

# 《软件安全》实验报告

姓名：王昱      学号：2212046      班级：信息安全班

## 一.实验名称：

Shellcode 编写及编码

## 二.实验要求：

复现第五章实验三，并讲产生的编码后的 shellcode 在示例 5-4 中进行验证，阐述 shellcode 编码的原理、shellcode 提取的思想

## 三.实验过程：

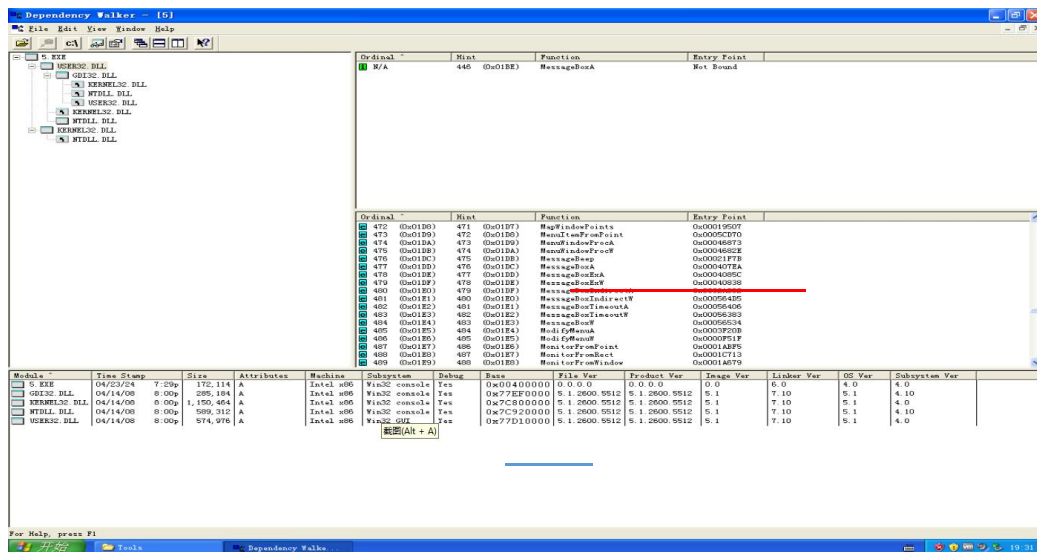
### （一）获取编码前的 shellcode：

我们要通过覆盖返回地址让进程去执行植入的 Shellcode 来进行漏洞利用，获取一个会弹出”Hello world”消息框的 shellcode 代码。那么首先我们要能够编写出一个能够弹出显示”Hello world”消息框的 shellcode。

#### 1.获取函数入口地址

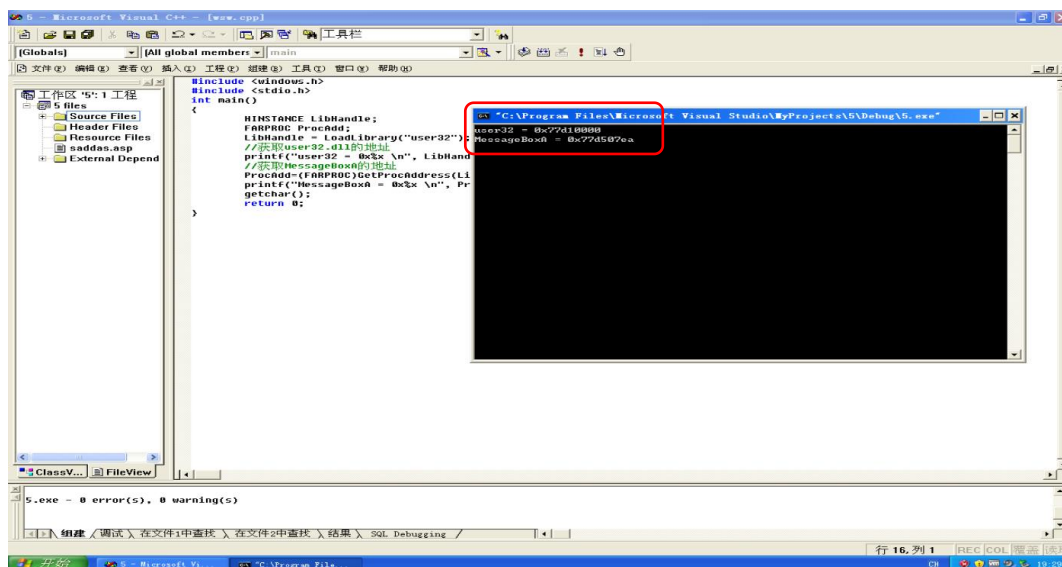
（1）由于需要调用”user32.dll”中的”MessageBoxA”方法，因此我们首先要获得这个方法的地址。MessageBoxA 的入口地址可以通过 user32.dll 在系统中加载的基址和 MessageBoxA 在库中的偏移相加得到。可以使用 VC6.0 自带的小工具”Dependency Walker”获得这些信息。

效果如下图所示，得到的 MessageBoxA 方法在内存中的地址为 0x77d507ea



(2)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);

效果如图所示，基址为 0x77d10000,MessageBoxA 的地址为 0077d507ea



## 2.提取 shellcode 代码

得到了 MessageBoxA 的入口地址之后，我们就可以首先将函数所需要的 4 个参数依次入栈，然后通过 CALL 指令直接调用函数。

int MessageBox(

HWND hWnd, // handle to owner window

LPCSTR lpText, // text in message box

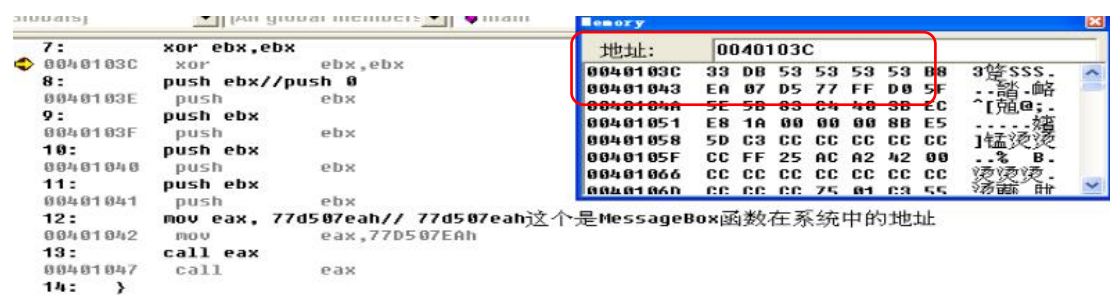
```
LPCTSTR lpCaption,    // message box title
UINT uType            // message box style
);
```

调用以上函数，进行反汇编，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。

```

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 (0042a2ac)]
00401038  cmp     esi, esp
0040103A  call    __chkesp (00401070)
6:      return;
```

根据汇编代码找到对应地址的机器码：



7: xor ebx, ebx  
0040103C xor ebx, ebx  
8: push ebx // push 0  
0040103E push ebx  
9: push ebx  
0040103F push ebx  
10: push ebx  
00401040 push ebx  
11: push ebx  
00401041 push ebx  
12: mov eax, 77d507eah // 77d507eah 这个是 MessageBox 函数在系统中的地址  
00401042 mov eax, 77D507EAh  
13: call eax  
00401047 call eax  
14: }

为了能够在弹出的消息窗口中展示出”Hello World”，我们需要将”Hello World”的字符串首先存入到内存中，4 字节存入，硬编码空格是 0x20。不足 4 字节，可以在最后的字节里补空格。“hello world”对应的 ASCII 码为：  
\x68\x65\x6C\x6C\x6F\x20\x77\x6F\x72\x6C\x64\x20。但是入栈的话，需要倒着来；考虑 bigendian 编码，存储顺序也得倒过来。



8: xor ebx, ebx  
0040103C xor ebx, ebx  
9: push ebx // push 0  
0040103E push ebx  
10: push 20646C72h  
0040103F push 20646C72h  
11: push 6F77206Fh  
00401044 push 6F77206Fh  
12: push 6C6C6568h  
00401049 push 6C6C6568h  
13: mov eax, esp  
0040104E mov eax, esp  
14: push ebx // push 0  
00401050 push ebx  
15: push eax  
00401051 push eax  
16: push eax  
00401052 push eax  
17: push ebx  
00401053 push ebx  
18: mov eax, 77d507eah  
00401054 mov eax, 77D507EAh  
19: call eax  
00401059 call eax  
20: call eax  
21: }  
22: return;  
23: }

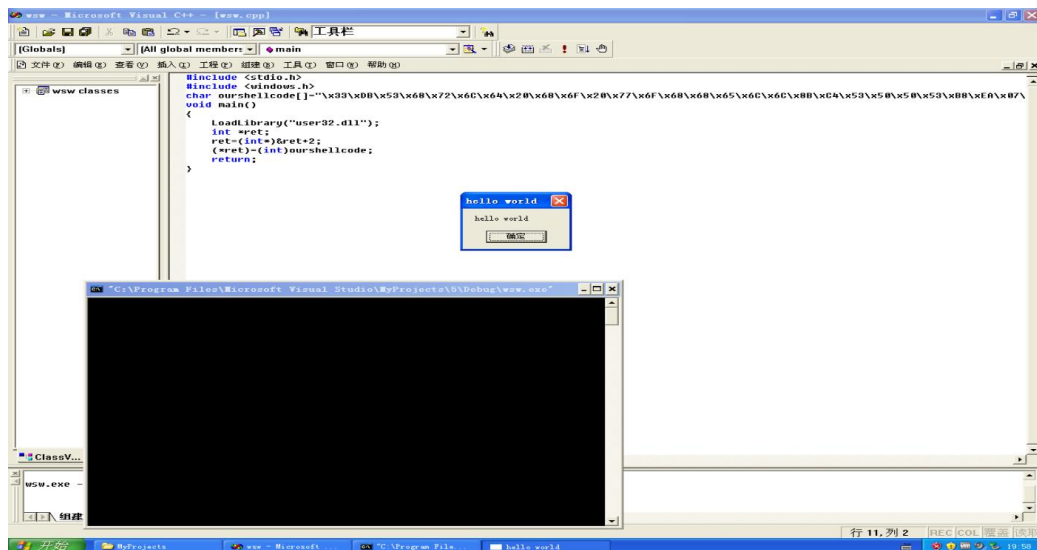
通过加入断点，查看反汇编代码，然后根据汇编代码的地址直接定位机器码的内容，得到了能够弹出”Hello world”消息框的 shellcode，提取到的 shellcode 如

下：

\x33\xDB\x53\x68\x72\x6C\x64\x20\x68\x6F\x20\x77\x6F\x68\x68\x65\x6C\x6C\x8B\xC4\x53\x50\x50\x53\xB8\xEA\x07\xD5\x77\xFF\xD0

### 3.运行结果：

可以看到，程序能够运行，弹出了 hello world 的窗口信息！



### （二）编码

Shellcode 代码编制过程通常需要进行编码，因为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”。

编写对于 shellcode 的编码程序，并将编码后的结果输出到文件中。本次编码程序采用异或编码：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。此外，还有一些自定义编解码方法被采用，包括简单加解密、alpha\_upper 编码、计算编码等。

将得到的机器码和指定的一个 key 值进行异或操作，原因是异或操作是可逆的，程序代码如下：

```

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
void encoder(char* input, unsigned char key)
{
    int i = 0, len = 0;
    FILE * fp;
    len = strlen(input);
    unsigned char * output = (unsigned char *)malloc(len + 1);
    for (i = 0; i < len; i++)
        output[i] = input[i] ^ key;
    fp = fopen("encode.txt", "w+");
    fprintf(fp, "");
    for (i = 0; i < len; i++)
    {
        fprintf(fp, "\\x%0.2x", output[i]);
        if ((i + 1) % 16 == 0)
            fprintf(fp, "\\n\\");
    }
    fprintf(fp, "\\");
    fclose(fp);
    printf("dump the encoded shellcode to encode.txt OK!\n");
    free(output);
}

int main()
{
    char sc[] = "\\x33\\x0b\\x53\\x68\\x72\\x6c\\x64\\x20\\x68\\x6f\\x20\\x77\\x6f\\x68\\x65\\x6c\\x6c\\x8b\\xc4\\x53\\x50\\x53\\x8b\\xe8\\x07\\xd5\\x28\\x28\\xcf\\x80\\x17\\x14\\x14\\x17\\xfc\\xae\\x43\\x91\\x33\\xbb\\x94\\xd4"
    ....
    return 0;
}

```

得到编码后的 shellcode 如下：

```

"\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"
....

```

### \*\*注意事项:

(1) 传入的待编码 sc[]数组最后要加上“\x90”，用于后面的解码程序来判断是否结束。

(2) 当“\x90”作为结束符时，编码中间不能再出现任何的“\x90”，否则会影响解码程序的正常运行。

(3) 在选取编码字节（此处为 0x44）时，不可与已有字节相同，否则会出现 0。

### (三) 解码:

#### 1.eax 记录 shellcode 当前的起始地址

用如下代码:

```

#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;
}

```

其中核心语句在于

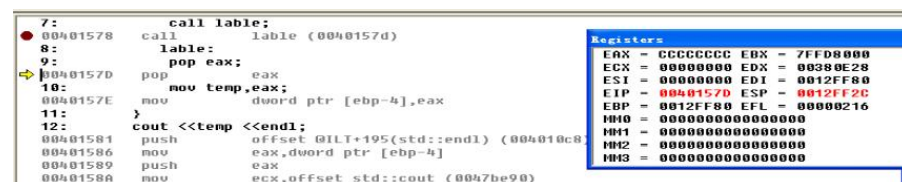
call lable;

lable: pop eax

经过这两句汇编代码之后，`eax` 的值就是当前指令地址了。原因是 `call lable` 的时候，会将当前 `EIP` 的值（也就是下一条指令 `pop eax` 的指令地址）入栈。

截图分析：

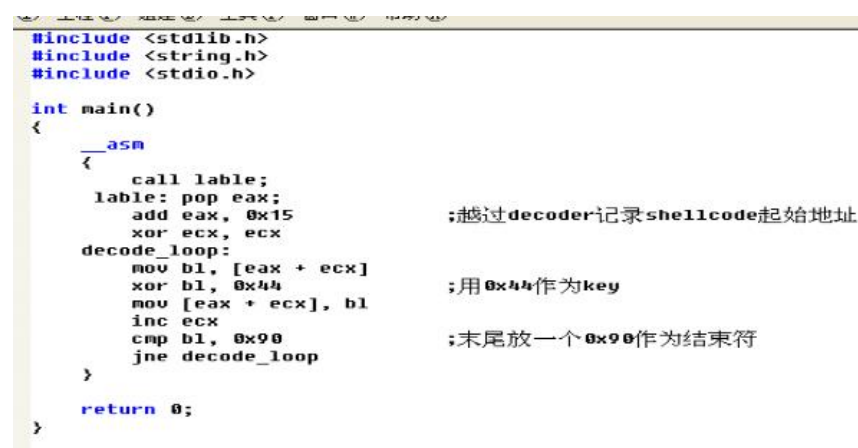
下图中，先将下一条指令的地址 `EIP` 压入栈中，在 `ESP` (`0012FF2C`) 栈顶位置存放了 `0040157D`。



之后 `pop eax`, `eax` 变成 `0040157D`，即获取了 `label` 对应指令 (`pop eax`) 所在地址。



2.完整的解码程序如下：



· `add eax, 0x15`

首先，这段汇编指令的总长度是 `0x15`，`eax` 是当前解码程序的起始地址；

其次，所生成的解码器会与编码后的 `shellcode` 联合执行；

综上，`eax + 0x15` 得到的就是这段汇编指令的下一块地址，即编码后的 `shellcode` 的起始地址。

· `xor ecx, ecx`

将 `ecx` 置 0，用于下面的循环判断。

· `decode_loop: mov bl, [eax + ecx]`

寄存器 b1 存放 eax+ecx 地址里的数据（shellcode 起始地址+ecx 偏移），由于 ecx 从 0 开始每次循环+1.所以 b1 在循环中会依次拿到 shellcode 的每个字节。

`xor bl, 0x44 ;`

将编码后的 shellcode 的每个字节与 0x44 异或，实现解码

`mov [eax + ecx], bl`

将解码后的结果放回原位

`inc ecx`

`cmp bl, 0x90 ;`

由于在编码时对编码前的 shellcode 多加了一个“\x90”的结束符，所以在  
这里判断 b1 与“\0x90”是否相等，若相等，说明解码结束。

`jne decode_loop`

跳转

### 3.得到的完整编码：

（1）提取到的机器码为：

"\xE8\x00\x00\x00\x00\x58\x83\xC0\x15\x33\xC9\x8A\x1C\x08\x80\xF3\x44\x88\x1C\x08\x41\x80\FB\x90\x75\F1"

Address:	E80401020
00401020	48 8B 00 00 00 58 83 C0 15 33 C9 8A 1C 08 80 F3
00401028	44 8B 1C 8B 41 80 FB 90 75 F1 33 C8 5F 5E 8B 83
00401030	C4 4B 3B EC EB EF 8B 8B 8B 8B E5 5B CB CE CE
00401038	EC CE CE CE CE CE CE CE CE CE CE CE CE CE CE
00401040	EC CE CE CE CE CE CE CE CE CE CE CE CE CE CE

（2）编码后的 shellcode:

"\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"

(3)链接两段机器码后，得到完整 shellcode 如下：

"\xE8\x00\x00\x00\x00\x58\x83\xC0\x15\x33\xC9\x8A\x1C\x08\x80\xF3\x44\x88\x1C\x08\x41\x80\FB\x90\x75\F1\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"



## 4.验证

运行结果：说明 shellcode 编码和解码正确



## 四.心得体会:

### 1.shellcode 编码的原理

Shellcode 编码是为了避免在传输过程中被检测系统识别或是因为某些字符限制而无法执行。编码的过程通常涉及将 shellcode 转换成一种格式，使其在网络传输或是存储时不会触发安全系统的警报。例如，攻击者可能会使用百分比编码、转义序列编码或实体编码来隐藏 shellcode。编码后的 shellcode 需要在目标系统上解码才能执行原始的攻击代码。

首先通过编写常规的注入代码，将其转换成机器码获取 shellcode 的内容，然后通过覆盖返回地址，让 CPU 在执行完某函数返回时跳转去执行 shellcode。使其可以更稳定、绕过更多安全检测。

### 2.shellcode 提取的思想

Shellcode 提取是指从受感染的文件或内存中提取出 shellcode 的过程。这通常涉及到静态分析和动态分析的结合使用。静态分析可以帮助识别可疑的代码段，而动态分析则可以在运行时监视和提取 shellcode。

注入的 shellcode 若想正常执行，通常需要包含一些动态链接库，因此要提前定位好动态链接库的位置，然后再在动态链接库中定位所需要加载的函数的地



址，进而能够直接通过函数地址调用函数。对于如何提取 `shellcode`，可以先通过工具查看所需要的动态链接库和函数的地址，然后将自己想要植入的代码按照 C 或者汇编的方式写出来，编译后查看对应的机器码，然后将所得到的机器码组成 `shellcode`。

提前定位好自己植入的 `shellcode` 在内存中的地址，考虑在程序特定的函数中将其植入，覆盖该函数的返回地址，进而使得该函数返回的时候能够跳转去执行 `shellcode`。