**电子科技大学信息与软件工程学院**

**标 准 实 验 报 告**

**（实验）课程名称 网络安全技术**

**电子科技大学教务处制表**

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名：周玉川 学 号：2017221302006 指导教师：赵洋**

**实验地点：信软楼西303 实验时间：2019.11.19**

**一、实验室名称：实验中心**

**二、实验项目名称：缓冲区溢出实验**

**三、实验学时： 4 学时**

**（一）实验目的**

了解缓冲区溢出的相关概念；明确缓冲区溢出的危害；理解栈溢出、堆溢出、整型溢出、格式化字符串溢出及文件流溢出的原因；掌握安全编程技术。C/C++之类的高级语言通过指针直接对内存地址中所蕴藏的数据进行修改，因此可以利用C语言修改堆栈内容而修改程序的执行方式。本实验通过对缓冲区溢出利用的实践，理解缓冲区溢出的利用方法和带来的危害后果。

**（二）实验内容**

1. 初步认识栈溢出实验
2. 利用栈溢出漏洞实验
3. 溢出编程实验

**四、实验原理**

缓冲区溢出是目前最常见的一种安全问题，操作系统以及应用程序大都存在缓冲区溢出漏洞。缓冲区是一段连续内存空间，具有固定的长度。缓冲区溢出是由编程错误引起的，当程序向缓冲区内写入的数据超过了缓冲区的容量，就发生了缓冲区溢出，缓冲区之外的内存单元被程序“非法”修改。

一般情况下，缓冲区溢出导致应用程序的错误或者运行中止，但是，攻击者利用程序中的漏洞，精心设计出一段入侵程序代码，覆盖缓冲区之外的内存单元，这些程序代码就可以被CPU所执行，从而获取系统的控制权。

目前，操作系统（Windows、Linux、Unix）、数据库以及应用软件主要采用C/C++语言开发，但C/C++语言缺乏数组边界条件检查、程序执行不受控制等特点，因此，这些软件不可避免地存在缓冲区溢出漏洞，成为安全隐患。

**五、实验器材（设备、元器件）**

1. 实验人数50～80人，每人1台计算机；独立完成本实验。
2. 拓扑：（A、B范围中的主机分别简称为A主机和B主机）



**A**

**B**

1. 设备：以太网交换机2～4台；计算机50～80台
2. 软件：Visual Studio软件

**六、实验步骤**

1. **栈溢出的跟踪与解析实验**
2. 使用VS打开stack.dsw文件。

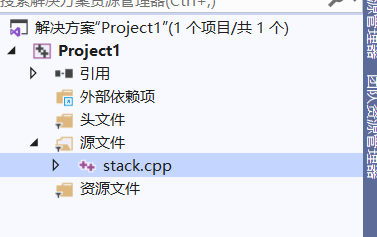
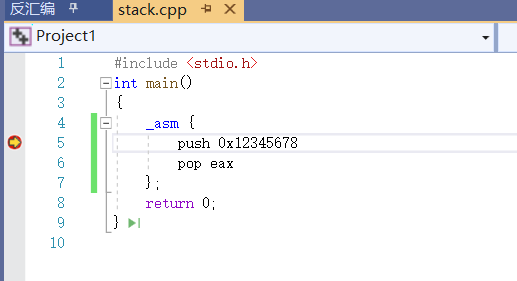
如图1-1

图1-1打开源代码

1. 在push语句前设置断点，按F5键进入调试界面。

图2-1设置断点，打开调试

图2-1

1. 打开Watch、Variables、Registers、Memory、Disassembly五个窗口。

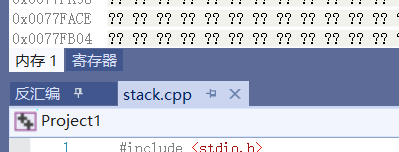
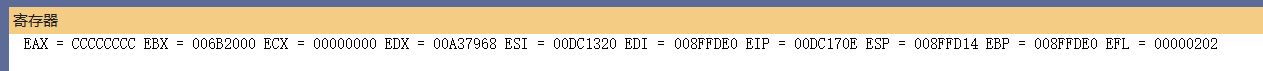
图3-1打开，内存，寄存器，临时变量，汇编窗口

图3-1

1. 记录EAX、ESP、EBP、EIP的值，并计算栈的大小为多少字节？

起初寄存器的值为



EAX=CCCCCCCC、ESP=008FFD14、EBP=008FFDE0、EIP=00DC170E

栈大小为EBP-EIP=CB

1. 按F11键执行push语句，观察栈、 ESP和EIP的值是否变化？为什么？



ESP=008FFD10，因为数据入栈减小了4

EIP=00DC1713,程序向下执行指令地址发生变化。

栈空间多了4字节为D0。

1. 继续执行pop语句，观察EAX、ESP、EIP的值是否变化？为什么？

程序运行后



EAX=12345678,因为栈中数据弹到eax中。

ESP=008FFD14,出栈ESP加4。

EIP继续增加。

1. 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。

发现函数返回值放在了eax中。

1. 使用VS打开hanshu.dsw文件。

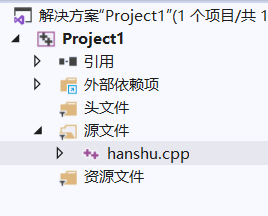
如图8-1

图8-1导入hanshu.cpp

1. 在ourfunction语句前设置断点，按F5键进入调试界面。

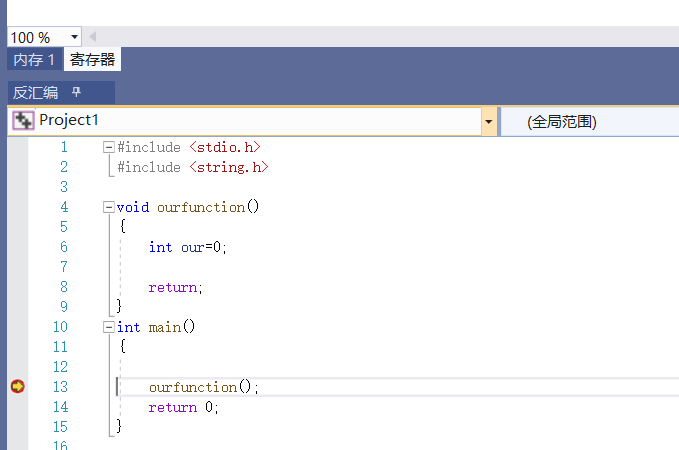
结果如图9-1

图9-1

1. 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？

ESP=007DF798,EBP=007DF868,EIP=002E16F0;

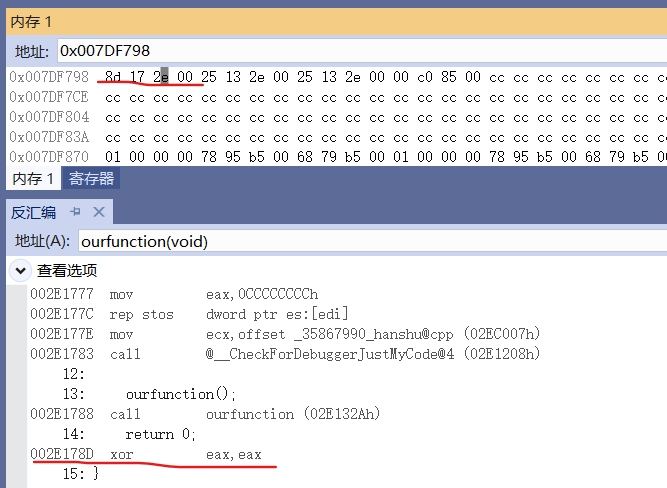
由图10-1可以看书esp指向的是函数ourfunction下一条指令return的地址。

图10-1

1. 继续执行，直到“int our=0”语句，观察每一步寄存器值的改变。

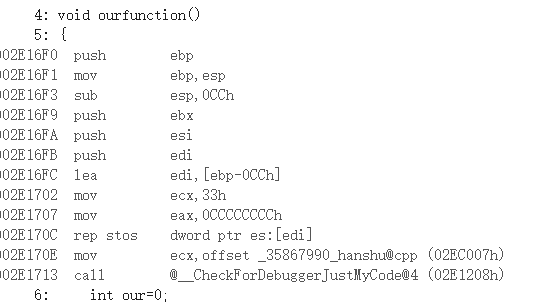
ebp改变一次，eip一直在变大，esp变小，根据图11-1可得出原因

图11-1

1. 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？

cur作为函数的局部变量，被放在ebp上方不远处。如图12-1所示

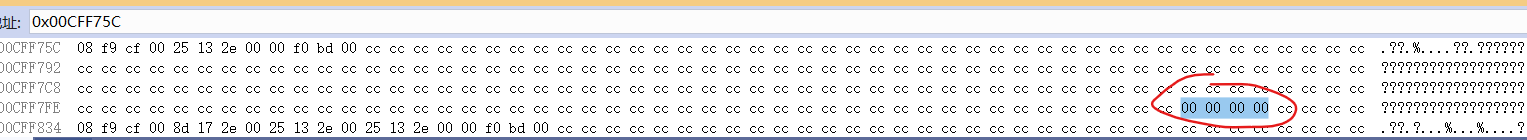


图12-1

1. 继续执行，直到“ret”语句，记录ESP，EBP的值。

ESP和EBP会恢复到执行函数前的值，因为执行函数时保存了ebp值，函数结束后恢复了ebp值。esp也是同样的道理。

1. “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？

程序会跳转到主函数的return语句，也就是ourfunction()的下一条指令处。ESP与EBP和步骤10的值相同，因为函数调用结束后，esp会栈平衡，ebp会恢复原来的值。

1. 按shift+F5结束调试。

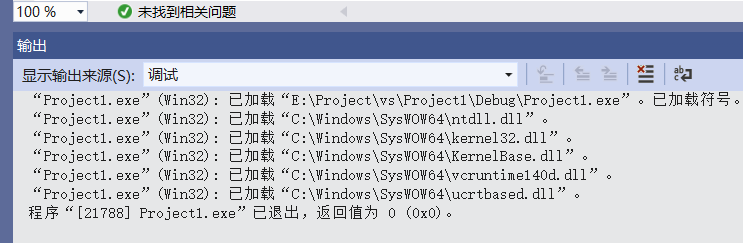
如图15-1所示

图15-1

1. 使用VS打开overflow.dsw文件。

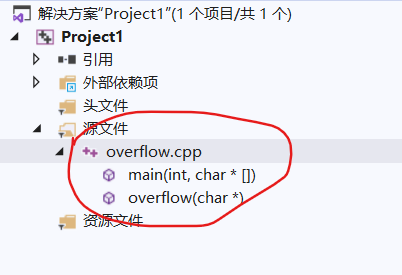
如图16-1所示

图16-1

1. 在“char longbuf[100]=……”语句前设置断点，F5调试，F11执行。

设置断点后按F5调试

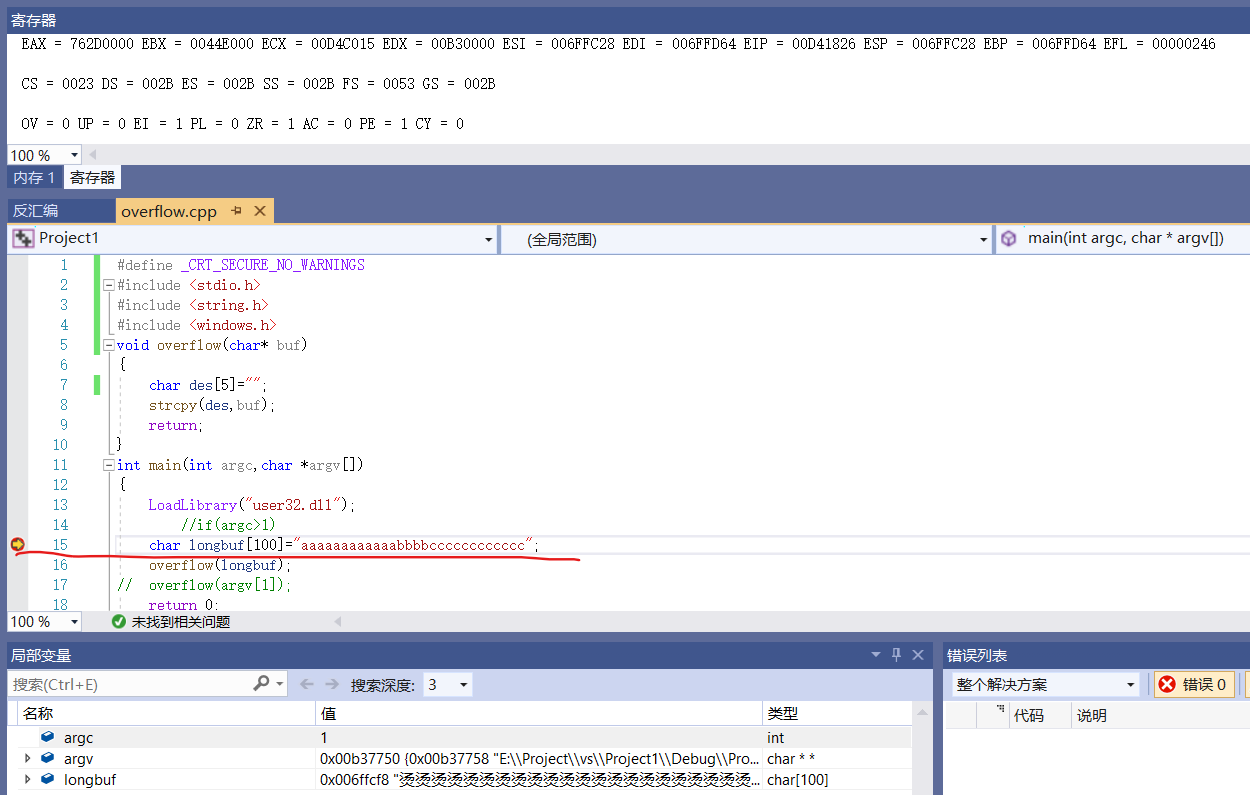
如图17-1所示

图17-1

1. 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？

由图17-1可知，EAX=762D0000,EBP=006FFD64，ESP=006FFC28。longbuf的存放地址是006FFCF8，计算其与EBP相差字节数006FFD64-006FFCF8=6C个字节。

1. 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？

EAX=006FFCF8,EAX值将会入栈，对EAX没有影响。EBP和EAX不变，ESP由于数据入栈，减少4.

1. 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？

lea eax,[ebp-64h]是将栈基地址ebp向上64h个字节的实际地址赋给EAX寄存器。

1. 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？

前者mov edx,dword ptr[ebp+8]是将栈基地址向下8个字节，也就是传入函数的参数值赋值给edx，而后者是将函数的局部变量（ebp向上8个字节）的地址赋值给eax。

1. 继续执行到“ret”语句，查看将弹出什么值到EIP中？

执行到ret时，栈顶数据为一个地址，经对比发现是main函数中调用overflow函数的下一条指令的地址，即add esp，4的地址。

1. 利用栈溢出漏洞实验
2. 使用VS打开liyong.dsw文件。
3. 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。

 此时EBP=0019FEF0,ESP=0019FE40。执行完赋值语句后查看longbuf的存放起始地址为0019FE0Bh和结束地址为0019FEEEh。记录EBP为0019FEF0、ESP为0019FE40。见图2-2-1

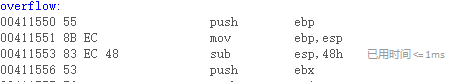
图2-2-1

1. 继续执行，“lea eax,[ebp-5fh]；push ecx”是将什么地址压栈？

 “call @ILT+1175(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004116E9，存放在0019FE36处。如图 2-3-1

图2-3-1

1. “call @ILT+1175(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？

 调用overflow函数后，新分配给它的栈顶地址为0019FDC2,栈底地址为0019FE34。栈的大小是48h(72)个字节。

1. 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？



1. “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？

如图2-6-1

图2-6-1

1. 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?

mov语句是将buf的起始地址传递给edx。lea语句是将des的偏移地址传递给eax.

1. Strcpy完成后，查看des变量空间是否有改变？

Strcpy完成后，查看des变量空间变为了和longbuf非空元素相同的内容。

1. 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？

ret语句执行完后， EIP=00411586,ESP=0019FE38,EBP=61616161,程序跳 转到了77D8AEC8执行。

1. **溢出编程实验**
2. 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。

|  |
| --- |
| #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;  } |

因为buff数组大小为1，buff偏移2的地方是缓冲区之外，存在缓冲区溢出文体。实际上buff向下8个字节的位置是函数函数的返回地址，在程序调用ret返回到其调用函数时使会跳转到why\_here()出执行shellcode代码。

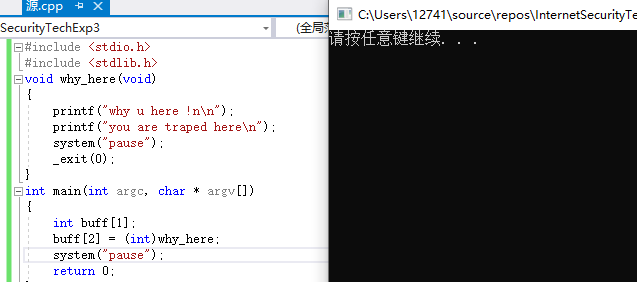
如图3-1-1

图3-1-1

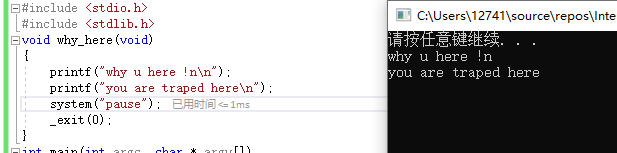
下图显示执行why\_here函数。如图3-1-2

图3-1-2

2,附加题思路。

由于操作系统每次加载可执行文件到进程空间的位置都是无法预测的， 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。 为了能准确定位shellcode的地址，需要借助一些额外的操作，其中最经典的 是借助跳板的栈溢出方式。

如果在函数的返回地址填入一个地址，该地址指向的内存保存了一条特殊的指令jmp esp——跳板。那么函数返回后，会执行该指令并跳转到esp所在的位置。这样，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？“幸运”的是，在Windows操作系统加载的大量dll中，包含了许多这样的指令，比如kernel32.dll，ntdll.dll，这两个动态链接库是Windows程序默认加载的。而且更“神奇”的是Windows操作系统加载dll时候一般都是固定地址，因此这些dll内的跳板指令的地址一般都是固定的。我们可以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。

**七、实验数据及结果分析：**

数据和结果见实验步骤中。

**八、实验结论、心得体会和改进建议：**

通过本次实验，了解了汇编语言函数的执行流程和规则，实践了部分缓存区溢出，更加了解溢出攻击的原理。在实验的过程中，也了解到基本的防范方法，希望以后自己能够做其他的溢出漏洞来逃避Windows自带的漏洞检测系统，比如搜索系统库函数找到jmp esp指令，再修改栈中数据执行栈中的shellcode代码。

**九、对本实验过程及方法、手段的改进建议**

无。

**报告评分：**

**指导教师签字：**