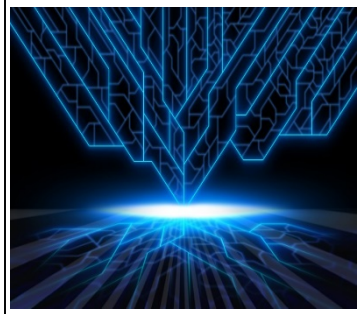
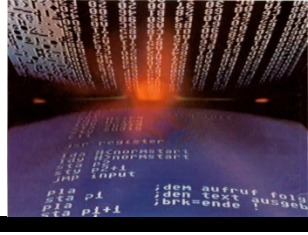
第12讲
内存访问



吴海军

南京大学计算机科学与技术系

主要内容

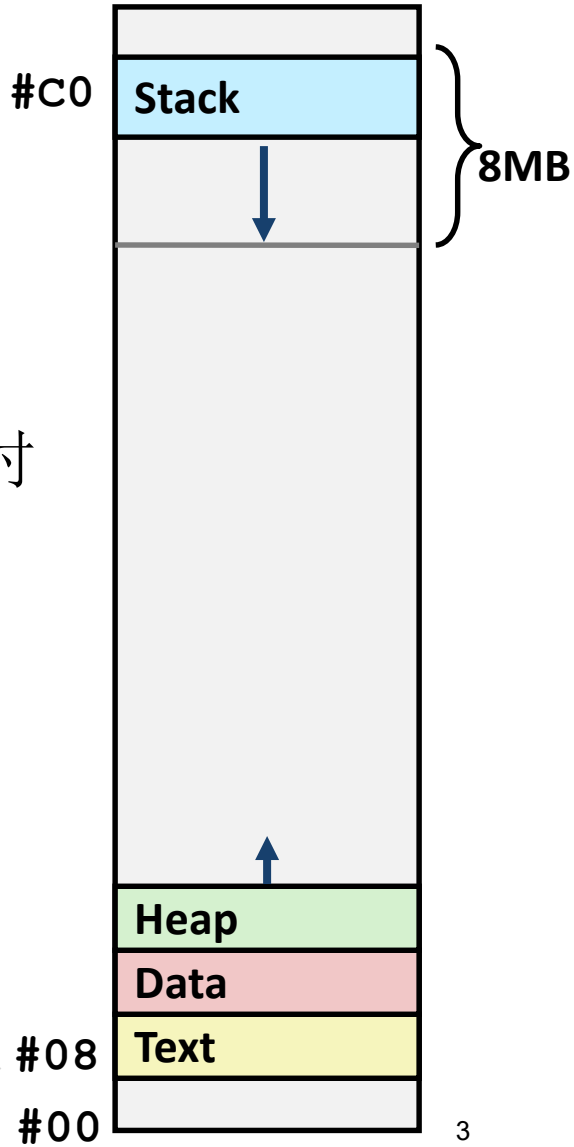


- 内存分配
- 越界访问
- 缓冲区溢出

Linux 内存布局

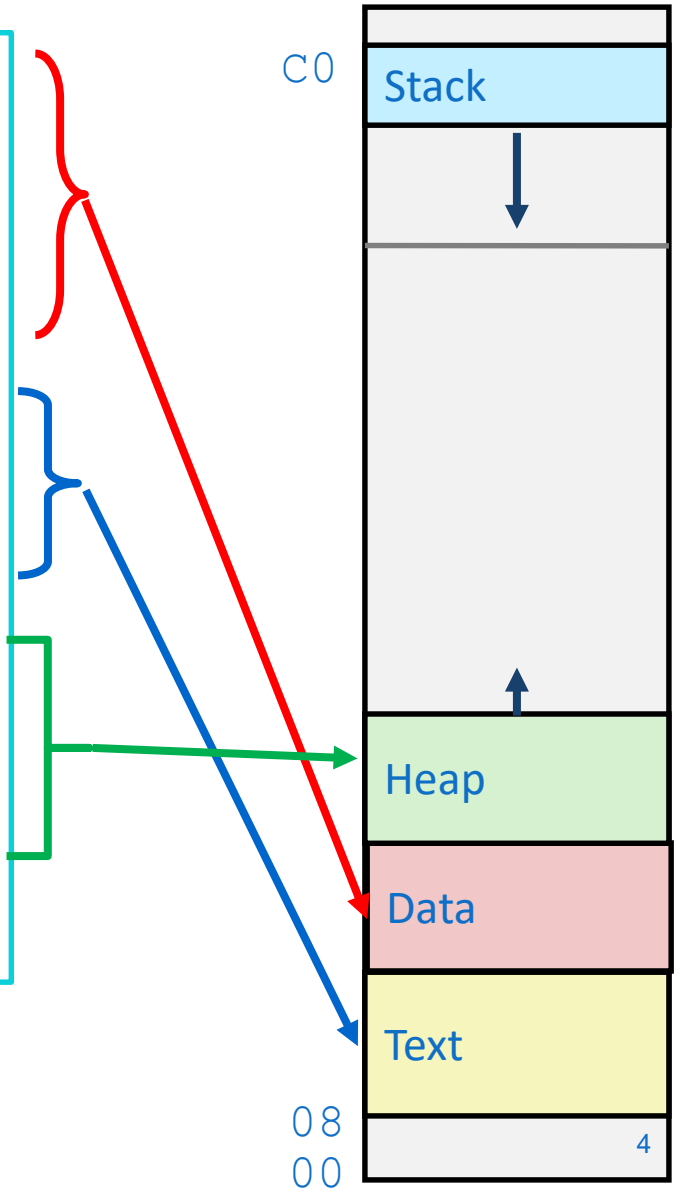
非比例设置

- 栈(stack)
  - 运行时栈 (8MB限制)
- 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

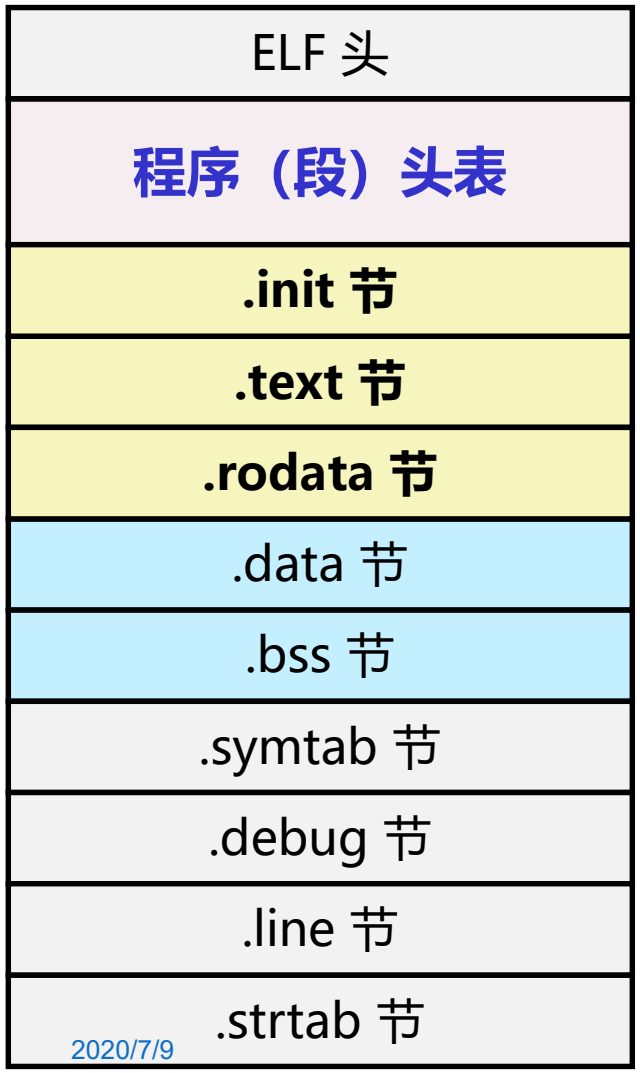
```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

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 ... */
}
```



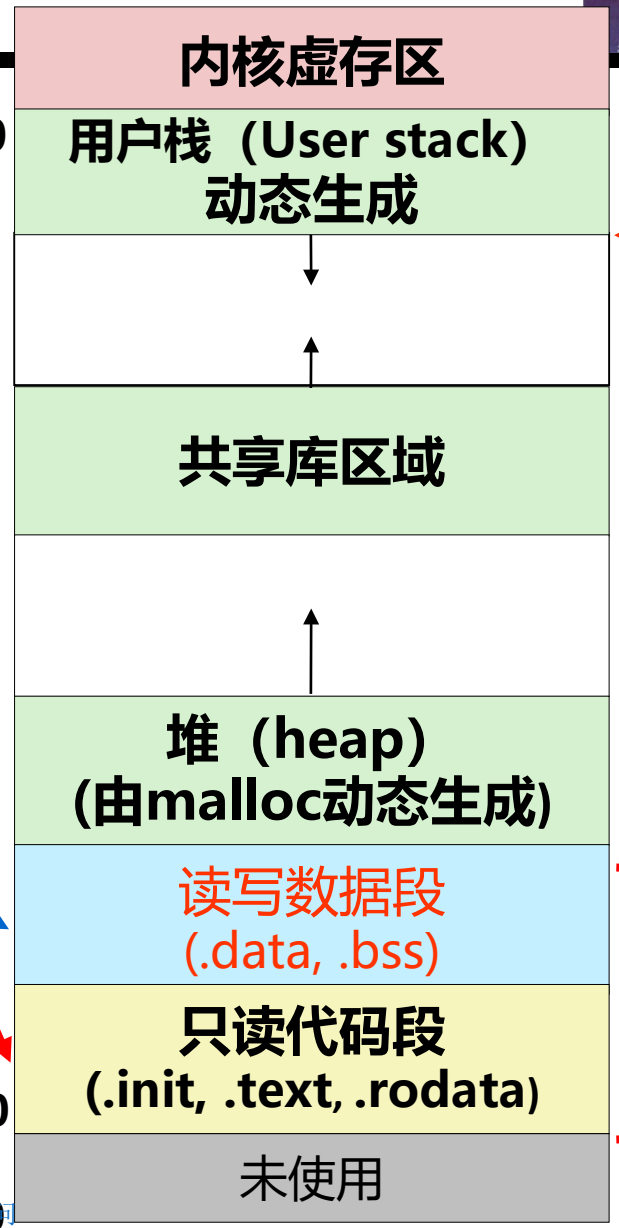
malloc()需要动态链接库，
地址在运行时确定

ELF可执行文件的存储器映像

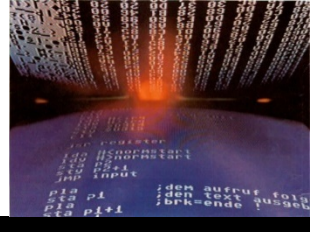
程序(段)头表描述如何映射



从高地址向低地址增长!



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];
}
```

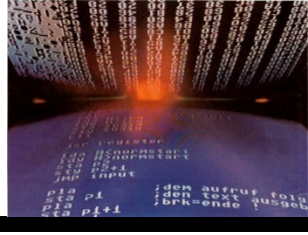
fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.0000061035156
fun(4) → 3.14, 然后存储保护错

为什么当 i>1 就有问题?

因为数组访问越界!

Saved State	4
d7 ... d4	3
d3 ... d0	2
a[1]	1
a[0]	0

越界访问和缓冲区溢出



- C语言中对数组的引用不做边界检查，而且局部变量和状态信息都保存在栈中。
- 对越界数组的写操作会破坏保存在栈中的状态信息，使用被破的状态数据可能导致严重错误。
- 一种常见的破坏状态称为缓冲区溢出(buffer overflow)。
- 缓冲区溢出在各种操作系统、应用软件中广泛存在。
- 利用缓冲区溢出漏洞所进行的攻击行动，可导致程序运行失败、系统关机、重新启动等后果。

字符串库函数

- Unix中gets()函数的实现

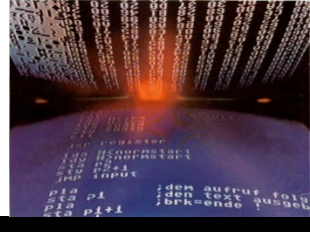
```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getc();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getc();
    }
    *p = '\0';
    return dest;
}
```

当接收到EOF或回车符号时，停止。

没有办法限制要读入的字符串长度。

- 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[8];  /* Way too small!
*/
    gets(buf);
    puts(buf);
}
```

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

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

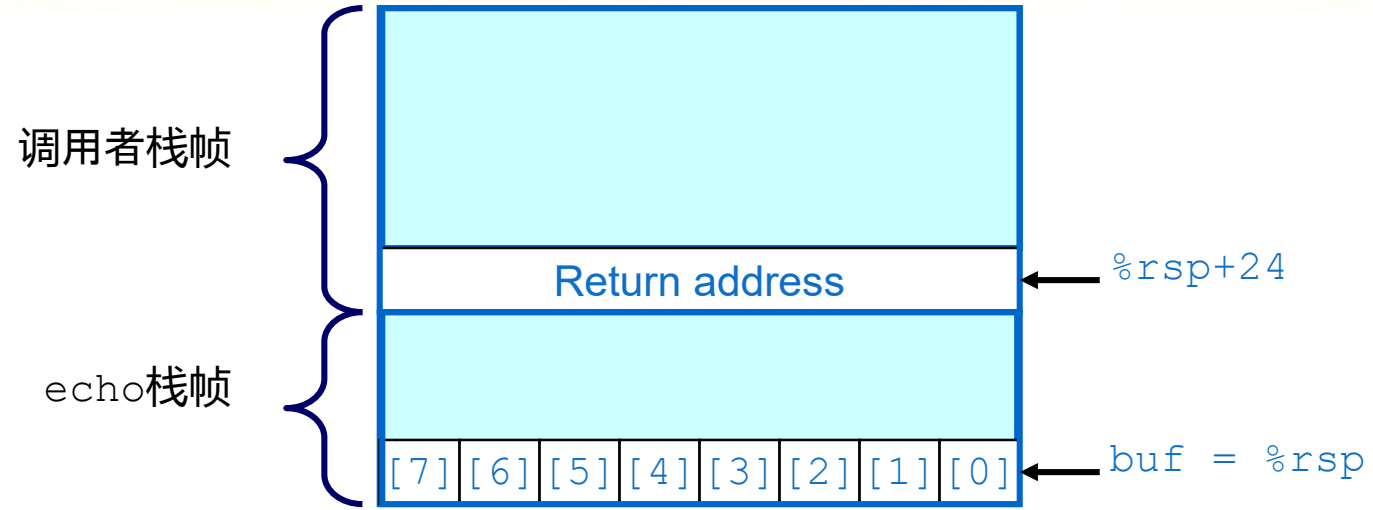
```
unix>./bufdemo
Type a string:12345678
Segmentation Fault
```

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

缓冲区溢出反汇编

echo:

```
     void echo()
1    echo:
2      subq    $24, %rsp       Allocate 24 bytes on stack
3      movq    %rsp, %rdi      Compute buf as %rsp
4      call    gets            Call gets
5      movq    %rsp, %rdi      Compute buf as %rsp
6      call    puts            Call puts
7      addq    $24, %rsp       Deallocate stack space
8      ret                     Return
```

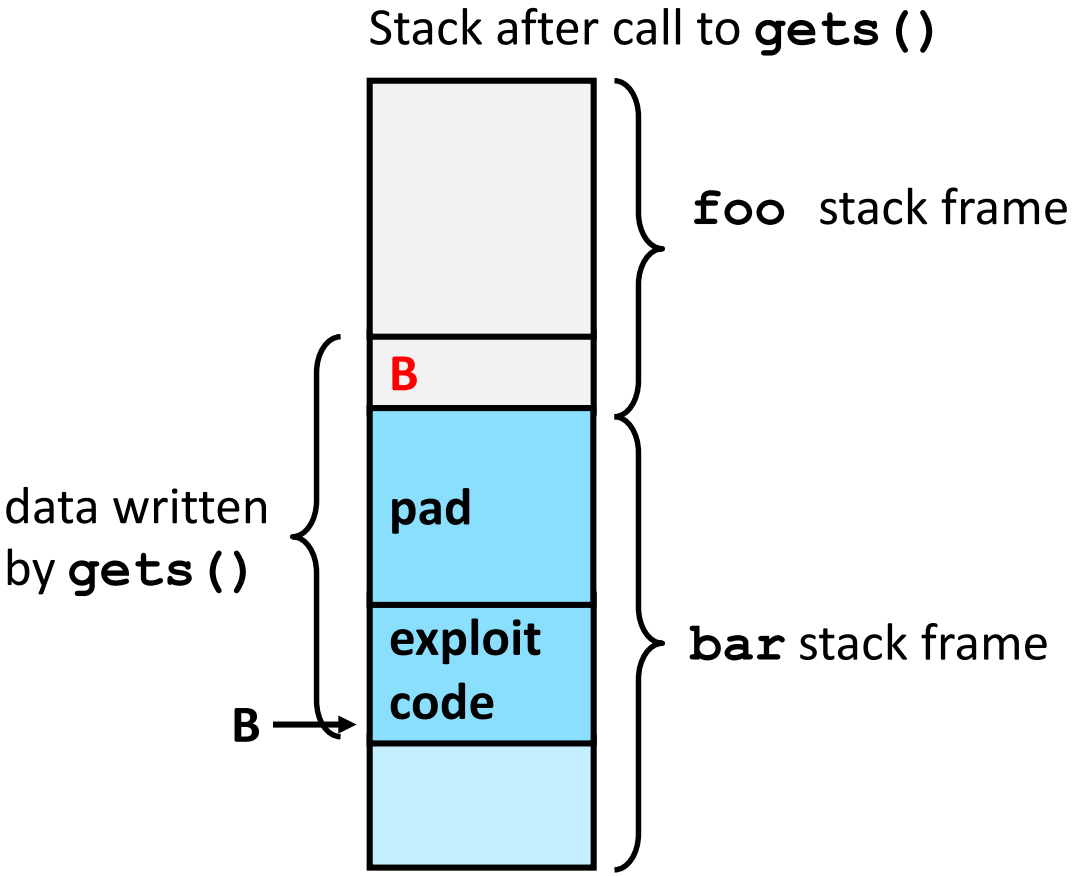


恶意利用缓冲区溢出

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

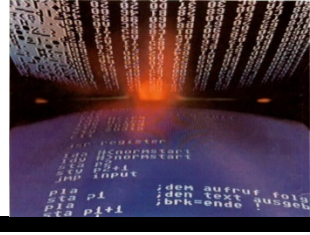
Return address A

```
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

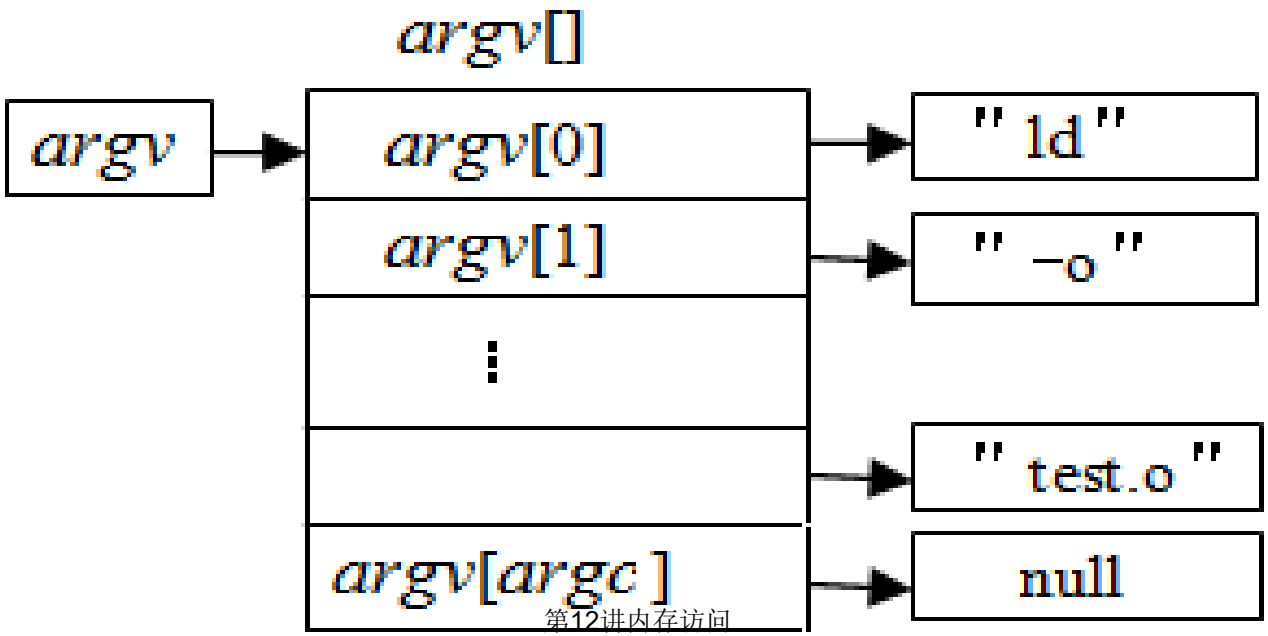
main()函数的原型

- 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



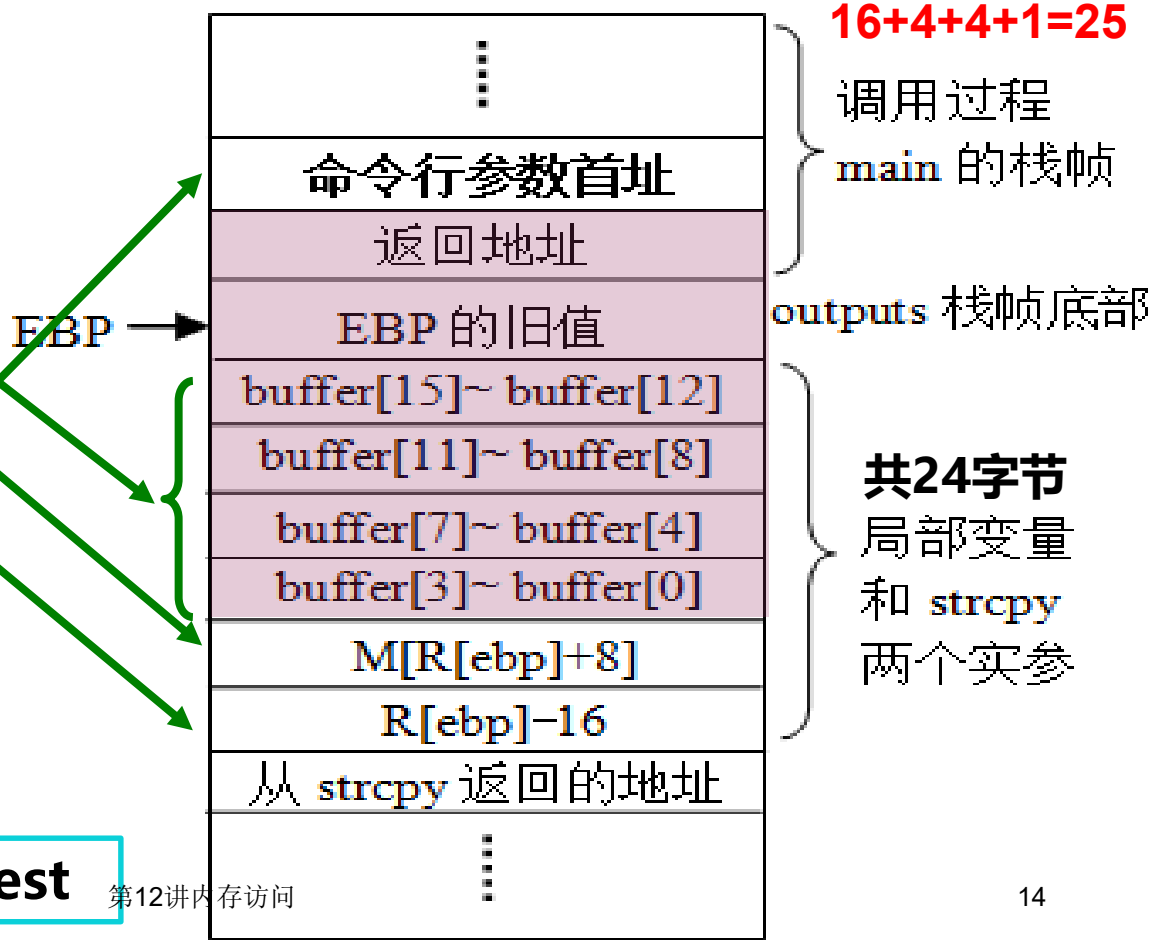
越界访问和缓冲区溢出

- 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;
}
```

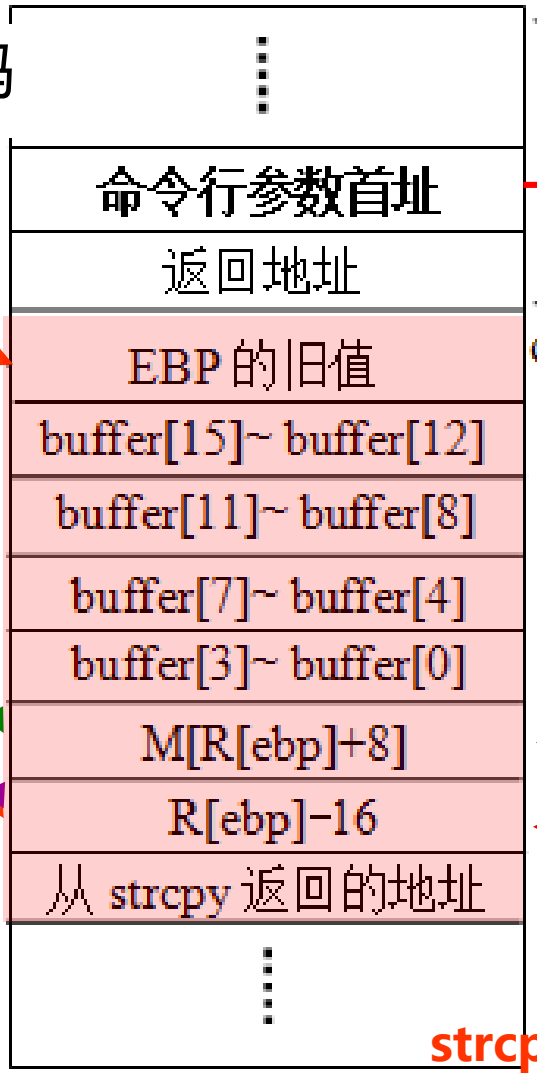


假定可执行文件名为test

越界访问和缓冲区溢出

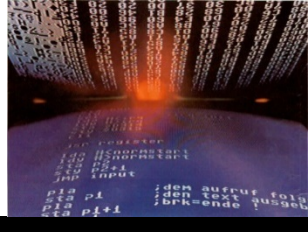
test被反汇编得到的outputs汇编代码

```
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

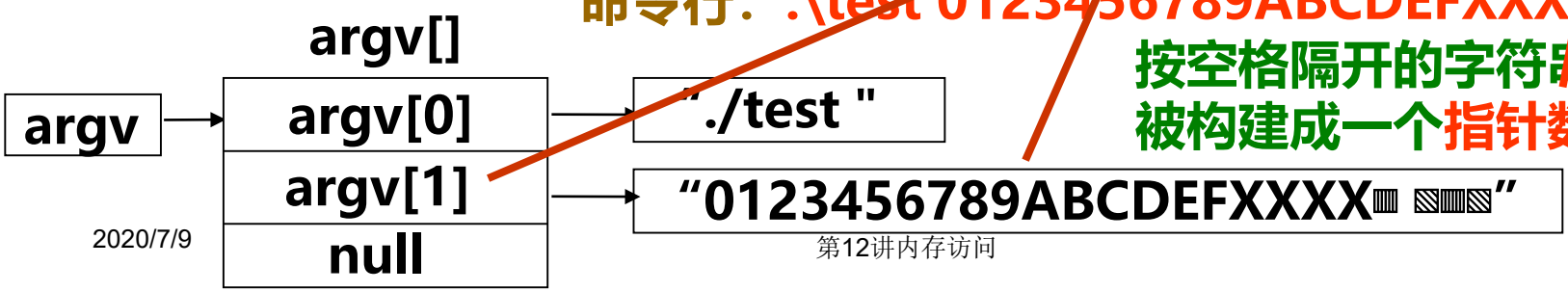
```
#include "stdio.h"
char code[]=
    "0123456789ABCDEFXXXX"
    "\x11\x84\x04\x08"
    "\x00";
int main(void)
{
    char *argv[3];
    argv[0]="./test";
    argv[1]=code;
    argv[2]=NULL;
    execve(argv[0],argv,NULL);
    return 0;
}
```

```
#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▥ ▧▥▧

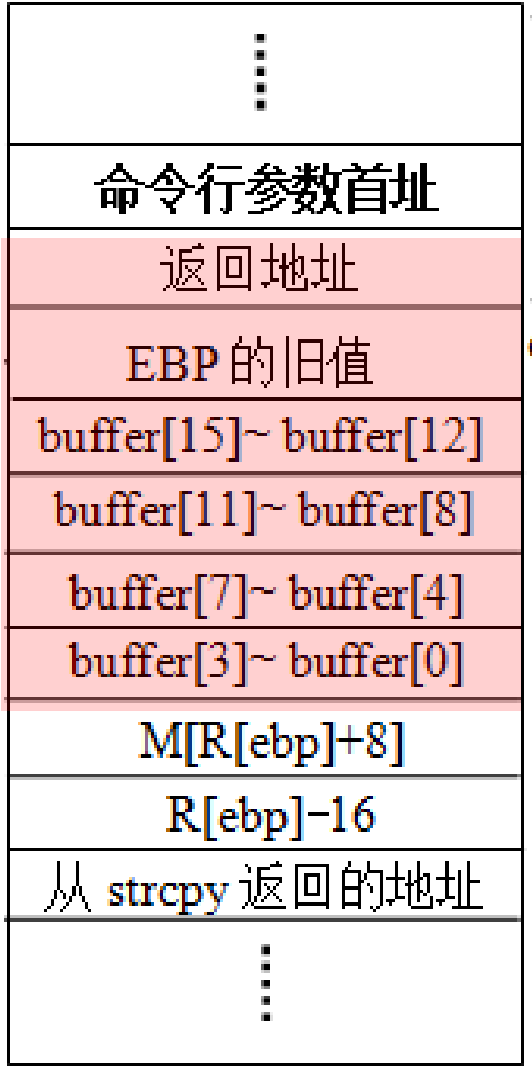
按空格隔开的字符串
被构建成一个指针数组



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：

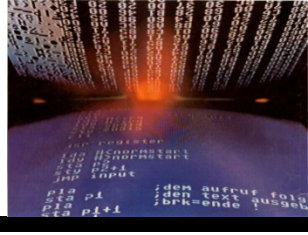
"0123456789ABCDEFXXXX

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

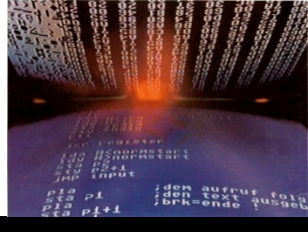


- 两个方面的防范
  - 从程序员角度去防范
    - 用辅助工具帮助程序员查漏，例如，用grep来搜索源代码中容易产生漏洞的库函数（如strcpy和sprintf等）的调用；用fault injection查错
  - 从编译器和操作系统方面去防范
    - 地址空间随机化ASLR

      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

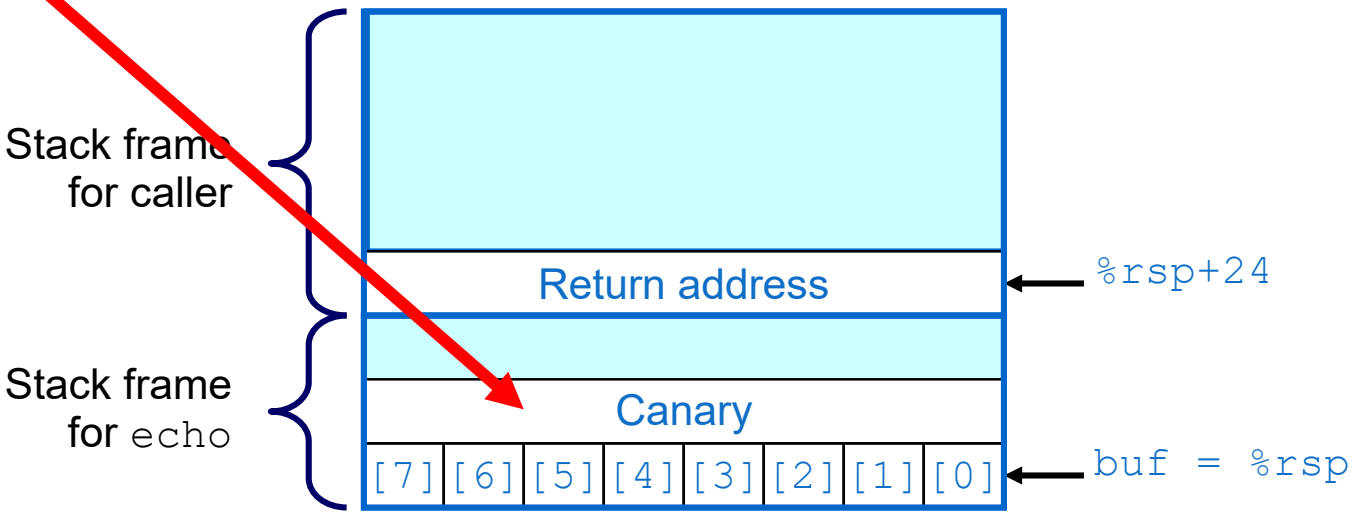
缓冲溢出攻击防范



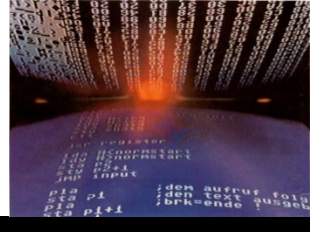
- 地址空间随机化
  - 只要操作系统相同，则栈位置就一样，若攻击者知道漏洞程序使用的栈地址空间，就可设计一个针对性攻击，在使用该程序机器上实施攻击。
  - 地址空间随机化（栈随机化）的基本思路是，将加载程序时生成的代码段、静态数据段、堆区、动态库和栈区各部分的首地址进行随机化处理，使每次启动时，程序各段被加载到不同地址起始处。
  - 对于随机生成的栈起始地址，攻击者不太容易确定栈的起始位置

缓冲区溢出攻击的防范

- 栈破坏检测
  - 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，在其栈帧中任何局部缓冲区底部与栈状态之间存储一个随机生成的特殊值canary；每次运行前，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



栈破坏检测



- 实例

```
    void echo()
1   echo:
2     subq    $24, %rsp
3     movq    %fs:40, %rax
4     movq    %rax, 8(%rsp)
5     xorl    %eax, %eax
6     movq    %rsp, %rdi
7     call    gets
8     movq    %rsp, %rdi
9     call    puts
10    movq    8(%rsp), %rax
11    xorq    %fs:40, %rax
12    je      .L9
13    call    __stack_chk_fail
14  .L9:
15    addq    $24, %rsp
16    ret
```

定义栈帧

从段地址中获取随机数

存储到栈帧中

保存缓冲区地址

保存缓冲区地址

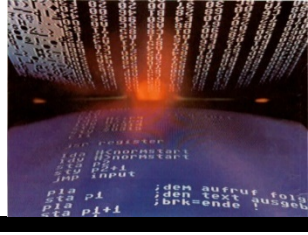
从栈帧中读取特殊值

与原形比对

检测失败，异常处理

释放栈帧

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。