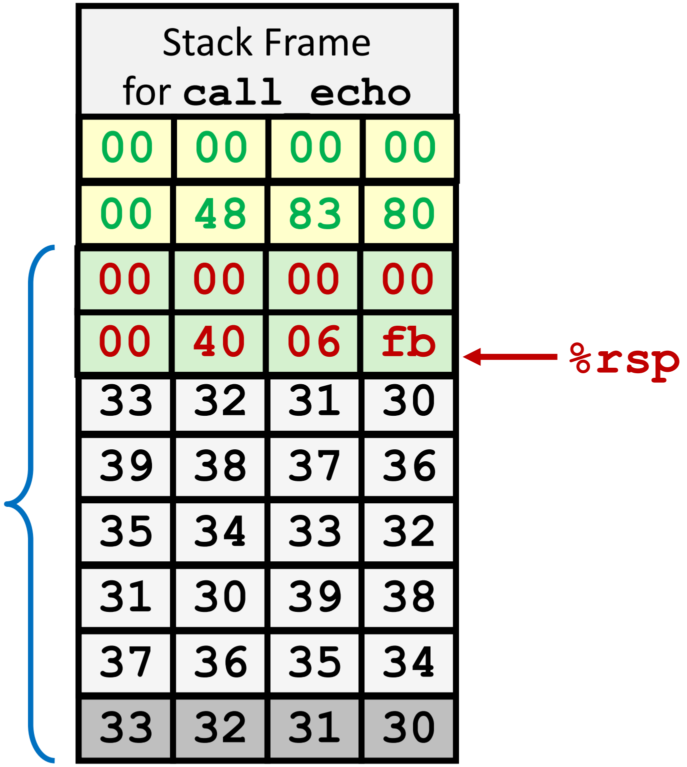
```
void smash() {
  printf("I've been smashed!\n");
  exit(0);
}
```

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

***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
fb 06 40 00 00 00 00 00
```

# Smashing String Effect



Stack Frame for call_echo

| | | | |
|---|---|---|---|
| 00 | 00 | 00 | 00 |
| 00 | 48 | 83 | 80 |
| 00 | 00 | 00 | 00 |
| 00 | 40 | 06 | fb |
| 33 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

*Target Code*

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

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

*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
fb 06 40 00 00 00 00 00
```

# 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!
```

- 将十六进制序列放入文件smash-hex.txt
- 使用 hexify 程序将十六进制数字转换为字符
- 其中一些是非打印的
- 作为易受攻击计划的输入

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

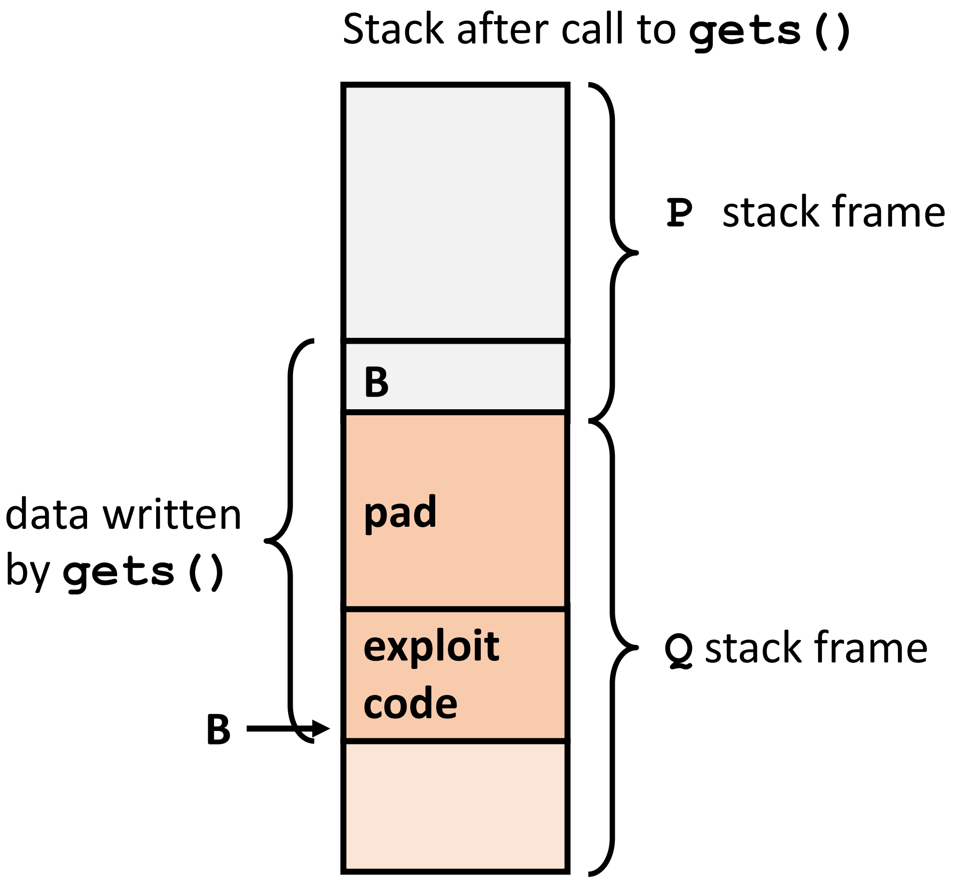
```
30 31 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36 37 38 39 30 31 32 33
c8 06 40 00 00 00 00 00
```

# Code Injection Attacks

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

return address A

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



Stack after call to **gets()**

**P** stack frame

**Q** stack frame

data written by **gets()**

**B**

**pad**

**exploit code**

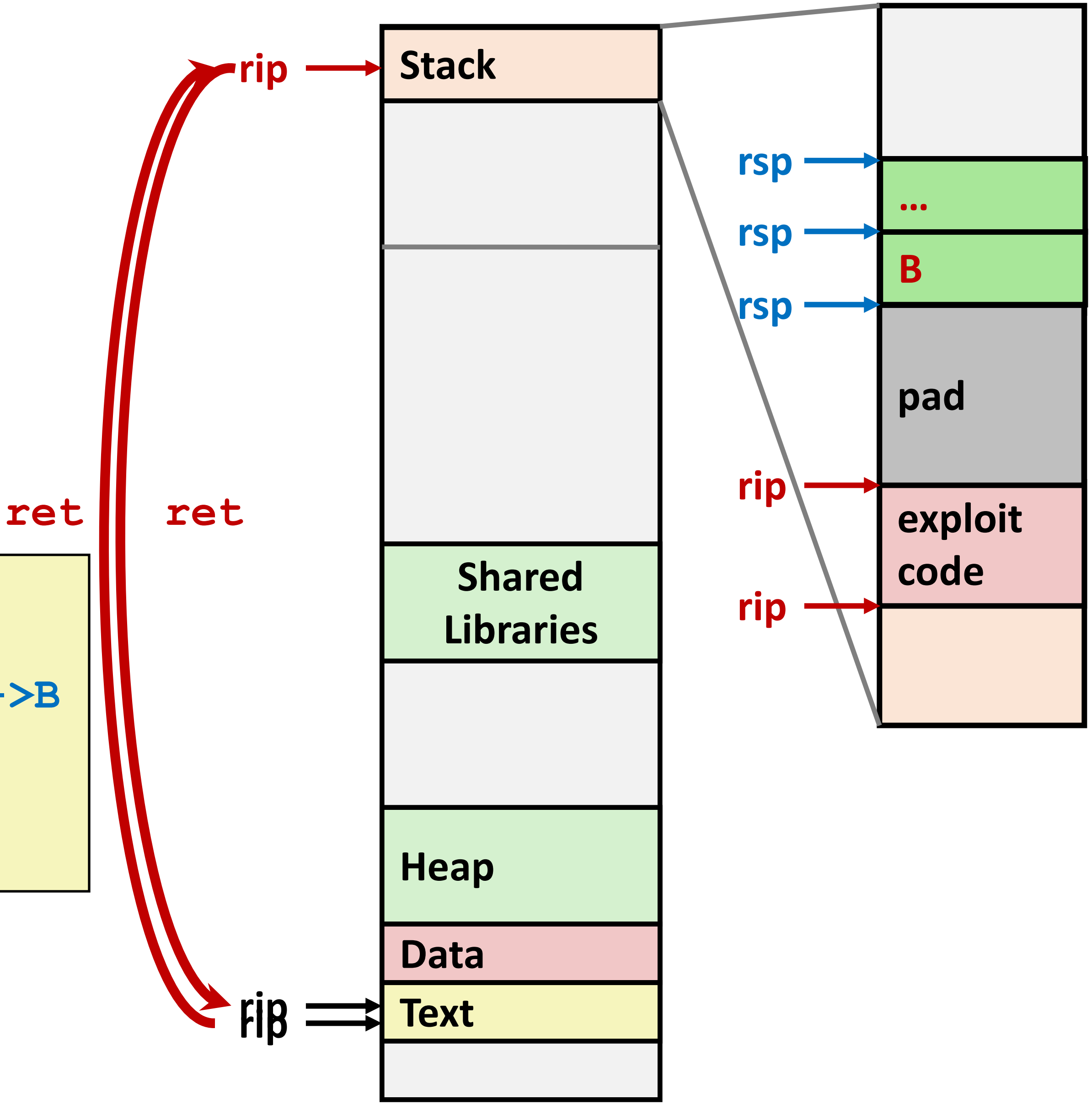
**B**

- 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

# 攻击执行流程

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

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



# 基于缓冲区溢出的漏洞

- *缓冲区溢出错误可能允许远程计算机在受攻击的计算机上执行任意代码*
- 在实际程序中非常常见
  - 程序员不断犯同样的错误☹
  - 最近的措施使这些攻击变得更加困难
- 几十年来的例子
  - “Internet worm” (1988)
  - “IM wars” (1999)
  - Twilight hack on Wii (2000s)
  - … and many, many more
- 您将在 attacklab 中学习一些技巧
  - 希望能说服你永远不要在你的程序中留下这样的漏洞！！

# Internet worm (1988)

- 利用一些漏洞进行传播
  - 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.
- 一旦进入计算机，扫描其他计算机以进行攻击
  - 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…

# Morris Worm

- Robert Tappan Morris
  - born November 8, 1965

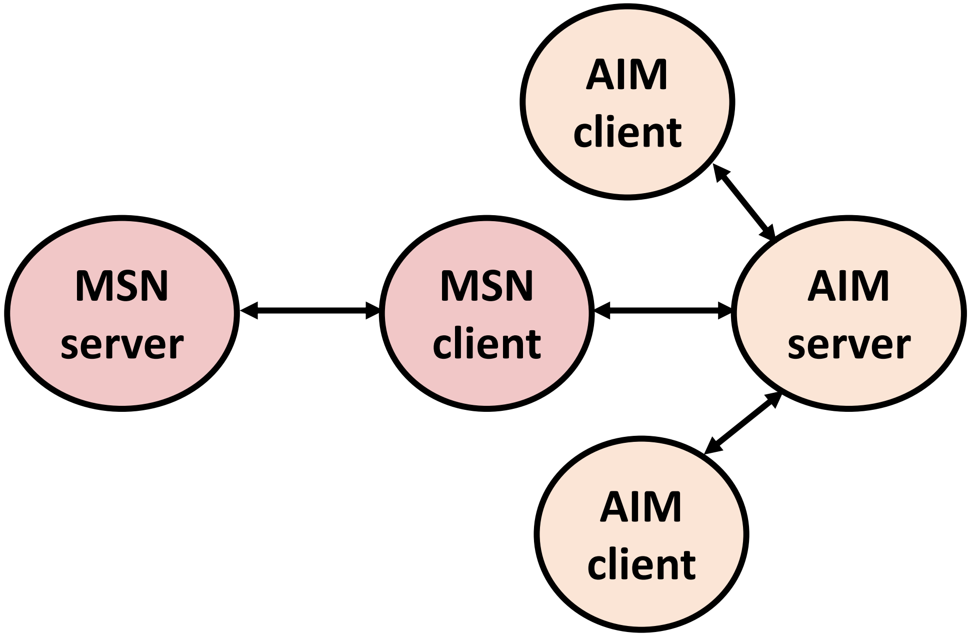
**父亲罗伯特·莫里斯为贝尔实验室计算机安全专家**
**他16岁上初中时，发现UNIX系统漏洞**
**1983年考入哈佛大学，拿到了文学学士（A.B.）学位**
**1988年成为康奈尔大学研究生**
**1988年散布了“莫里斯蠕虫”，造成约6000个系统瘫痪，给这些用户总共带来约200万到6000万美元的损失。**
**现为麻省理工学院教授 (2006 tenure, 2015 ACM Fellow)**

**MIT 6.824 Distributed Systems**

**https://www.bilibili.com/video/BV1qk4y197bB?from=search&seid=7720468517381990376**

# Example 2: IM War

- 1999 年 7 月
  - Microsoft 推出 MSN Messenger（即时通讯系统）。
  - Messenger 客户端可以访问流行的 AOL Instant Messaging Service （AIM） 服务器



# IM War (cont.)

- 1999 年 8 月
  - Messenger 客户端无法再访问 AIM 服务器
  - Microsoft 和 AOL 开始了 IM 战争：
    - AOL 更改服务器以禁止 Messenger 客户端
    - Microsoft 对客户端进行更改以阻止 AOL 更改
    - 至少 13 次此类小规模冲突
  - 到底发生了什么？
    - AOL 在他们自己的 AIM 客户端中发现了一个缓冲区溢出错误
    - 他们利用它来检测和阻止 Microsoft：漏洞利用代码将 4 字节的签名（AIM 客户端中某个位置的字节）返回给服务器
    - 当 Microsoft 更改代码以匹配签名时，AOL 更改了签名位置

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

- 蠕虫：一个程序
  - 可以自行运行
  - 可以将自身的完整工作版本传播到其他计算机

- 病毒：代码
  - 将自身添加到其他程序
  - 不独立运行

- 两者都（通常）被设计为在计算机之间传播并造成严重破坏

# 如何应对缓冲区溢出攻击

- 避免溢出漏洞
- 采用系统级保护
- 让编译器使用 “stack canary”
- 让我们谈谈每一个……

# 代码中避免溢出漏洞

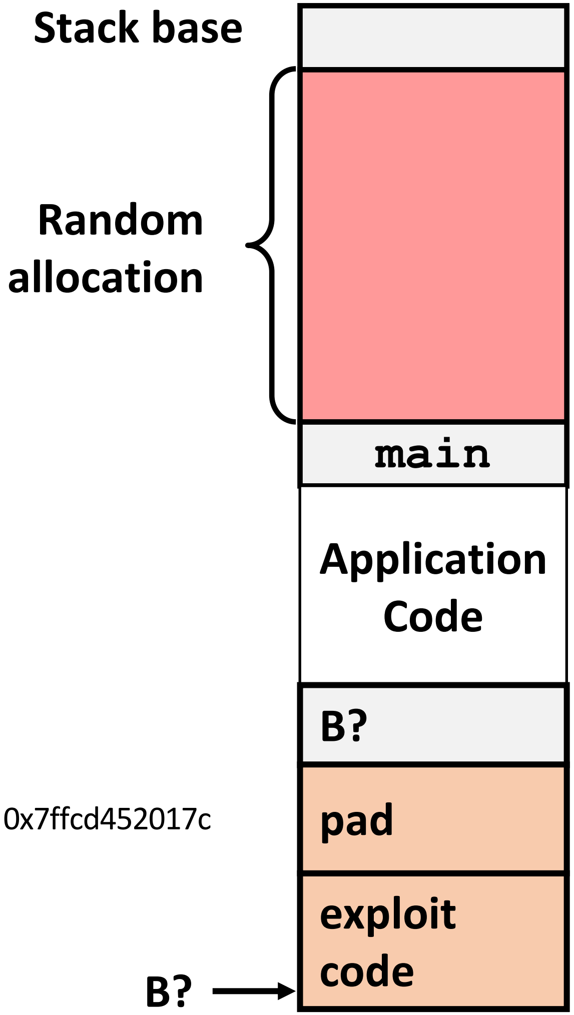
```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- 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

# 系统级保护

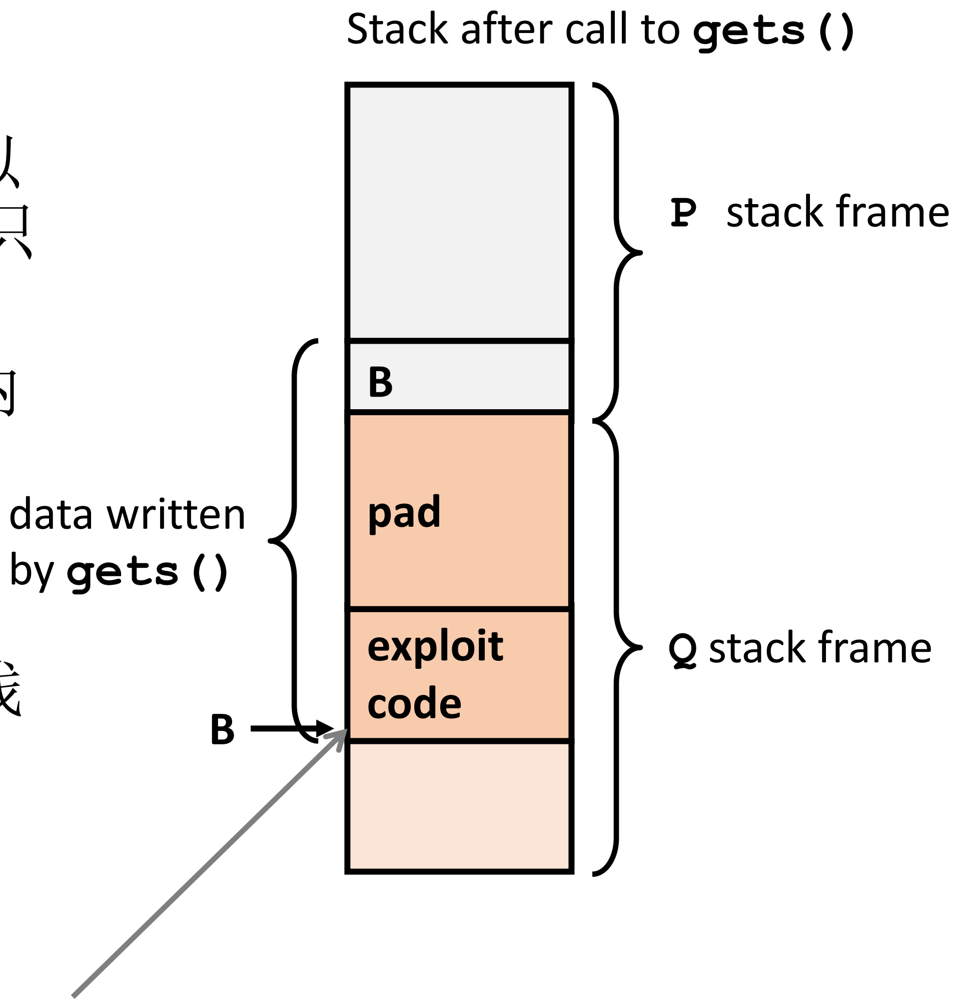
- 随机栈偏移量
  - 在程序开始时，在栈上分配随机数量的空间
  - 转移整个程序的栈地址
  - 使黑客难以预测插入代码的开头位置
  - 例如：内存分配代码的 5 次执行
  - 每次程序执行时重新定位堆栈

```
local        0x7ffe4d3be87c    0x7fff75a4f9fc    0x7ffeadb7c80c    0x7ffeaea2fdac    0x7ffcd452017c
```



# 系统级保护

- 不可执行的代码段
  - 在传统的 x86 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内容
  - X86-64 添加了显式的“execute”权限
  - 标记为不可执行的堆栈



**Any attempt to execute this code will fail**

# 面向返回的编程攻击

- 挑战（针对黑客）
  - 栈随机化使得很难预测缓冲区位置
  - 将栈标记为不可执行会使插入二进制代码变得困难
- 替代策略
  - 使用现有代码
    - 例如，来自 stdlib 的库代码
  - 将片段串在一起以实现总体预期结果
  - 不克服堆栈金丝雀
- 从 gadgets 构建程序
  - 以 ret 结尾的指令序列
  - 由单字节 0xc3 编码
  - 代码位置在每次运行之间是固定的
  - 代码是可执行的

# 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
```

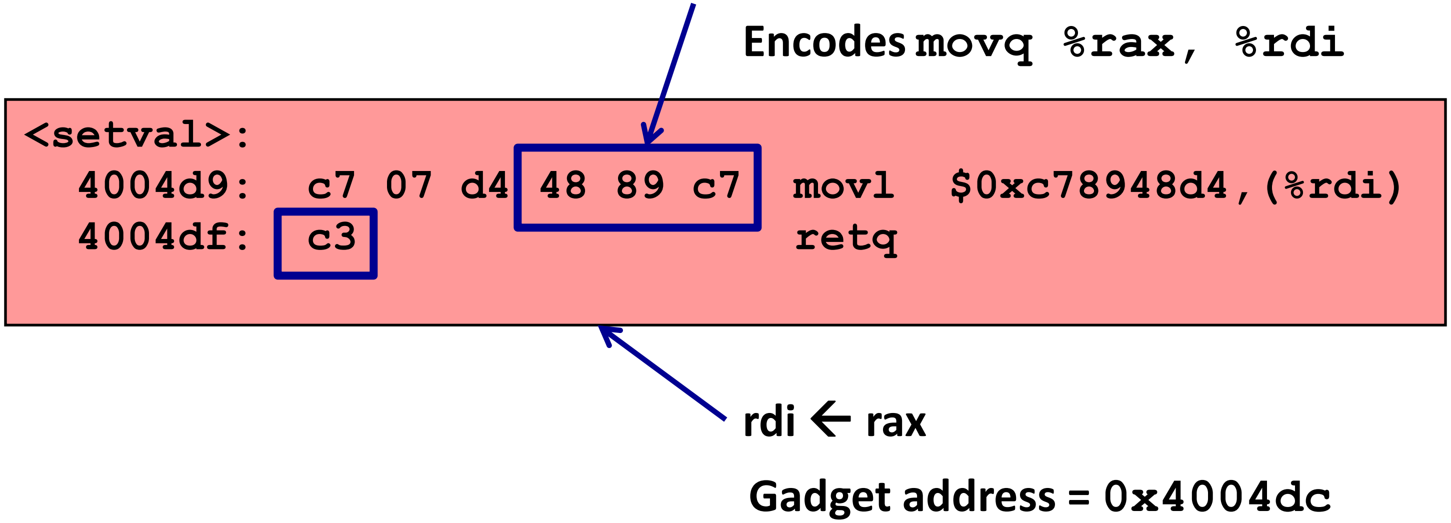
rax ← rdi + rdx

Gadget address = `0x4004d4`

- Use tail end of existing functions

# Gadget Example #2

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



Encodes `movq %rax, %rdi`

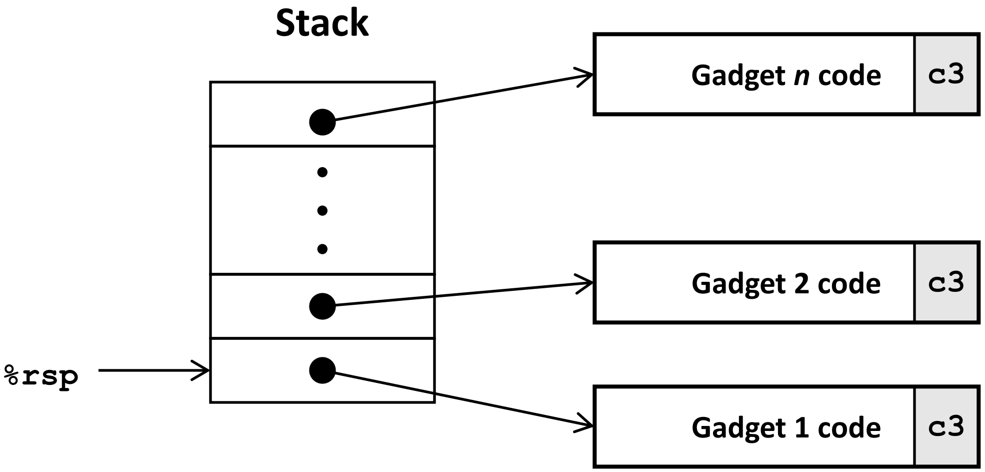
```
<setval>:
  4004d9:  c7 07 d4 48 89 c7  movl   $0xc78948d4,(%rdi)
  4004df:  c3                 retq
```

rdi ← rax

Gadget address = `0x4004dc`

- Repurpose byte codes

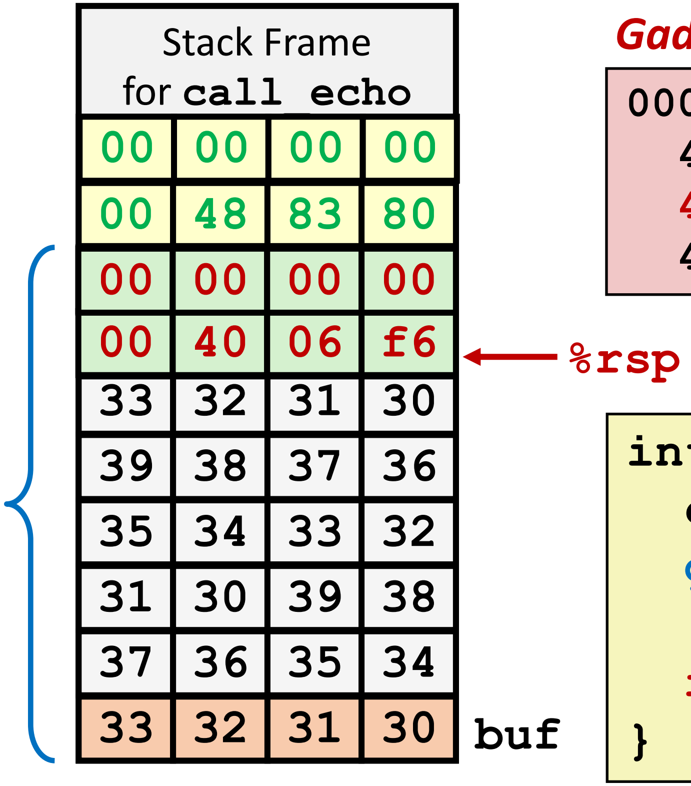
# ROP Execution



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

Shacham, H. (October 2007). "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)". *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*. pp. 552–561. ISBN 978-1-59593-703-2. doi:10.1145/1315245.1315313

# Crafting an ROP Attack String



Stack Frame for call_echo

| | | | |
|---|---|---|---|
| 00 | 00 | 00 | 00 |
| 00 | 48 | 83 | 80 |
| 00 | 00 | 00 | 00 |
| 00 | 40 | 06 | f6 |
| 33 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

***Gadget***

rax ← rdi + rdx

```
0000000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```

Attack: `int echo()` returns rdi + rdx

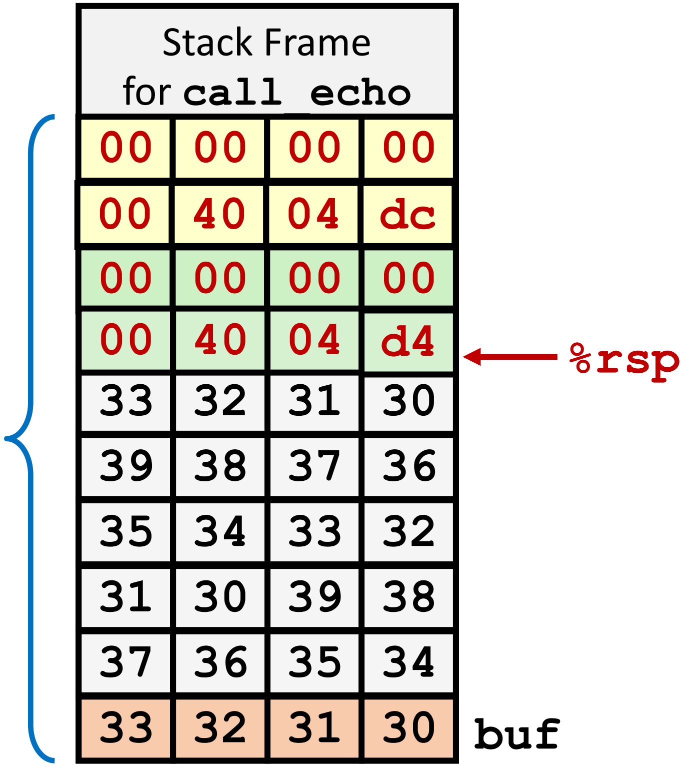
```
int echo() {
  char buf[4];
  gets(buf);
  ...
  return ...;
}
```

***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
```

Multiple gadgets will corrupt stack upwards

# Crafting an ROP Attack String



| Stack Frame for call_echo | | | |
|---|---|---|---|
| 00 | 00 | 00 | 00 |
| 00 | 40 | 04 | dc |
| 00 | 00 | 00 | 00 |
| 00 | 40 | 04 | d4 |
| 33 | 32 | 31 | 30 |
| 39 | 38 | 37 | 36 |
| 35 | 34 | 33 | 32 |
| 31 | 30 | 39 | 38 |
| 37 | 36 | 35 | 34 |
| 33 | 32 | 31 | 30 |

- Gadget #1
  - `0x4004d4` rax ← rdi + rdx
- Gadget #2
  - `0x4004dc` rdi ← rax
- Combination

  rdi ← rdi + rdx

***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
```

Multiple gadgets will corrupt stack upwards

# 3. Stack Canaries

- 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:

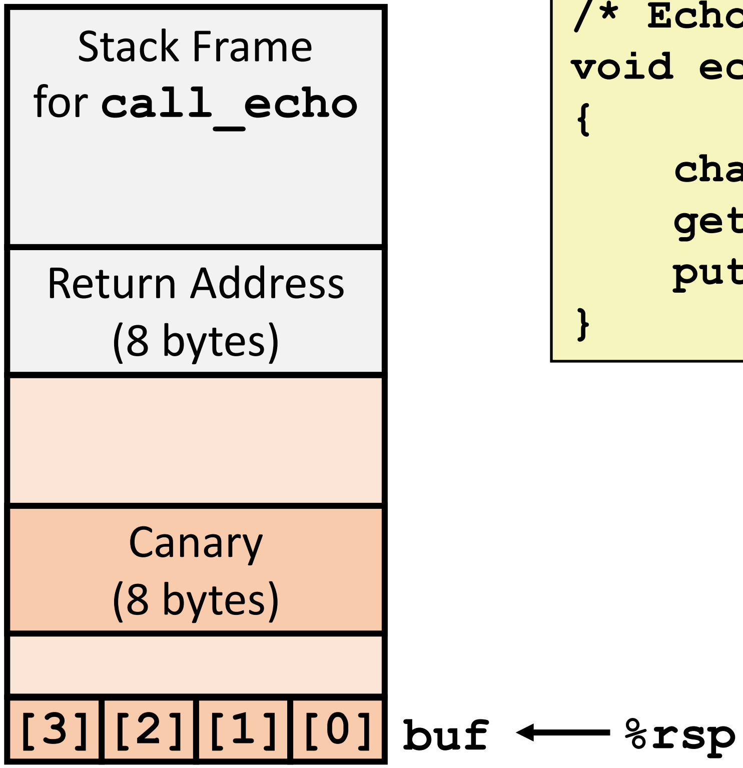
```
   40072f:   sub    $0x18,%rsp
   400733:   mov    %fs:0x28,%rax
   40073c:   mov    %rax,0x8(%rsp)
   400741:   xor    %eax,%eax
   400743:   mov    %rsp,%rdi
   400746:   callq  4006e0 <gets>
   40074b:   mov    %rsp,%rdi
   40074e:   callq  400570 <puts@plt>
   400753:   mov    0x8(%rsp),%rax
   400758:   xor    %fs:0x28,%rax
   400761:   je     400768 <echo+0x39>
   400763:   callq  400580 <__stack_chk_fail@plt>
   400768:   add    $0x18,%rsp
   40076c:   retq
```

**Aside: `%fs:0x28`**

- 使用分段寻址从内存中读取
- **Segment 是只读的**
- 每次程序运行时随机生成值

# 设置Canary

*Before call to gets*



Stack Frame for `call_echo`

Return Address (8 bytes)

Canary (8 bytes)

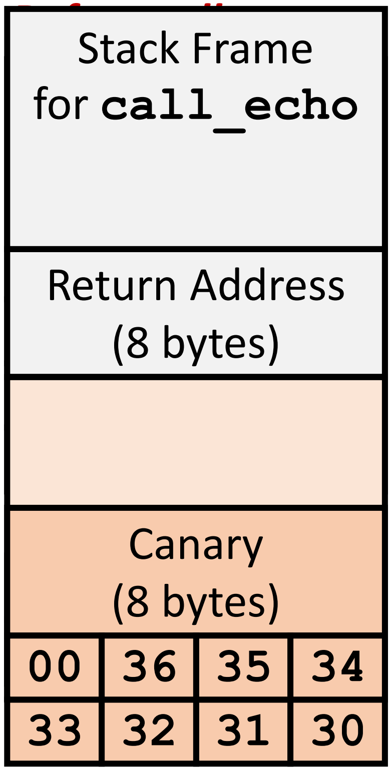
[3] [2] [1] [0] buf ← %rsp

```
/* 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 Canary

*After call to gets*



buf ⟵ %rsp

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

**Input:** ***0123456***

```
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:         . . .
```

# Attack Lab

- 祝贺完成“为正义而战”的Bomb Lab
- 欢迎来到Dark World！！！