Chp3 机器级程序设计

第5讲

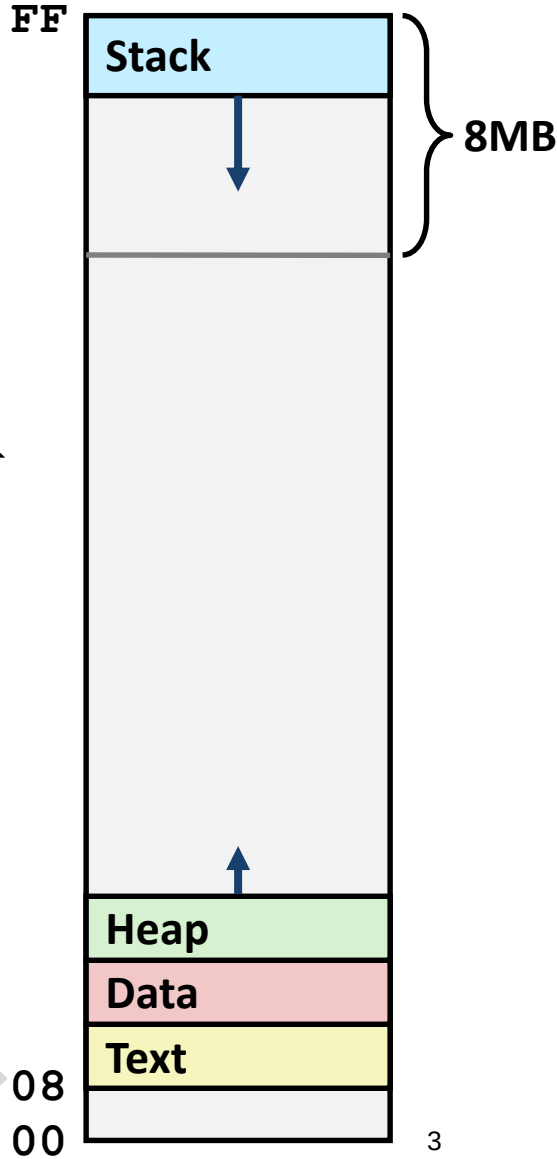
主要内容

- 数据传送
- 算术和逻辑操作
- 条件、循环、分支控制
- 堆栈、过程、递归和指针
- 数组、结构、联合
- **内存分配、越界访问、缓冲区溢出、X86-64**

IA32 Linux 内存布局

非比例设置

- 栈(stack)
  - 运行时栈 (8MB限制)，存放局部变量
- 堆(heap)
  - 动态分配存储
  - 当调用malloc(),calloc(),new()时
- 数据(data)
  - 静态分配的数据
  - E.g., 代码中声明的数组 &字符串
- 代码(text)
  - 可以执行的机器指令
  - 只读



内存分配的例子

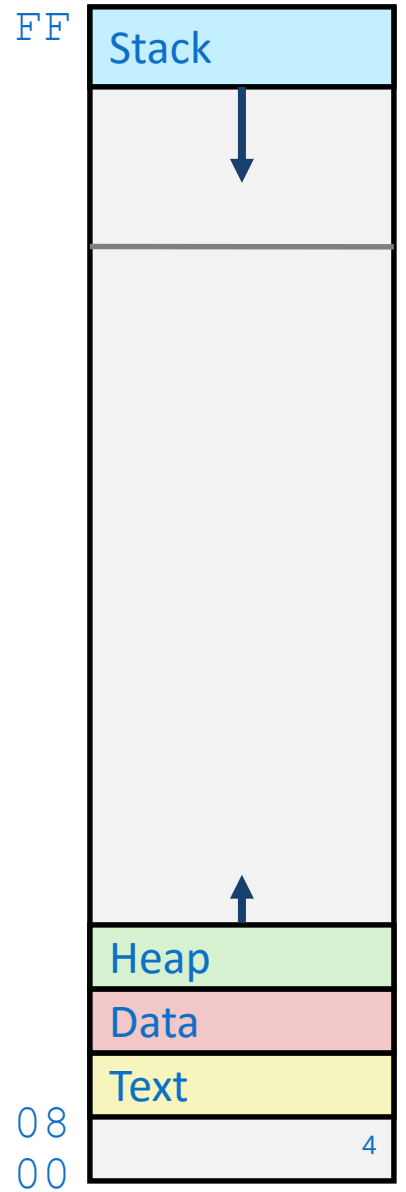
```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

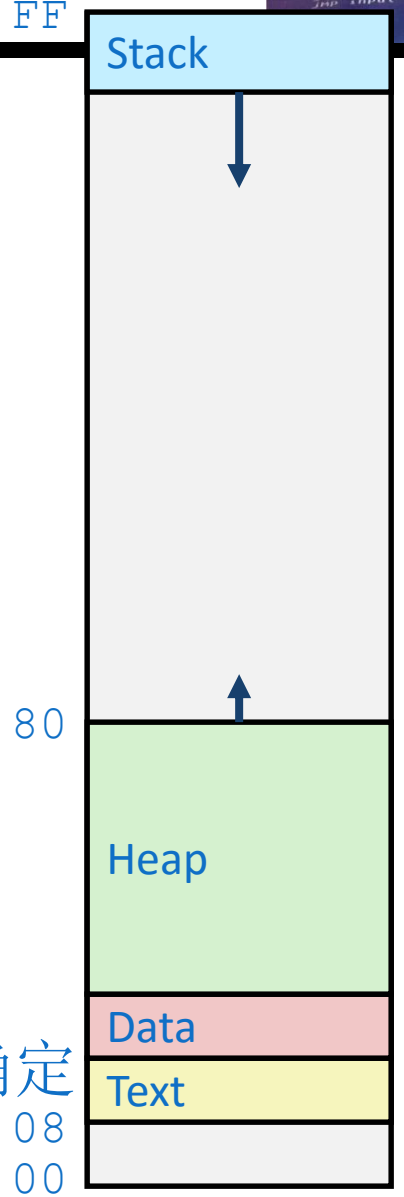
Where does everything go?



IA32内存分配示例

地址范围：2^{32}

$esp	0xffffbcd0
p3	0x65586008
p1	0x55585008
p4	0x1904a110
p2	0x1904a008
&p2	0x18049760
&beyond	0x08049744
big_array	0x18049780
huge_array	0x08049760
main()	0x080483c6
useless()	0x08049744
final malloc()	0x006be166

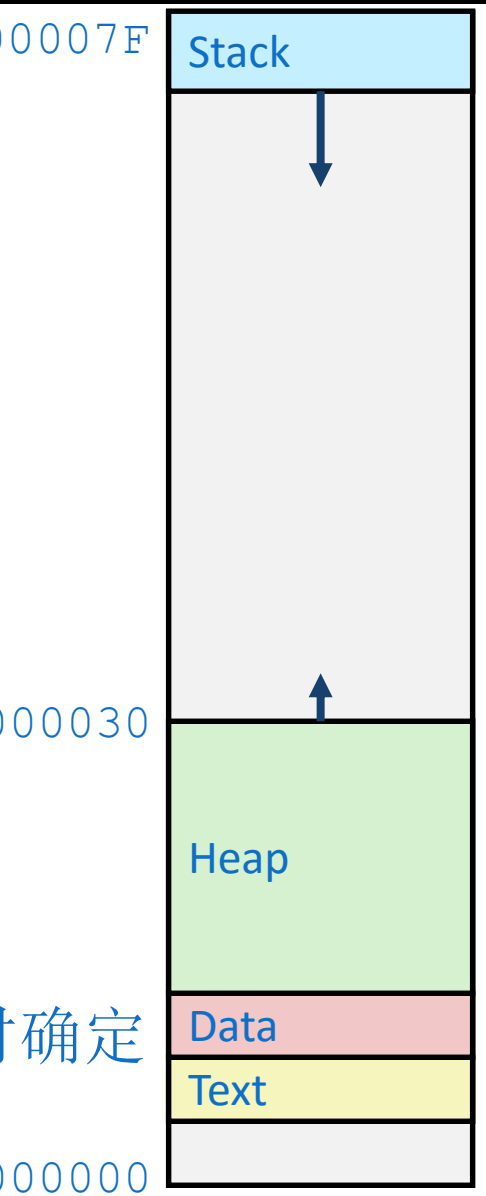


malloc()需要动态链接库，地址在运行时确定

x86-64内存分配示例

地址范围： $\sim 2^{47}$

$rsp	0x00007fffffff8d1f8
p3	0x00002aaabaadd010
p1	0x00002aaaaaadc010
p4	0x0000000011501120
p2	0x0000000011501010
&p2	0x0000000010500a60
&beyond	0x0000000000500a44
big_array	0x0000000010500a80
huge_array	0x0000000000500a50
main()	0x0000000000400510
useless()	0x0000000000400500
final malloc()	0x000000386ae6a170



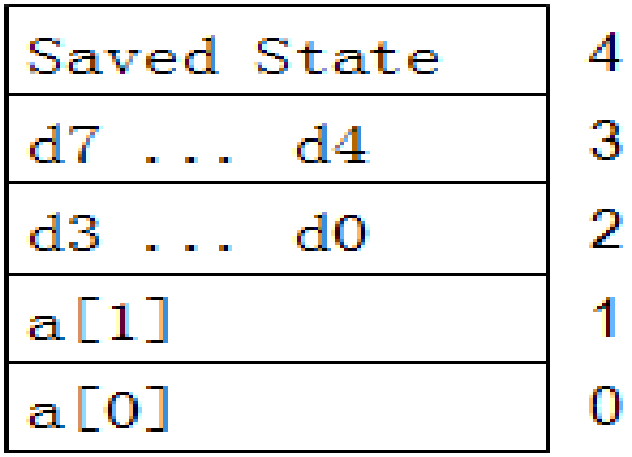
malloc()需要动态链接库，地址在运行时确定

越界访问和缓冲区溢出

大家还记得以下的例子吗？

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.0000061035156
fun(4) → 3.14, 然后存储保护错



为什么当 i>1 就有问题？

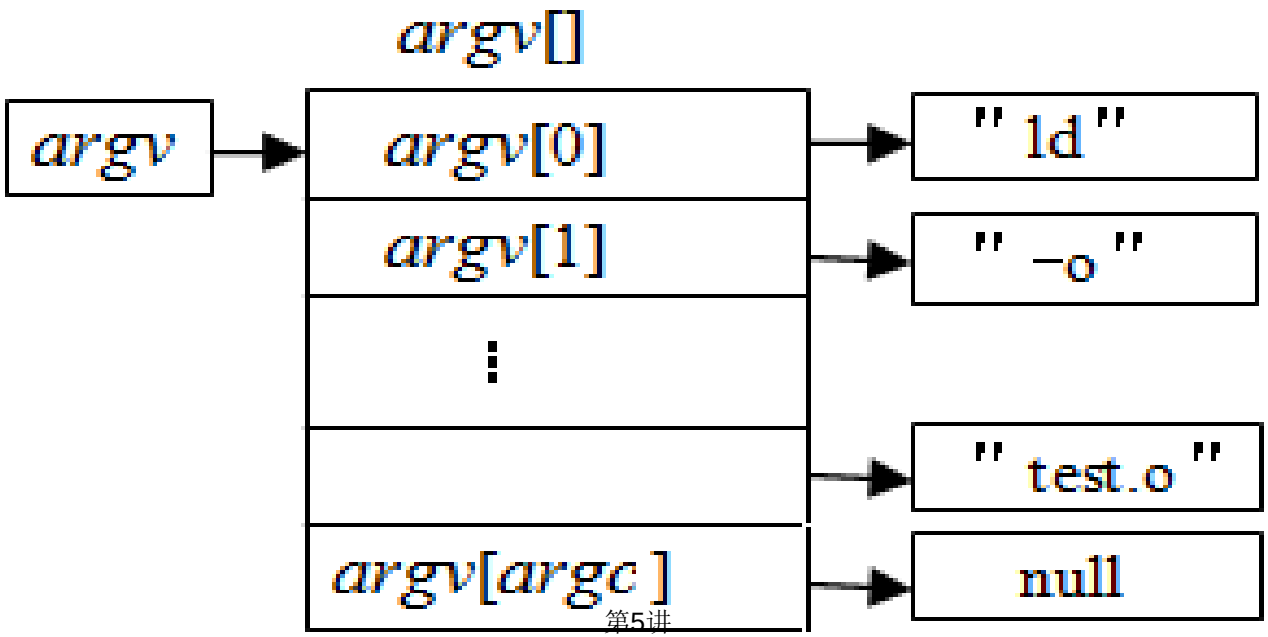
因为数组访问越界！

越界访问和缓冲区溢出

- C语言中的数组元素可使用指针来访问，因而对数组的引用没有边界约束，也即程序中对数组的访问可能会有意或无意地超越数组存储区范围而无法发现。
- 数组存储区可看成是一个缓冲区，超越数组存储区范围的写入操作称为缓冲区溢出。
- 例如，对于一个有10个元素的char型数组，其定义的缓冲区有10个字节。若写一个字符串到这个缓冲区，那么只要写入的字符串多于9个字符（结束符 ‘\0’ 占一个字节），就会发生“写溢出”。
- 缓冲区溢出是一种非常普遍、非常危险的漏洞，在各种操作系统、应用软件中广泛存在。
- 缓冲区溢出攻击是利用缓冲区溢出漏洞所进行的攻击行动。利用缓冲区溢出攻击，可导致程序运行失败、系统关机、重新启动等后果。

main()函数的原型

- 主函数main()的原型形式如下：
  int main(int argc, char **argv, char **envp);　或
  int main(int argc, char *argv[], char *envp[]);
  argc：参数列表长度，参数列表中开始是命令名（可执行文件名），最后以NULL结尾。例: 当“.\hello” 时，argc=1
  例：当“ld -o test main.o test.o” 时，argc=5



越界访问和缓冲区溢出

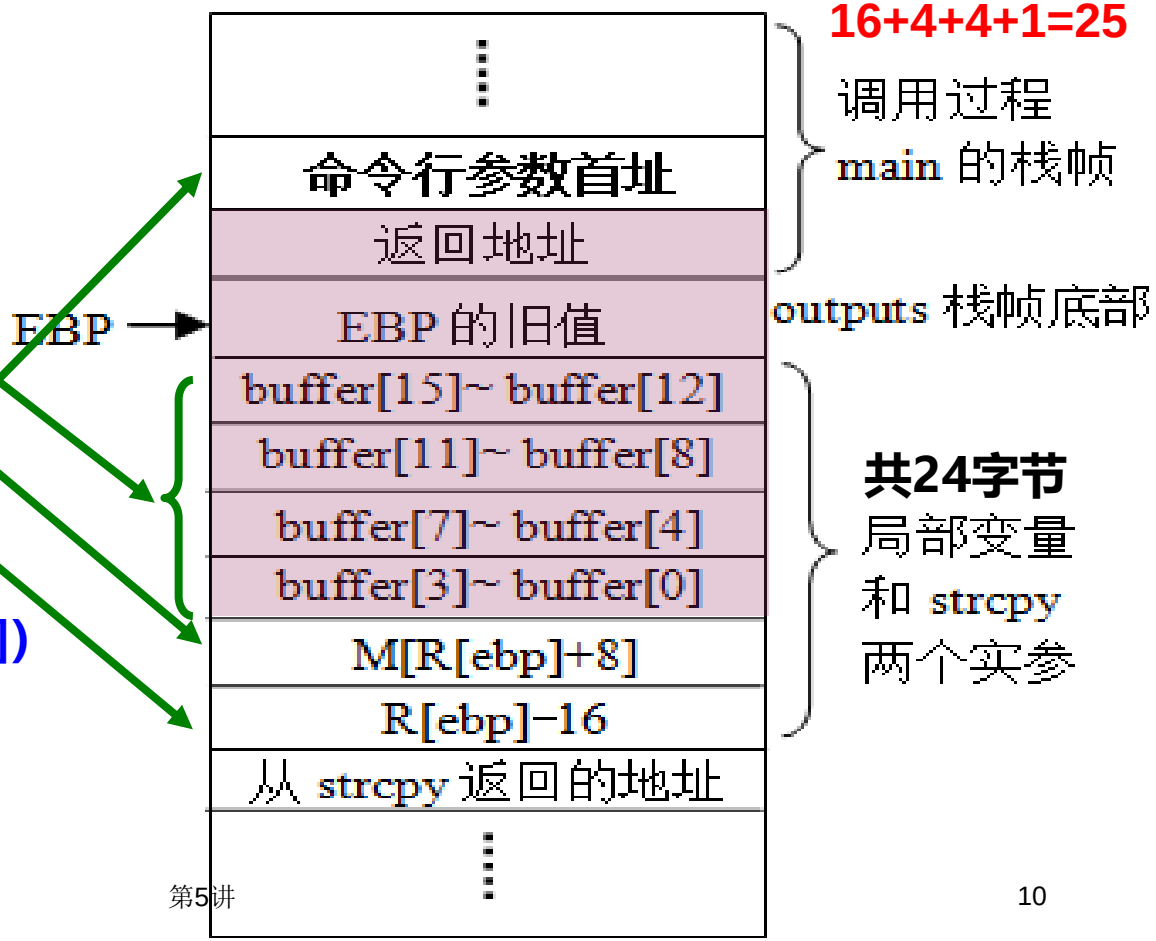
- 造成缓冲区溢出的原因是没有对栈中作为缓冲区的数组的访问进行越界检查。

举例：利用缓冲区溢出转到自设的程序hacker去执行

outputs漏洞：当命令行中字符串超25个字符时，使用strcpy函数就会使缓冲buffer造成写溢出并破坏返址

```
#include "stdio.h"
#include "string.h"
void outputs(char *str)
{
    char buffer[16];
    strcpy(buffer,str);
    printf("%s \n", buffer);
}
void hacker(void)
{
    printf("being hacked\n");
}
int main(int argc, char *argv[])
{
    outputs(argv[1]);
    return 0;
}
```

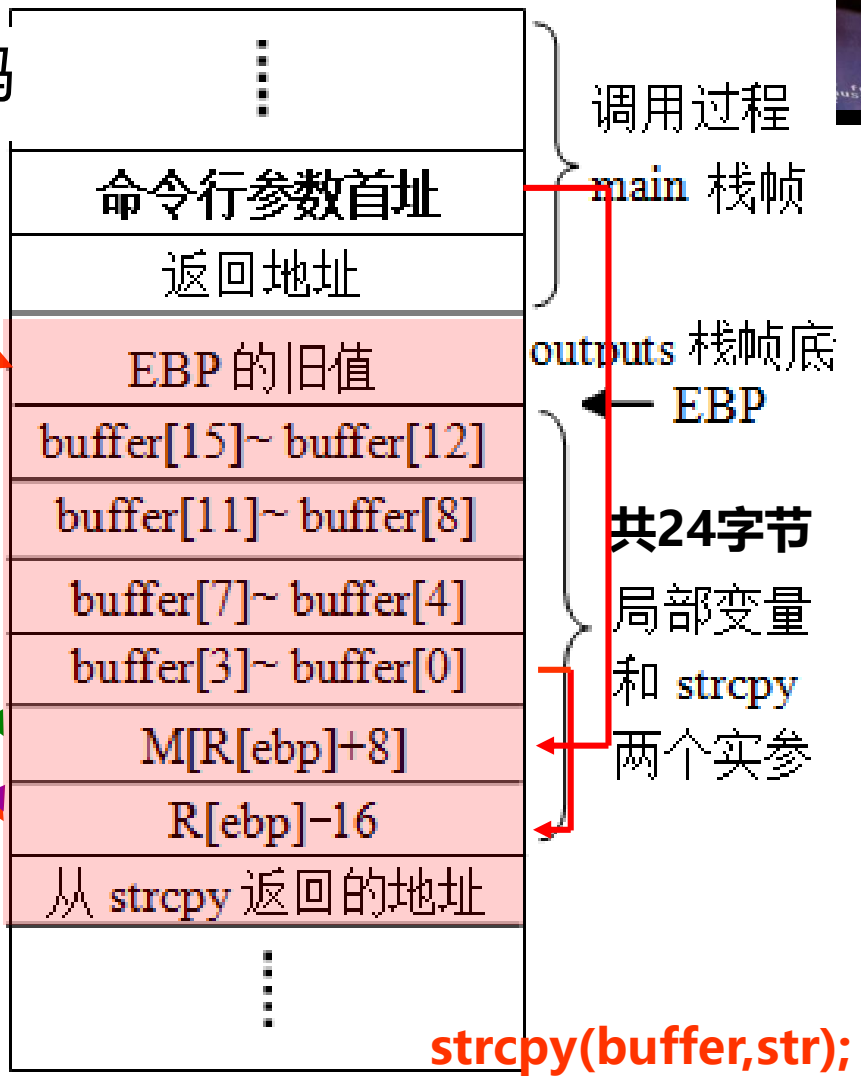
假定可执行文件名为test



越界访问和缓冲区溢出

test被反汇编得到的outputs汇编代码

```
080483e4 push   %ebp
080483e5 mov    %esp,%ebp
080483e7 sub    $0x18,%esp
080483ea mov    0x8(%ebp),%eax
080483ed mov    %eax,0x4(%esp)
080483f1 lea    0xfffffff0(%ebp),%eax
080483f4 mov    %eax,(%esp)
080483f7 call   0x8048330 <strcpy>
080483fc lea    0xfffffff0(%ebp),%eax
080483ff mov    %eax,0x4(%esp)
08048403 movl   $0x8048500,(%esp)
0804840a call   0x8048310
0804840f  leave
08048410 ret
```



若strcpy复制了25个字符到buffer中，并将hacker首址置于结束符 '\0' 前4个字节，则在执行strcpy后，hacker代码首址被置于main栈帧返回地址处，当执行outputs代码的ret指令时，便会转到hacker函数实施攻击。

程序的加载和运行

- UNIX/Linux系统中，可通过调用execve()函数来加载并执行程序。
- execve()函数的用法如下：

  int execve(char *filename, char *argv[], *envp[]);

  filename是加载并运行的可执行文件名(如./hello)，可带参数列表argv和环境变量列表envp。若错误（如找不到指定文件filename），则返回-1，并将控制权交给调用程序； 若函数执行成功，则不返回，最终将控制权传递到可执行目标中的主函数main。
- 主函数main()的原型形式如下：

  int main(int argc, char **argv, char **envp); 或者：

  int main(int argc, char *argv[], char *envp[]);

前述例子： ".\test 0123456789ABCDEFXXXX▥ ▧▥▧" ,argc=2

argv[0] **argv[1]**

缓冲区溢出攻击

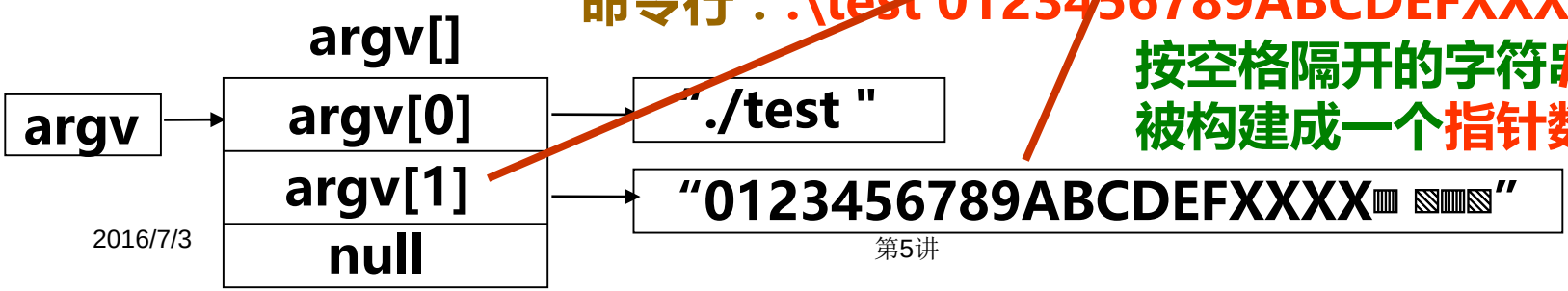
```
#include "stdio.h"
char code[]=
    "0123456789ABCDEFXXXX"
    "\x11\x84\x04\x08"
    "\x00";
int main(void)
{
    char *argv[3];
    argv[0]="./test";
    argv[1]=code;
    argv[2]=NULL;
    execve(argv[0],argv,NULL);
    return 0;
}
```

```
#include "stdio.h"
#include "string.h"
void outputs(char *str)
{
    char buffer[16];
    strcpy(buffer,str);
    printf("%s \n", buffer);
}
void hacker(void)
{
    printf("being hacked\n");
}
int main(int argc, char *argv[])
{
    outputs(argv[1]);
    return 0;
}
```

可执行文件名为test

命令行：.\test 0123456789ABCDEFXXXX▯ ▧▯▧

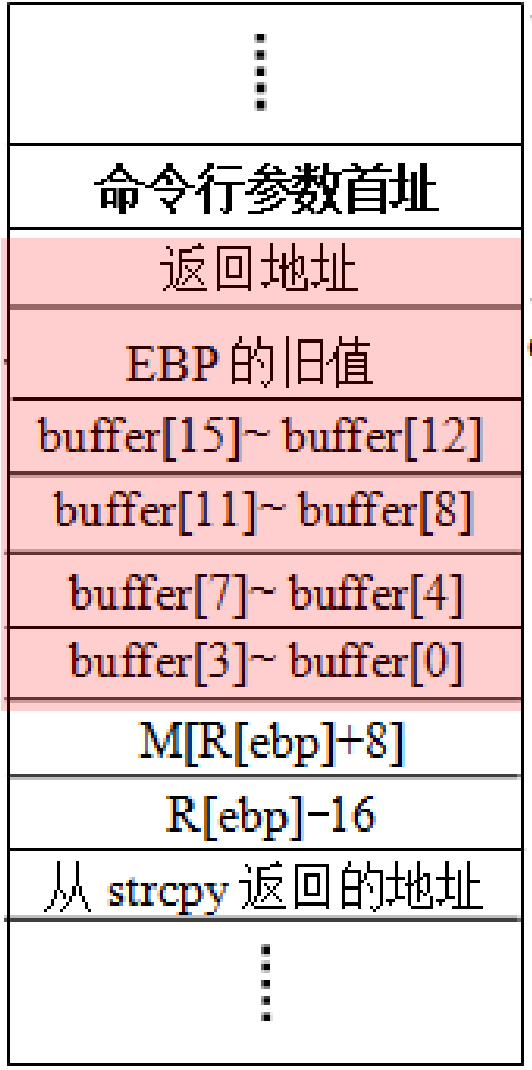
按空格隔开的字符串
被构建成一个指针数组



越界访问和缓冲区

假定hacker首址为0x08048411

```
void hacker(void) {
    printf("being hacked\n");
}
#include "stdio.h"
char code[]=
      "0123456789ABCDEFXXXX"
      "\x11\x84\x04\x08"
      "\x00";
int main(void) {
      char *argv[3];
      argv[0]="./test";
      argv[1]=code;
      argv[2]=NULL;
      execve(argv[0],argv,NULL);
      return 0;
}
```



执行上述攻击程序后的输出结果为：

"0123456789ABCDEFXXXX

being hacked

Segmentation fault

最后显示“Segmentation fault”，原因是执行到hacker过程的ret指令时取到的“返回地址”是一个不确定的值，因而可能跳转到数据区或系统区或其他非法访问的存储区执行，因而造成段错误。

缓冲区溢出攻击的防范（自学）

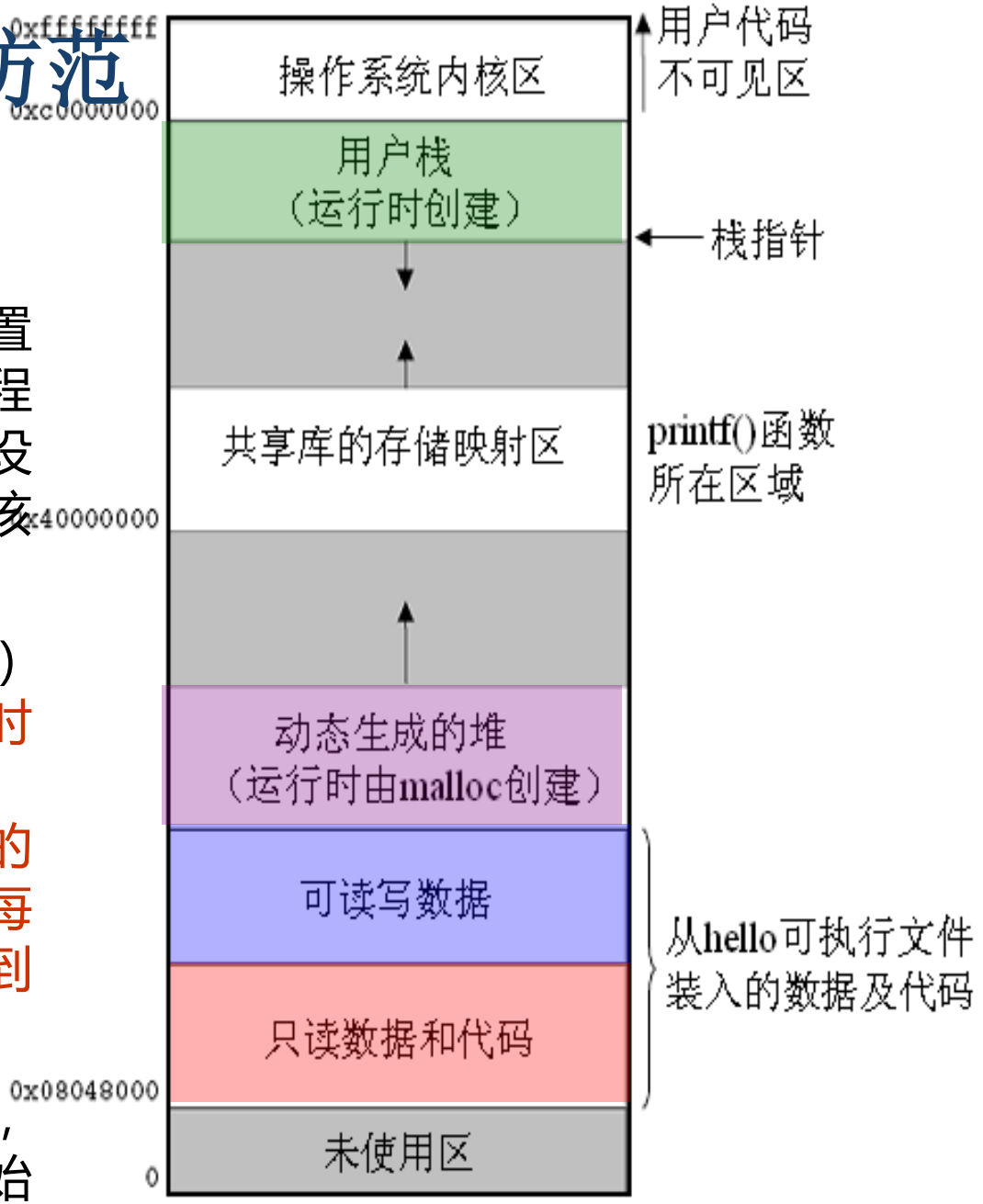
- 两个方面的防范
  - 从程序员角度去防范
    - 用辅助工具帮助程序员查漏，例如，用grep来搜索源代码中容易产生漏洞的库函数（如strcpy和sprintf等）的调用；用fault injection查错
  - 从编译器和操作系统方面去防范
    - 地址空间随机化ASLR

      是一种比较有效的防御缓冲区溢出攻击的技术

      目前在Linux、FreeBSD和Windows Vista等OS使用
    - 栈破坏检测
    - 可执行代码区域限制
    - 等等

缓冲溢出攻击防范

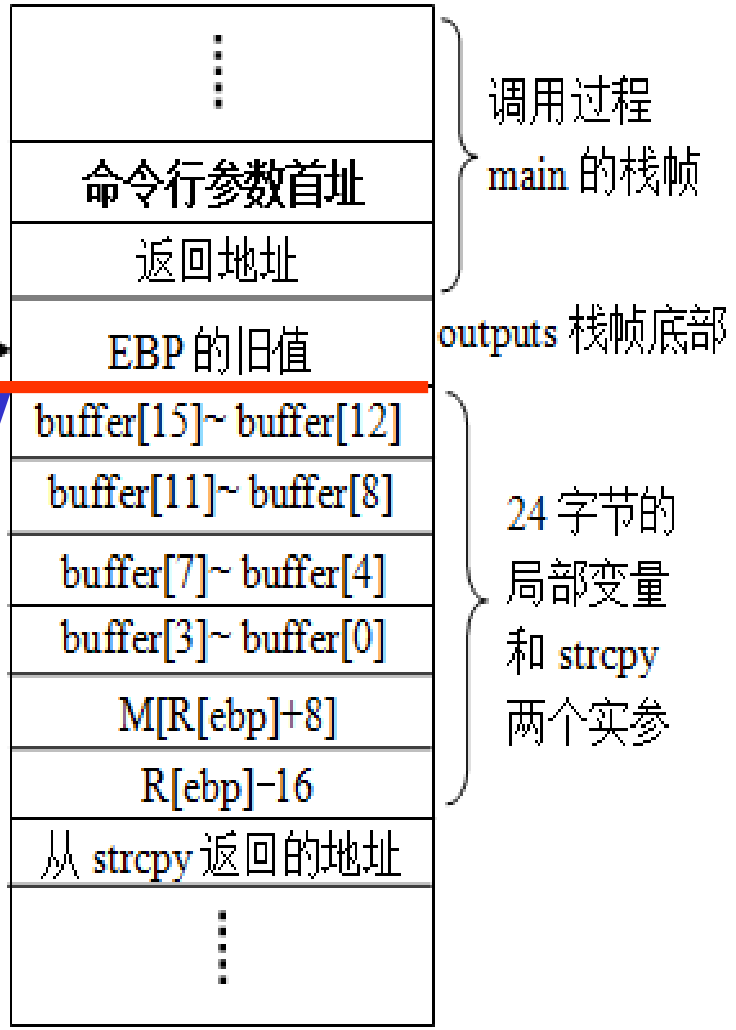
- 地址空间随机化
  - 只要操作系统相同，则栈位置就一样，若攻击者知道漏洞程序使用的栈地址空间，就可设计一个针对性攻击，在使用该程序机器上实施攻击
  - 地址空间随机化（栈随机化）的基本思路是，将加载程序时生成的代码段、静态数据段、堆区、动态库和栈区各部分的首地址进行随机化处理，使每次启动时，程序各段被加载到不同地址起始处
  - 对于随机生成的栈起始地址，攻击者不太容易确定栈的起始位置



Linux虚拟地址空间映像

缓冲区溢出攻击的防范

- 栈破坏检测
  - 若在程序跳转到攻击代码前能检测出程序栈已被破坏，就可避免受到严重攻击
  - 新GCC版本在代码中加入了一种栈保护者（stack protector）机制，用于检测缓冲区是否越界
  - 主要思想：在函数准备阶段，在其栈帧中缓冲区底部与保存寄存器之间（如buffer[15]与保留的EBP之间）加入一个随机生成的特定值；在函数恢复阶段，在恢复寄存器并返回到调用函数前，先检查该值是否被改变。若改变则程序异常中止。因为插入在栈帧中的特定值是随机生成的，所以攻击者很难猜测出它是什么



缓冲区溢出攻击的防范

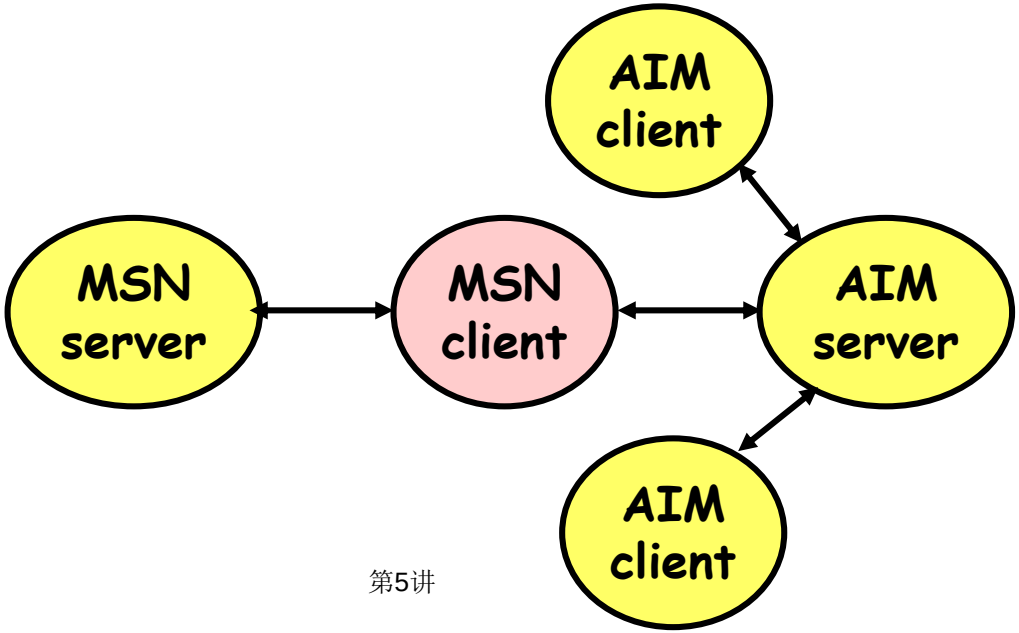
- 可执行代码区域限制
  - 通过将程序栈区和堆区设置为不可执行，从而使得攻击者不可能执行被植入在输入缓冲区的代码，这种技术也被称为非执行的缓冲区技术。
  - 早期Unix系统只有代码段的访问属性是可执行，其他区域的访问属性是可读或可读可写。但是，近来Unix和Windows系统由于要实现更好的性能和功能，允许在栈段中动态地加入可执行代码，这是缓冲区溢出的根源。
  - 为保持程序兼容性，不可能使所有数据段都设置成不可执行。不过，可以将动态的栈段设置为不可执行，这样，既保证程序的兼容性，又可以有效防止把代码植入栈（自动变量缓冲区）的溢出攻击。

缓冲区溢出Buffer Overflow

- Vulnerability
- Protection

Internet蠕虫和IM大战

- 1988年11月
  - Internet 蠕虫攻击了成千上万Internet主机
  - 它是如何发生的?
- 1999年7月
  - Microsoft 推出MSN Messenger (即时消息系统).
  - Messenger客户端能够访问流行的AOL Instant Messaging(AIM)服务器



Internet蠕虫和IM大战(续)

- 1999年8月
  - 神秘地, Messenger 客户端不能登陆AIM服务器.
  - Microsoft 和AOL开始了IM 大战:
    - AOL修改了服务器不允许Messenger客户端
    - Microsoft客户端不断尝试模拟AOL IM的协议，至少13次
    - 但是AOL就是能够确定一个用户运行的客户端版本.
  - 这是怎么回事?

- Internet 蠕虫和AOL/Microsoft大战都是基于栈缓冲区溢出漏洞!
  - 许多Unix函数不做参数大小的检查
  - 允许目标缓冲溢出.

字符串库函数

- Unix中gets()函数的实现

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

  - 没有办法限制要读入的字符串大小
- 相似的问题出现在Unix其它函数中
  - Strcpy,strcat: 任意长度字符串拷贝
  - scanf, fscanf, sscanf, 当给出%s格式转换符

脆弱的缓冲区代码

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

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

```
unix>./bufdemo
Type a string:1234567
1234567
```

```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

```
unix>./bufdemo
Type a string:123456789ABC
Segmentation Fault
```

缓冲区溢出反汇编

echo:

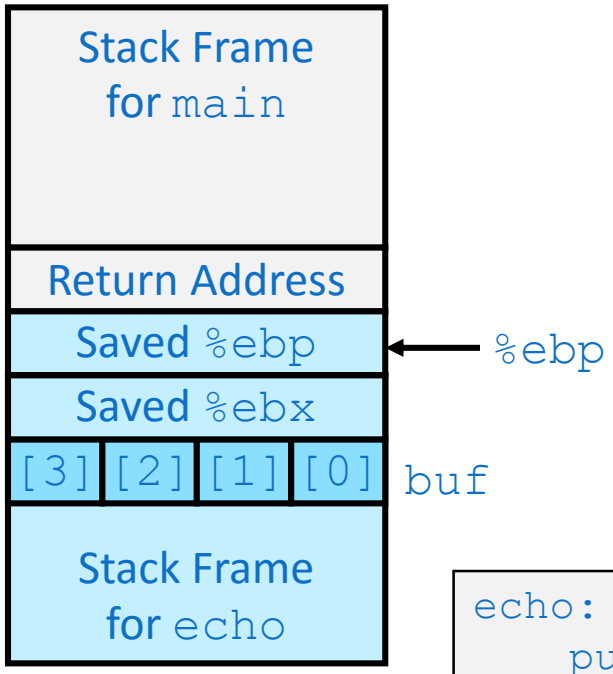
```
80485c5:  55                    push   %ebp
80485c6:  89 e5                 mov    %esp,%ebp
80485c8:  53                    push   %ebx
80485c9:  83 ec 14              sub    $0x14,%esp
80485cc:  8d 5d f8              lea 0xfffffff8(%ebp),%ebx
80485cf:  89 1c 24              mov    %ebx,(%esp)
80485d2:  e8 9e ff ff ff        call   8048575 <gets>
80485d7:  89 1c 24              mov    %ebx,(%esp)
80485da:  e8 05 fe ff ff        call   80483e4 <puts@plt>
80485df:  83 c4 14              add    $0x14,%esp
80485e2:  5b                    pop    %ebx
80485e3:  5d                    pop    %ebp
80485e4:  c3                    ret
```

call_echo:

```
80485eb:  e8 d5 ff ff ff        call   80485c5 <echo>
80485f0:  c9                    leave
80485f1:  c3                    ret
```

缓冲区溢出时的栈

Before call to gets



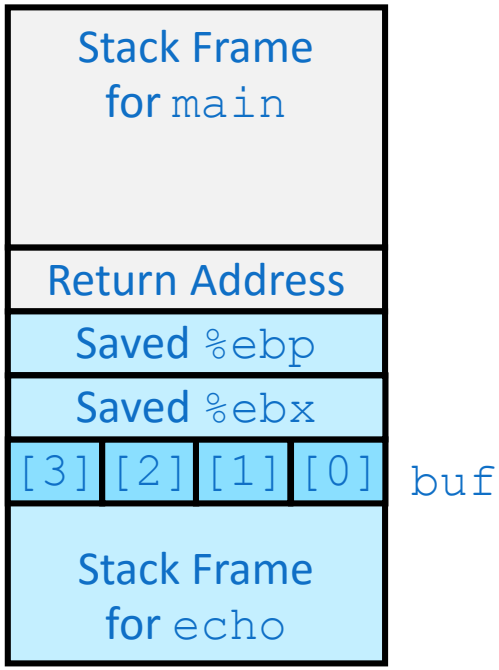

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    pushl %ebp              # Save %ebp on stack
    movl  %esp, %ebp
    pushl %ebx              # Save %ebx
    subl  $20, %esp         # Allocate stack space
    leal  -8(%ebp),%ebx     # Compute buf as %ebp-8
    movl  %ebx, (%esp)      # Push buf on stack
    call  gets              # Call gets
    . . .
```

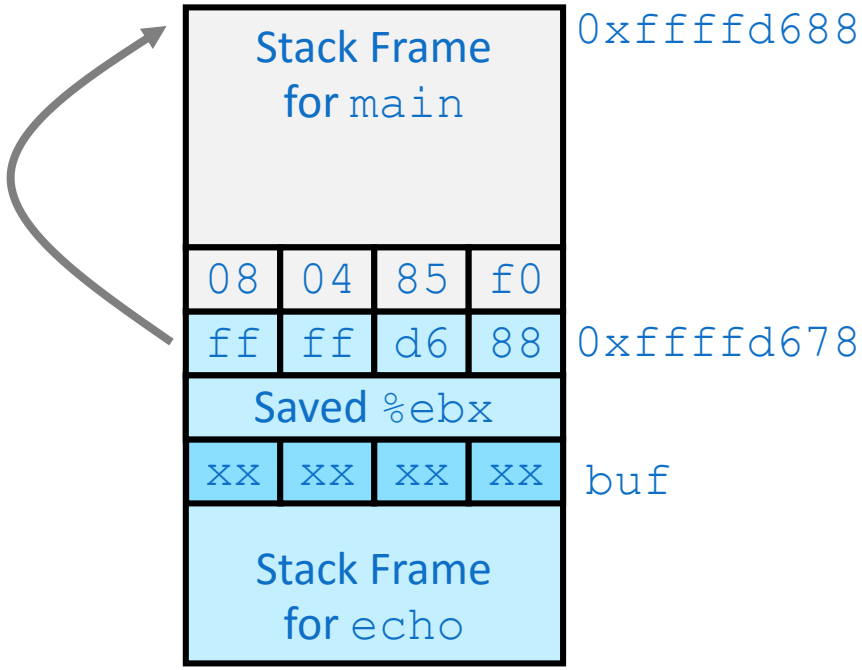
缓冲区溢出栈例子

```
unix> gdb bufdemo
(gdb) break echo
Breakpoint 1 at 0x80485c9
(gdb) run
Breakpoint 1, 0x80485c9 in echo ()
(gdb) print /x $ebp
$1 = 0xffffd678
(gdb) print /x *(unsigned *)$ebp
$2 = 0xffffd688
(gdb) print /x *((unsigned *)$ebp + 1)
$3 = 0x80485f0
```

Before call to gets

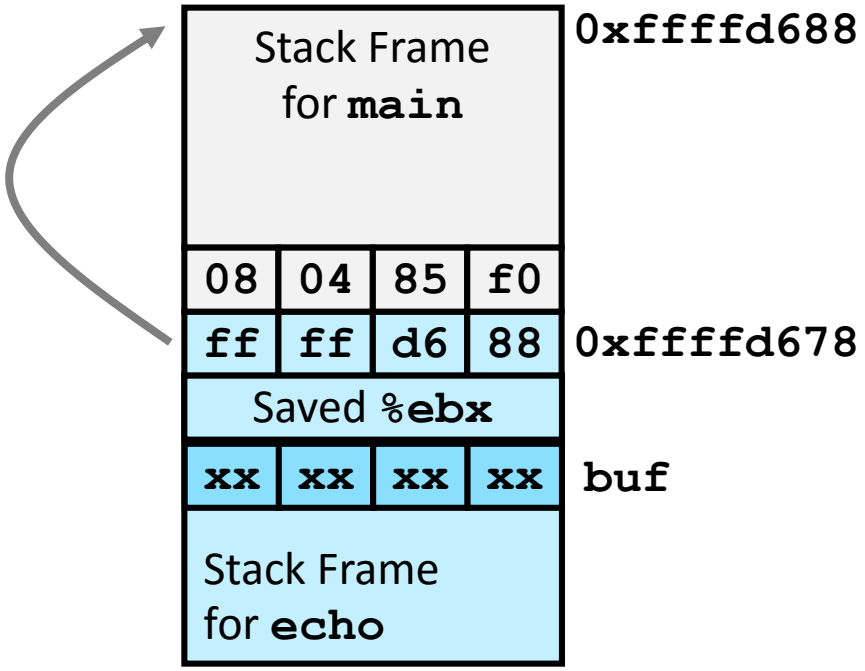


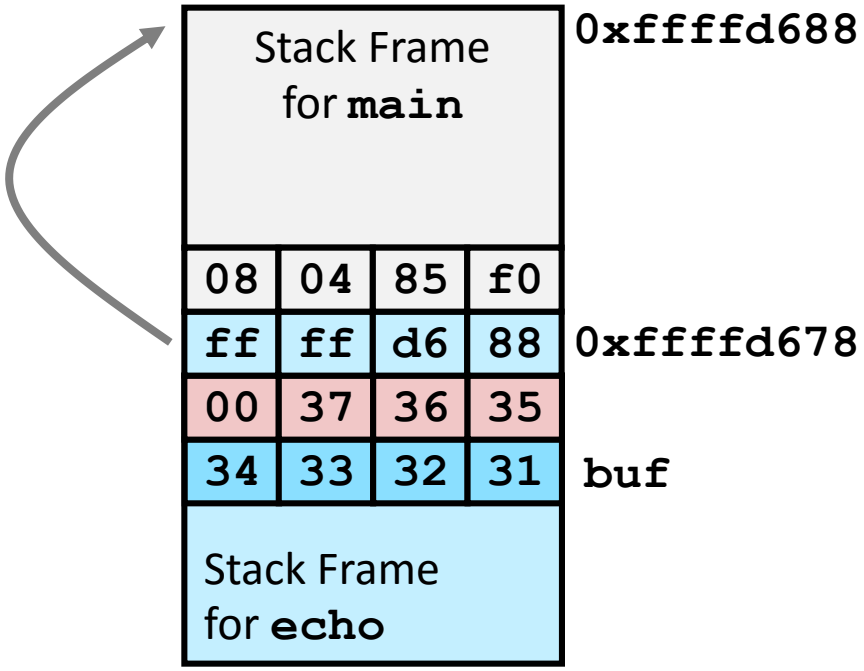
Before call to gets




```
80485eb:  e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:  c9                leave
```

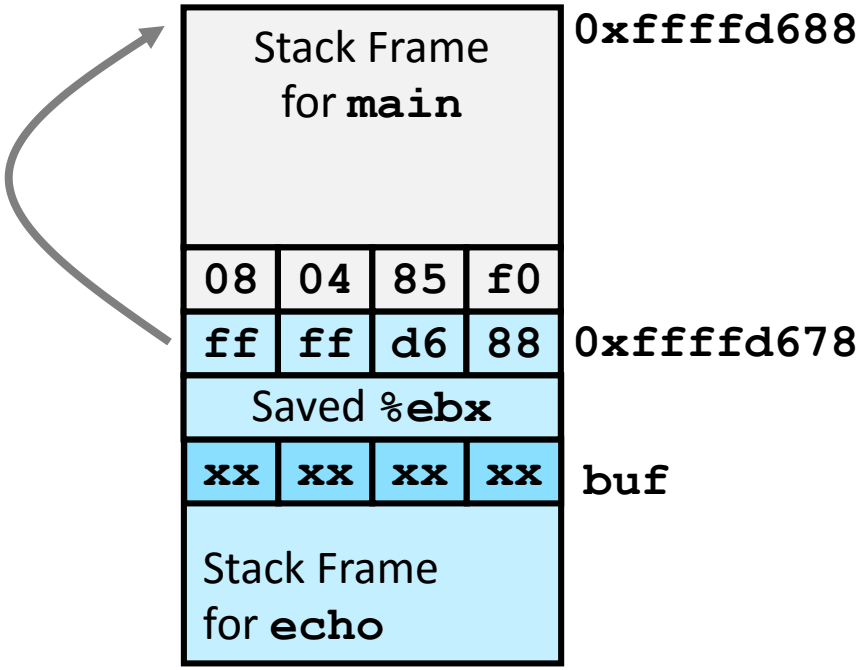
缓冲区溢出例子#1

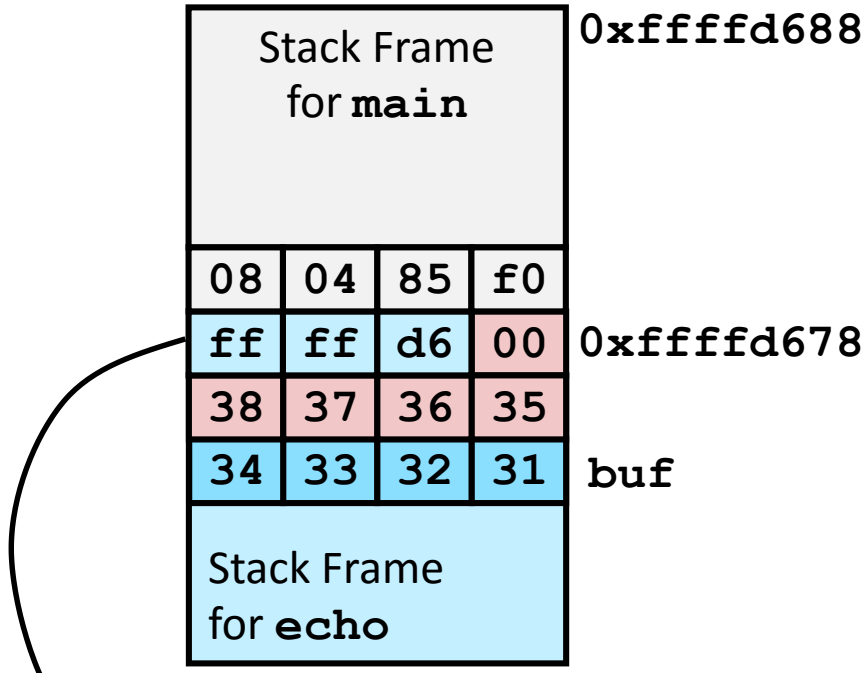




缓冲区溢出,破坏 %ebx内容,
但没有引发问题

缓冲区溢出栈例子#2



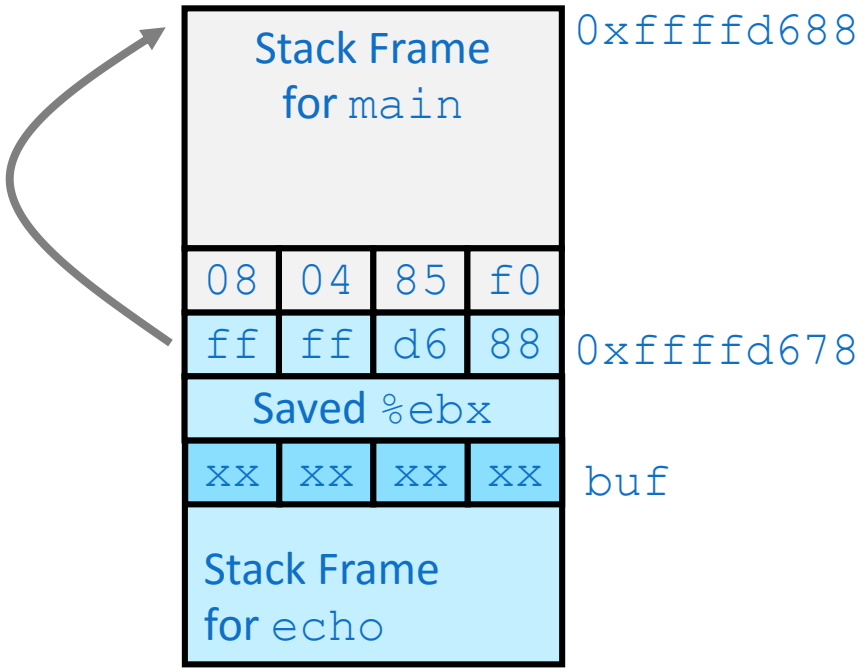


破坏 %ebp内容，段地址错

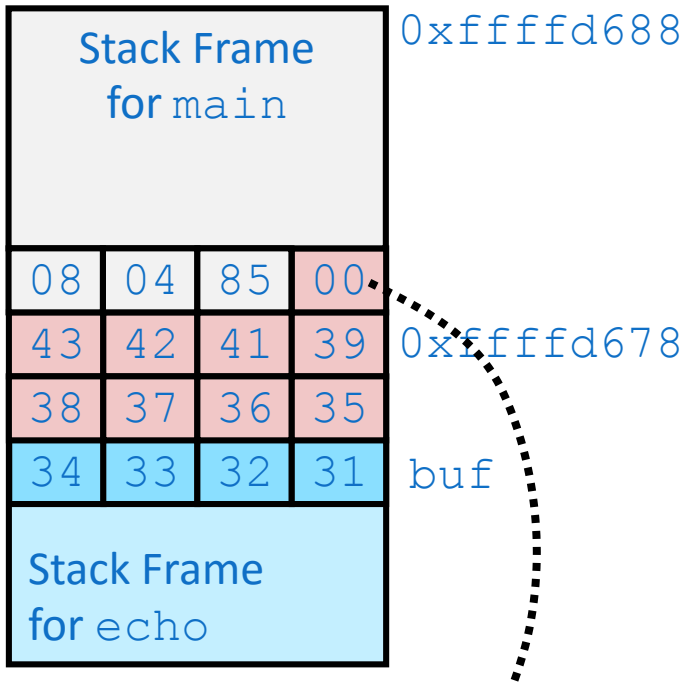
```
. . .
80485eb:    e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:    c9                leave    # Set %ebp to corrupted value
80485f1:    c3                ret
```

缓冲区溢出栈例子#3

Before call to gets



Input 123456789ABC



%ebp 和返回地址被破坏

不再指向期望的返回点

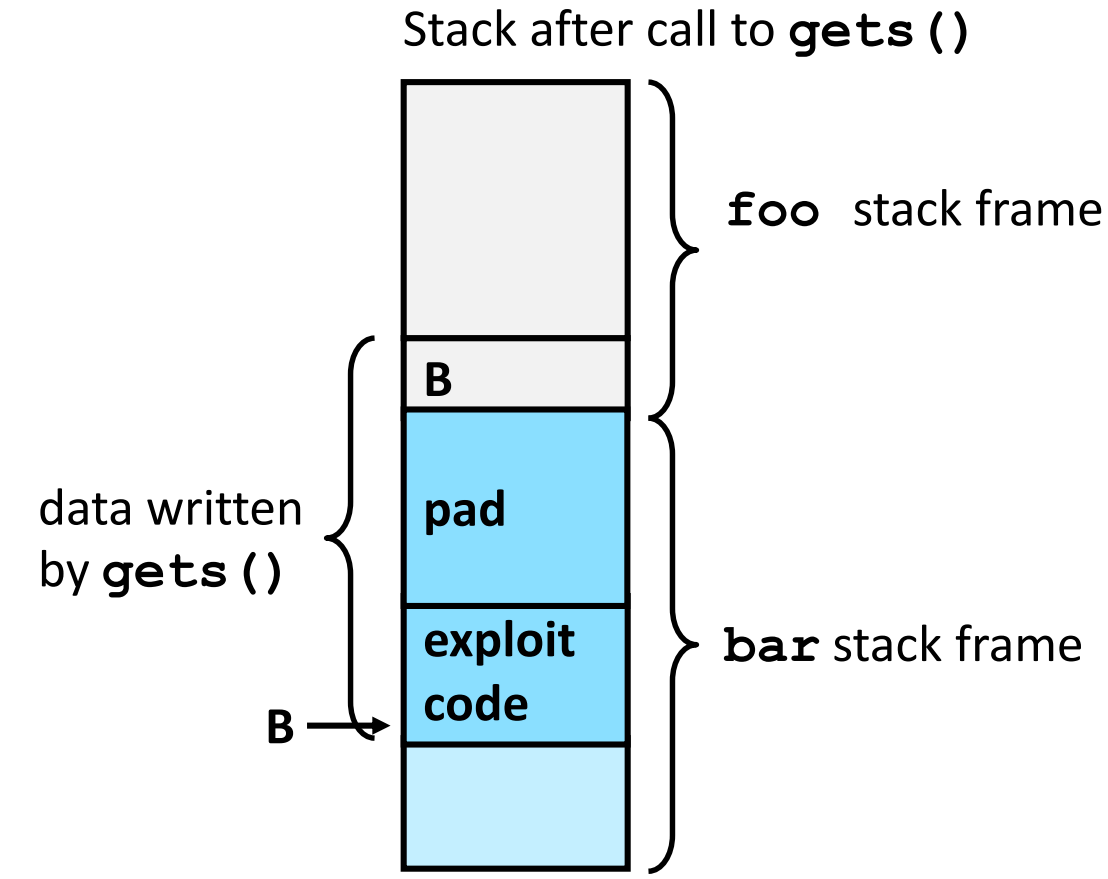
```
80485eb:    e8 d5 ff ff ff    call   80485c5 <echo>
80485f0:    c9                leave    # Desired return point
```

恶意利用缓冲区溢出

```
void
foo(){
   bar();
   ...
}
```

Return address A

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



- 输入字符串包含可执行代码的字节表示
- 用缓冲区的地址覆盖返回地址
- 当 bar() 执行ret时, 会跳到漏洞代码处

基于缓冲区溢出的漏洞

- *缓冲区溢出错误允许攻击者在感染机器上执行任意代码*
- Internet蠕虫
  - 早期版本的finger服务器用`gets()`从客户端获得参数:
    - `finger droh@cs.cmu.edu`
  - 蠕虫攻击fingerd服务器通过发送假的参数给服务器:
    - `finger "exploit-code padding new-return-address"`
    - 漏洞入侵代码: 在感染机器上执行root权限的shell程序，直接TCP连接到攻击者

IM War正是基于缓冲区溢出

- *缓冲区溢出允许远程攻击者在系统中执行任意代码*
- IM（即时消息） War
  - AOL 利用 AIM客户端缓冲区溢出的漏洞
  - 漏洞入侵代码: 返回4个字节的特征码 (这些字节位于客户端的某些位置) 给服务器.
  - 当Microsoft改变代码适合特征码时，AOL改变取特征码的位置.

- 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

后来被发现这封邮件是从Microsoft内部发的!

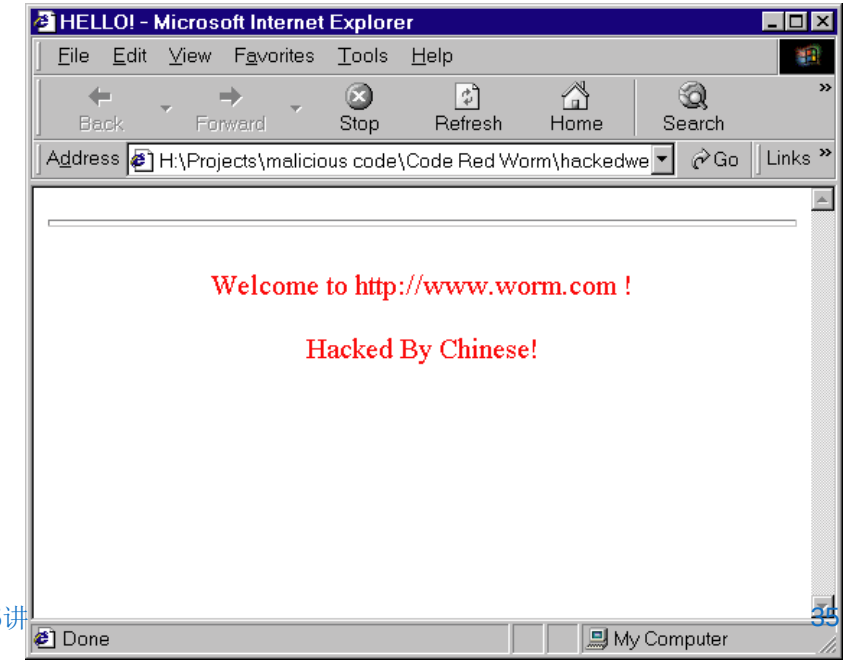
红色代码蠕虫

- 历史
  - 2001年6月18日。Microsoft宣布IIS Internet服务器容易受到缓冲区溢出漏洞
  - 2001年7月19日。 超过250,000台机器9小时内全部感染
  - 白宫必须改变它的IP地址。五角大楼也关闭共网服务器一天。
- 当设置CS:APP站点时
  - 收到的字符串形式

```
GET
   /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
   N....NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN%u909
   0%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u
   6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u8b0
   0%u531b%u53ff%u0078%u0000%u00=a
HTTP/1.0" 400 325 "-" "-"
```

红色代码

- 同时启动100线程运行
- 自我传播Spread self
  - 生成随机IP地址并且发送攻击字符串
  - 每个月1号和19号
- 攻击网站 www.whitehouse.gov
  - 发送 98,304数据包; 等待4-1/2 小时; 重复
    - 拒绝服务攻击
  - 每个月21号和27号
- 损害服务器主页
  - 在等待2小时以后



红色代码影响

- 后面的版本甚至更加厉害
  - 红色代码 II
  - 在2002年4月, 超过18,000 台机器感染
  - 还在传播
- 为NIMDA所仿效
  - 多种传播途径
  - 一种就是利用红色代码 II的传播路径
- ASIDE (security flaws start at home)
  - .rhosts used by Internet Worm
  - Attachments used by MyDoom (1 in 6 emails Monday morning!)

避免溢出漏洞

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

- 使用限制串长度的库函数
  - 用fgets代替gets
  - 用strncpy 代替strcpy
  - 不再使用带有%s格式转换符的scanf
    - 使用fgets 来读取字符串
    - 或者使用%ns，n为合适的整数。

系统层面的保护

- Randomized stack offsets
  - At start of program, allocate random amount of space on stack
  - Makes it difficult for hacker to predict beginning of inserted code

```
unix> gdb bufdemo
(gdb) break echo

(gdb) run
(gdb) print /x $ebp
$1 = 0xffffc638

(gdb) run
(gdb) print /x $ebp
$2 = 0xffffbb08

(gdb) run
(gdb) print /x $ebp
$3 = 0xffffc6a8
```

- Nonexecutable code segments
  - In traditional x86, can mark region of memory a either “read-only” or “writeable”
    - Can execute anything readable
  - X86-64 added explicit “execute” permission

堆栈“金丝雀”

- Idea
  - Place special value (“canary”) on stack just beyond buffer
  - Check for corruption before exiting function
- GCC Implementation
  - **`-fstack-protector`**
  - **`-fstack-protector-all`**

```
unix>./bufdemo-protected
Type a string:1234
1234
```

```
unix>./bufdemo-protected
Type a string:12345
*** stack smashing detected ***
```

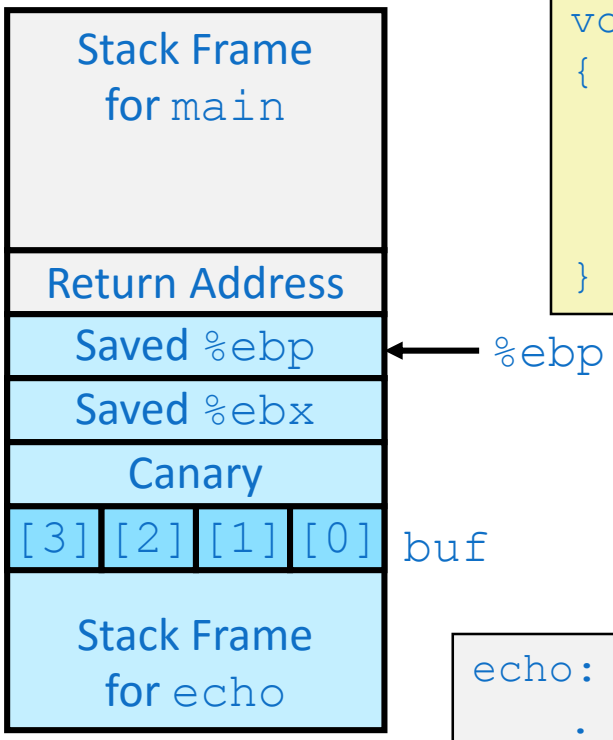
Protected Buffer Disassembly

echo:

```
804864d:  55                    push   %ebp
 804864e: 89 e5                 mov    %esp,%ebp
 8048650: 53                    push   %ebx
 8048651: 83 ec 14              sub    $0x14,%esp
 8048654: 65 a1 14 00 00 00     mov    %gs:0x14,%eax
 804865a: 89 45 f8              mov    %eax,0xfffffff8(%ebp)
 804865d: 31 c0                 xor    %eax,%eax
 804865f: 8d 5d f4              lea    0xfffffff4(%ebp),%ebx
 8048662: 89 1c 24              mov    %ebx,(%esp)
 8048665: e8 77 ff ff ff        call   80485e1 <gets>
 804866a: 89 1c 24              mov    %ebx,(%esp)
 804866d: e8 ca fd ff ff        call   804843c <puts@plt>
 8048672: 8b 45 f8              mov    0xfffffff8(%ebp),%eax
 8048675: 65 33 05 14 00 00 00  xor    %gs:0x14,%eax
 804867c: 74 05                 je     8048683 <echo+0x36>
 804867e: e8 a9 fd ff ff        call   804842c <FAIL>
 8048683: 83 c4 14              add    $0x14,%esp
 8048686: 5b                    pop    %ebx
 8048687: 5d                    pop    %ebp
 8048688: c3                    ret
```

建立“Canary”

Before call to gets




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

```
echo:
    . . .
    movl    %gs:20, %eax     # Get canary
    movl    %eax, -8(%ebp)   # Put on stack
    xorl    %eax, %eax       # Erase canary
    . . .
```

检查“Canary”

Before call to gets

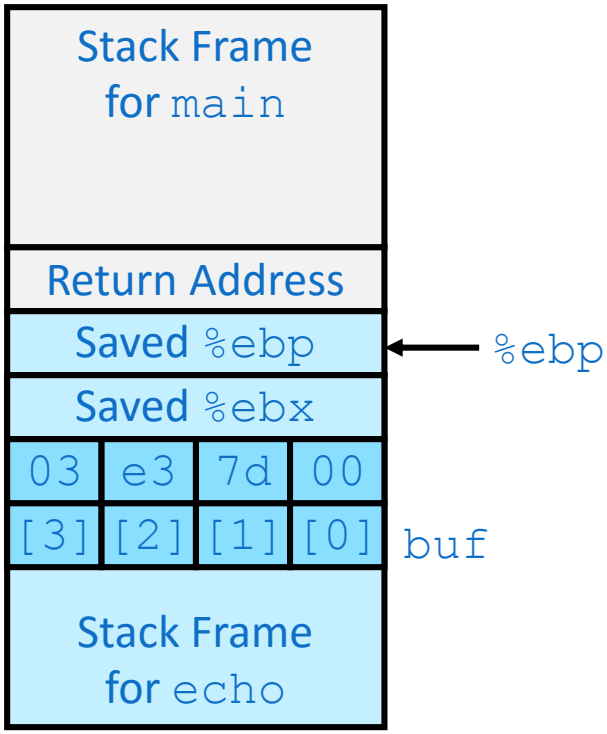
%ebp

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

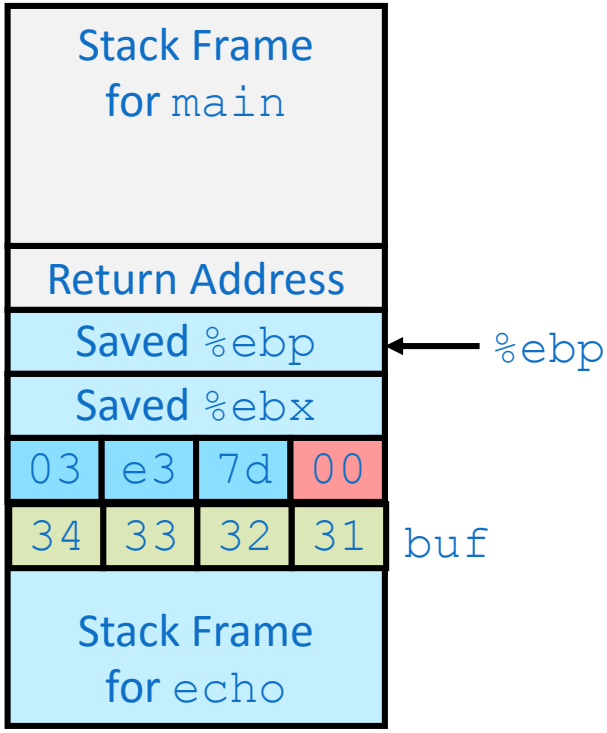
```
echo:
    . . .
    movl    -8(%ebp), %eax    # Retrieve from stack
    xorl    %gs:20, %eax      # Compare with Canary
    je      .L24              # Same: skip ahead
    call    __stack_chk_fail  # ERROR
.L24:
    . . .
```

“Canary”示例

Before call to gets



Input 1234




```
(gdb) break echo
(gdb) run
(gdb) stepi 3
(gdb) print /x *((unsigned *) $ebp - 2)
$1 = 0x3e37d00
```

开始破坏，程序允许仅仅破坏Canary

蠕虫与病毒

- 蠕虫：一个程序
  - 能够自主运行
  - 能够自我复制到其它计算机中
- 病毒：一段代码
  - 添加在其它程序里
  - 不能独自运行
- 都能在计算机之间进行扩散并造成严重破坏

X86-64架构

- 背景
  - Intel最早推出的64位架构是基于超长指令字VLIW技术的IA-64体系结构，Intel 称其为显式并行指令计算机EPIC （ Explicitly Parallel Instruction Computer ）。安腾和安腾2分别在2000年和2002年问世，它们是IA-64体系结构的最早的具体实现。
  - AMD公司利用Intel在IA-64架构上的失败，抢先在2003年推出兼容IA-32的64位版本指令集x86-64，AMD获得了以前属于Intel的一些高端市场。AMD后来将x86-64更名为AMD64。
  - Intel在2004年推出IA32-EM64T，它支持x86-64指令集。Intel为了表示EM64T的64位模式特点，又使其与IA-64有所区别，2006年开始把EM64T改名为Intel 64。

X86-64架构

- 与IA-32相比，x86-64架构的主要特点
  - 新增8个64位通用寄存器：R8、R9、R10、R11、R12、R13、R14和R15。可作为8位（R8B~R15B）、16位（R8W~R15W）或32位寄存器（R8D~R15D）使用
  - 所有GPRs都从32位扩充到64位。8个32位通用寄存器EAX、EBX、ECX、EDX、EBP、ESP、ESI和 EDI对应扩展寄存器分别为RAX、RBX、RCX、RDX、RBP、RSP、RSI和RDI；EBP、ESP、ESI和 EDI的低8位寄存器分别是BPL、SPL、SIL和DIL
  - 字长从32位变为64位，故逻辑地址从32位变为64位
  - long double型数据虽还采用80位扩展精度格式，但所分配存储空间从12B扩展为16B，即改为16B对齐，但不管是分配12B还是16B，都只用到低10B
  - 过程调用时，通常用通用寄存器而不是栈来传递参数，因此，很多过程不用访问栈，这使得大多数情况下执行时间比IA-32代码更短
  - 128位的MMX寄存器从原来的8个增加到16个，浮点操作采用基于SSE的面向XMM寄存器的指令集，而不采用基于浮点寄存器栈的指令集

X86-64架构

- x86-64的基本指令和对齐
  - 数据传送指令（汇编指令中助记符“q”表示操作数长度为四字（即64位））
    - movabsq指令用于将一个64位立即数送到一个64位通用寄存器中；
    - movq指令用于传送一个64位的四字；
    - movsbq、movswq、movslq用于将源操作数进行符号扩展并传送到一个64位寄存器或存储单元中；
    - movzbq、movzwq用于将源操作数进行零扩展后传送到一个64位寄存器或存储单元中；
    - pushq和popq分别是四字压栈和四字出栈指令；
    - movl指令的功能相当于movzlq指令 。

X86-64架构

- 数据传送指令举例

以下函数功能是将类型为source_type的参数转换为dest_type型数据并返回

```
dest_type  convert(source_type x) {
        dest_type y = (dest_type) x;
        return y;
}
```

根据参数传递约定知，x在RDI对应的适合宽度的寄存器（RDI、EDI、DI和DIL）中，y存放在RAX对应的寄存器（RAX、EAX、AX或AL）中，填写下表中的汇编指令以实现convert函数中的赋值语句

source_type	dest_type	汇 编 指 令
char	long	问题：每种情况对应的汇编指令各是什么？
int	long	
long	long	
long	int	
unsigned int	unsigned long	
unsigned long	unsigned int	
unsigned char	unsigned long	

X86-64架构

- 数据传送指令举例

以下函数功能是将类型为source_type的参数转换为dest_type型数据并返回

```
dest_type  convert(source_type x) {
        dest_type y = (dest_type) x;
        return y;
}
```

根据参数传递约定知，x在RDI对应的适合宽度的寄存器（RDI、EDI、DI和DIL）中，y存放在RAX对应的寄存器（RAX、EAX、AX或AL）中，填写下表中的汇编指令以实现convert函数中的赋值语句

source_type	dest_type	汇 编 指 令
char	long	movsbq %dil, %rax
int	long	movslq %edi, %rax
long	long	movq %rdi, %rax
long	int	movslq %edi, %rax //符号扩展到 64 位 movl %edi, %eax // **只需x的低32位**
unsigned int	unsigned long	movl %edi, %eax //零扩展到 64 位
unsigned long	unsigned int	movl %edi, %eax //零扩展到 64 位
unsigned char	unsigned long	movzbq %dil, %rax //零扩展到 64 位

X86-64架构

- 算术逻辑运算指令
  - addq（四字相加）
  - subq（四字相减）
  - imulq（带符号整数四字相乘）
  - orq（64位相或）
  - leaq（有效地址加载到64位寄存器）

以下是C赋值语句“x=a*b+c*d;”对应的x86-64汇编代码，已知x、a、b、c和d分别在寄存器RAX(x)、RDI(a)、RSI(b)、RDX(c)和RCX(d)对应宽度的寄存器中。根据以下汇编代码，推测x、a、b、c和d的数据类型

```
movslq %ecx, %rcx
imulq  %rdx, %rcx
movsbl %sil, %esi
imull %edi, %esi
movslq %esi, %rsi
leaq (%rcx, %rsi), %rax
```

d从32位符号扩展为64位，故d为int型

在RDX中的c为64位long型

在SIL中的b为char型

在EDI中的a是int型

在RAX中的x是long型

X86-64架构

- 过程调用的参数传递
  - 通过寄存器传送参数
  - 最多可有6个整型或指针型参数通过寄存器传递
  - 超过6个入口参数时，后面的通过栈来传递
  - 在栈中传递的参数若是基本类型，则都被分配8个字节
  - call（或callq）将64位返址保存在栈中之前，执行R[rsp]←R[rsp]-8
  - ret从栈中取出64位返回地址后，执行R[rsp]←R[rsp]+8

操作数宽度（字节）	入口参数						返回参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

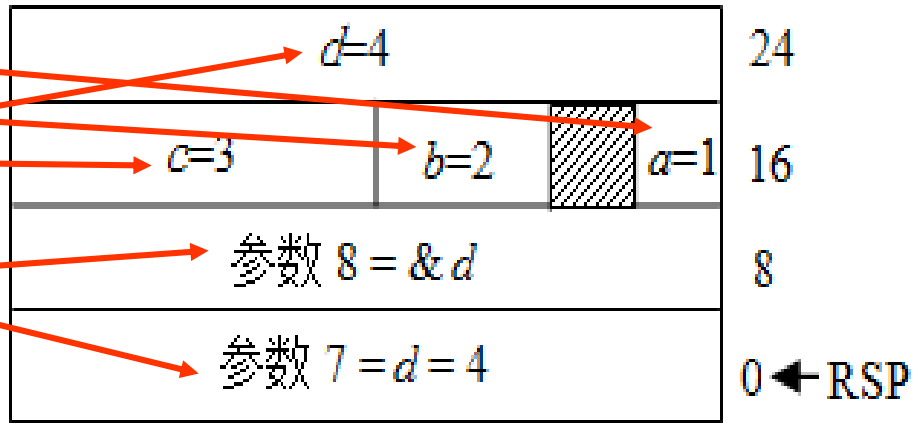
X86-64架构过程调用举例

```
long caller ( )
{
    char a=1 ;
    short b=2 ;
    int c=3 ;
    long d=4 ;
    test(a, &a, b, &b, c, &c, d, &d);
    return  a*b+c*d;
}
```

其他6个参数在哪里？

```
void test(char a, char *ap,
          short b, short *bp,
          int c, int *cp,
          long d, long *dp)
{
        *ap+=a;
        *bp+=b;
        *cp+=c;
        *dp+=d;
}
```

执行到 caller 的 call 指令时栈中情况



执行到caller的call指令前，栈中的状态如何？

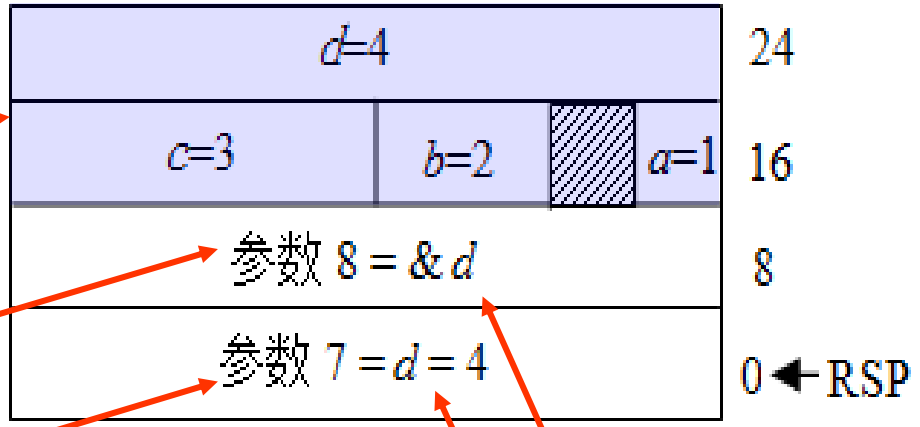
操作数宽度（字节）	入口参数						返回参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

X86-64架构过程调用举例

```
subq $32, %rsp          //R[rsp]←R[rsp]-32
movb $1, 16(%rsp)       //M[R[rsp]+16]←1
movw $2, 18(%rsp)       //M[R[rsp]+18]←2
movl $3, 20(%rsp)       //M[R[rsp]+20]←3
movq $4, 24(%rsp)       //M[R[rsp]+24]←4
leaq 24(%rsp), %rax     //R[rax]←R[rsp]+24
movq %rax, 8(%rsp)      //M[R[rsp]+8]←R[rax]
movq $4, (%rsp)         //M[R[rsp]]←4
leaq 20(%rsp), %r9      //R[r9]←R[rsp]+20
movl $3, %r8d           //R[r8d]←3
leaq 18(%rsp), %rcx     //R[rcx]←R[rsp]+18
movw $2, %dx            //R[dx]←2
leaq 16(%rsp), %rsi     //R[rsi]←R[rsp]+16
movb $1, %dil           //R[dil]←1
call test
```

第15条指令

执行到 caller 的 call 指令时栈中情况




```
long caller ( )
{
    char a=1 ;
    short b=2 ;
    int c=3 ;
    long d=4 ;
    test(a, &a, b, &b, c, &c, d, &d);
    return  a*b+c*d;
}
```

X86-64架构过程调用举例

```
movq 16(%rsp), %r10  //R[r10]←M[R[rsp]+16]
addb %dil, (%rsi)    //M[R[rsi]]←M[R[rsi]]+R[dil]
addw %dx, (%rcx)     //M[R[rcx]]←M[R[rcx]]+R[dx]
addl %r8d, (%r9)     //M[R[r9]]←M[R[r9]]+R[r8d]
movq 8(%rsp), %rax   //R[rax]←M[R[rsp]+8]
addq %rax, (%r10)    //M[R[r10]]←M[R[r10]]+R[rax]
ret
```

R[r10]←&d

*ap+=a;

*bp+=b;

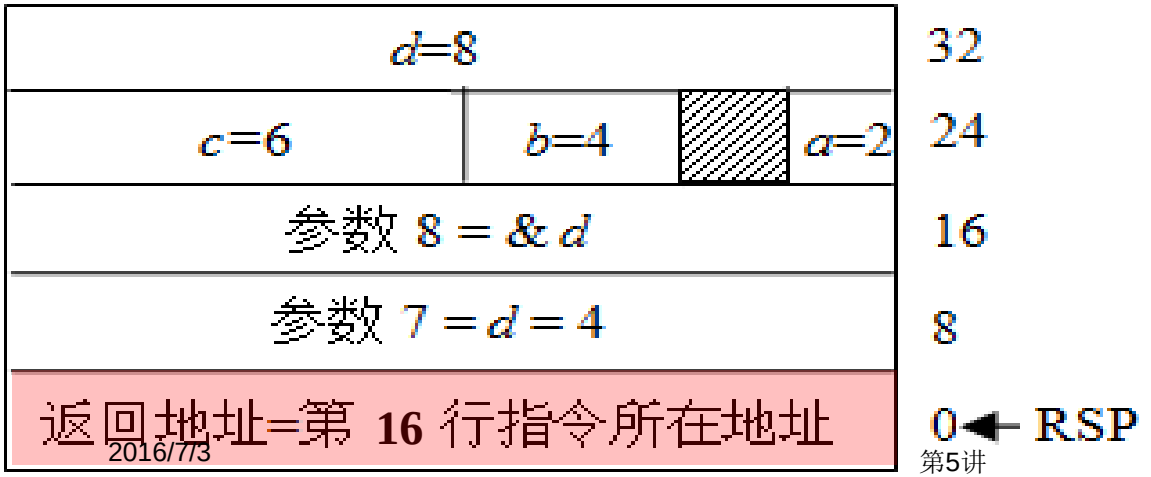
*cp+=c;

*dp+=d;

DIL、RSI、DX、RCX、R8D、R9

```
void test(char a, char *ap,
          short b, short *bp,
          int c, int *cp,
          long d, long *dp)
{
    *ap+=a;
    *bp+=b;
    *cp+=c;
    *dp+=d;
}
```

执行到test的ret指令前，栈中的状态如何？ret执行后怎样？



X86-64架构过程调用举例

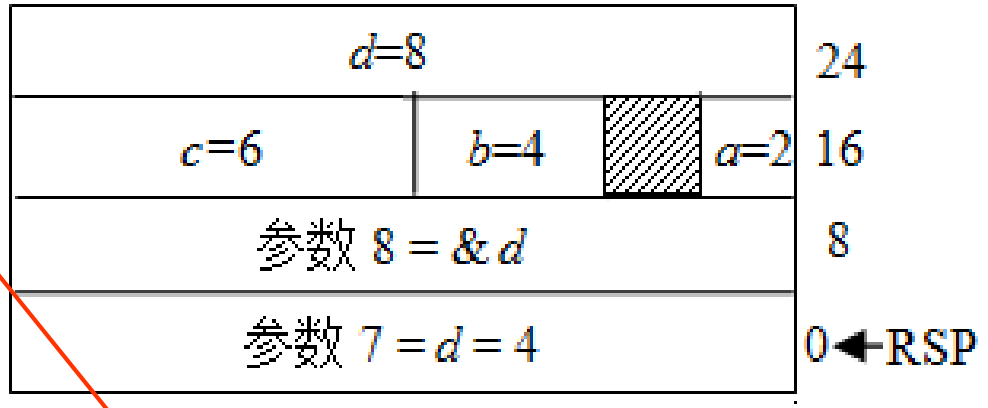
从第16条指令开始

```
movslq 20(%rsp), %rcx
movq   24(%rsp), %rdx
imulq  %rdx, %rcx
movsbw 16(%rsp), %ax
movw   18(%rsp), %dx
imulw  %dx, %ax
movswq %ax, %rax
leaq   (%rax, %rcx),%rax
addq   $32, %rsp
ret
```

执行到ret指令时，
RSP指向调用caller
函数时保存的返回值

释放caller的栈帧

执行test的ret指令后，栈中的状态如何？




```
long caller ( )
{
    char a=1 ;
    short b=2 ;
    int c=3 ;
    long d=4 ;
    test(a, &a, b, &b, c, &c, d, &d);
    return  a*b+c*d;
}
```

浮点操作与SIMD指令

- IA-32的浮点处理架构有两种：
  - x86配套的浮点协处理器x87FPU架构，80位浮点寄存器栈
  - 由MMX发展而来的SSE指令集架构，采用的是单指令多数据（Single Instruction Multi Data，SIMD）技术
- 对于IA-32架构， gcc默认生成x87 FPU 指令集代码
  - 如果想要生成SEE指令集代码，则需要设置适当的编译选项
- 在x86-64中，浮点运算采用SIMD指令
  - 浮点数存放在128位的XMM寄存器中

IA-32和x86-64的比较

例：以下是一段C语言代码：

```
#include <stdio.h>
int main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

在IA-32上运行时，打印结果为a=0

在x86-64上运行时，打印一个不确定值

为什么？

$10=1010B=1.01\times2^3$

阶码e=1023+3=10000000010B

10的double型表示为：

0 10000000010 0100…0B

即4024 0000 0000 0000H

先执行fldl，再执行fstpl

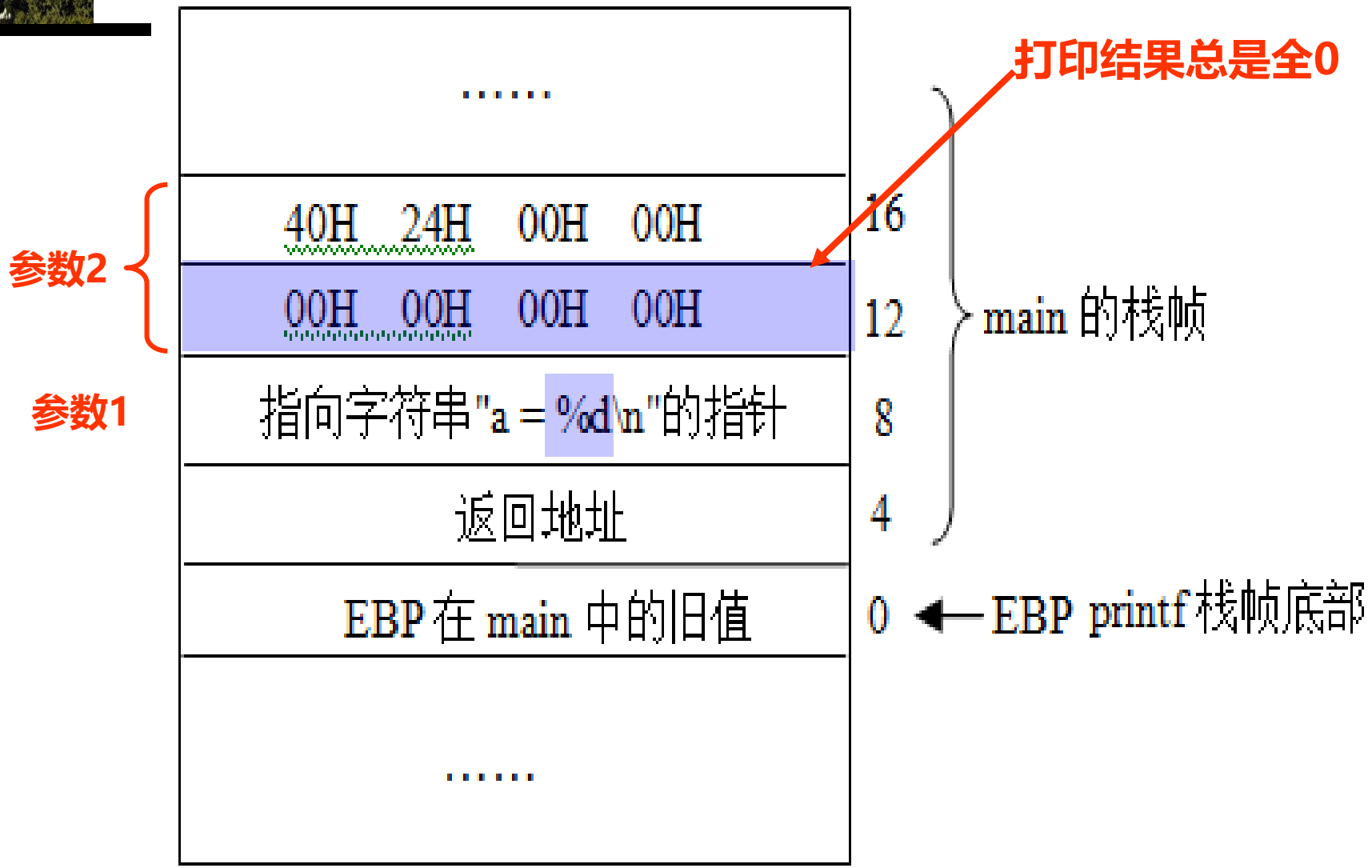
fldl：局部变量区→ST(0)

fstpl：ST(0) →参数区

在IA-32中a为float型又怎样呢？先执行flds，再执行fstpl

即：flds将32位单精度转换为80位格式入浮点寄存器栈，fstpl再将80位转换为64位送存储器栈中，故实际上与a是double效果一样！

IA-32过程调用参数传递



a的机器数对应十六进制为：40 24 00 00 00 00 00 00H

X64参数传递（Linux系统）

```
int main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

操作数宽度（字节）	入口参数						返回参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

```
.LC1:
        .string "a = %d\n "
......
movsd   .LC0(%rip), %xmm0 //a送xmm0
movl    $.LC1, %edi //RDI 高32位为0
movl    $1, %eax    //向量寄存器个数
call     printf
addq    $8, %rsp
ret
......
.LC0:
    .long   0
    .long   1076101120
```

00000000H

40240000H

因为printf第2个参数为double型，故向量寄存器个数为1

小端方式！0存在低地址上

printf中为%d，故将从ESI中取打印参数进行处理；但a是double型数据，在x86-64中，a的值被送到XMM寄存器中而不会送到ESI中。故在printf执行时，从ESI中读取的并不是a的低32位，而是一个不确定的值。

X64参数传递（Win64）

```
int main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

前4个参数分别通过

RCX、RDX、R8、R9传递

```
sub     $0x28, %rsp
callq   0x402269 <_main>
movsd   .LC0(%rip), %xmm0 //a送xmm0
movapd  %xmm0, %xmm1
movq    %xmm0, %rdx  //rdx低32位为0
Lea     0x2aef(%rip), %rcx //字符串首址
callq   0x402b18 <printf>
add     $0x28, %rsp
retq
......
.LC0:
    .long   0           ← 00000000H
    .long   1076101120  ← 40240000H
```

小端方式！0存在低地址上

printf中为%d，故将从RDX中低32位取打印参数进行处理；其中是double型数据10的低32位，因此为全0，故最后打印结果为0。

X86-64架构

- 数据的对齐
  - x86-64中各类型数据遵循一定的对齐规则，而且更严格
  - x86-64中存储器访问接口被设计成按8字节或16字节为单位进行存取，其对齐规则是，任何K字节宽的基本数据类型和指针类型数据的起始地址一定是K的倍数。
    - short型数据必须按2字节边界对齐
    - int、float等类型数据必须按4字节边界对齐
    - long型、double型、指针型变量必须按8字节边界对齐
    - long double型数据必须按16字节边界对齐

作业

- 练习：3.43，3.44，3.45，3.47，3.48