软件安全实验报告

姓名：郭子涵　学号：2312145　班级：信息安全、法学双学位班

1　实验名称：

shellcode编写及编码

2　实验要求：

复现第五章实验三，并将产生的编码后的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。

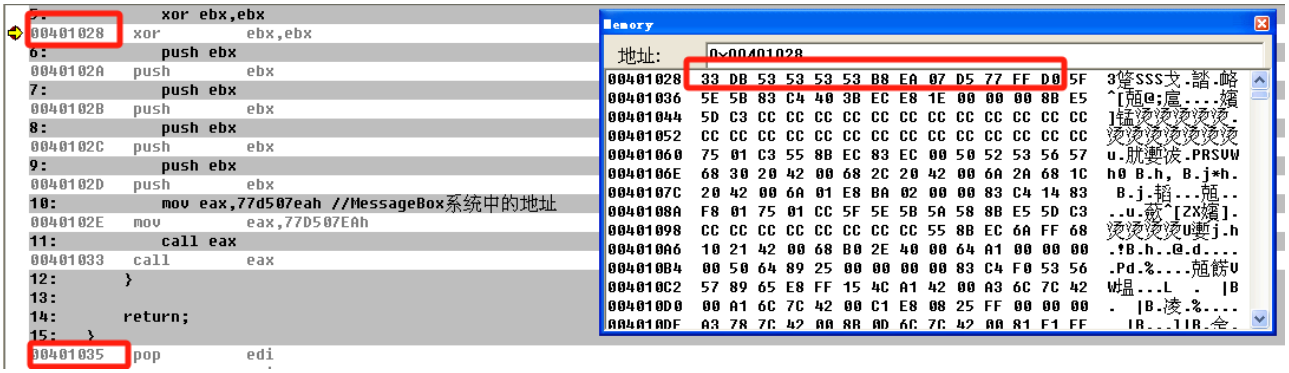
3　实验过程：

3.1　shellcode代码的编写和提取：

用C语言书写要执行的shellcode程序:

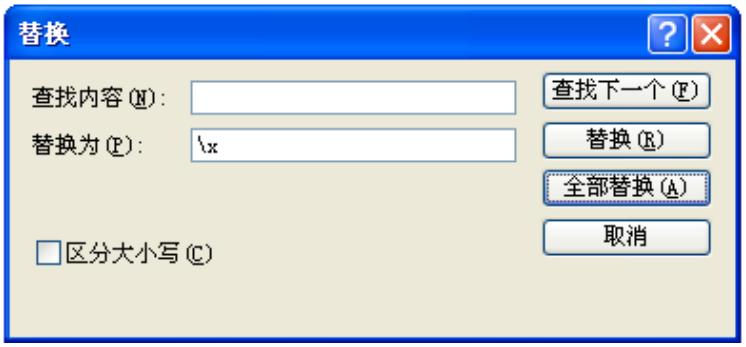
```
#include <stdio.h>
#include <windows.h>
void main()
{
MessageBox(NULL,NULL,NULL,0);
return;
}
```

换成对应的汇编代码，在代码的第一行处打断点，定位具体内存中的地址：



由上图可以看出，此段代码的地址为00401028-00401034，搜索地址可以看出对应的机器码应为：33 DB 53 53 53 53 B8 EA 07 D5 77 FF D0。利用记事本工具，用替换功能将空格转化为字节表示的方法：

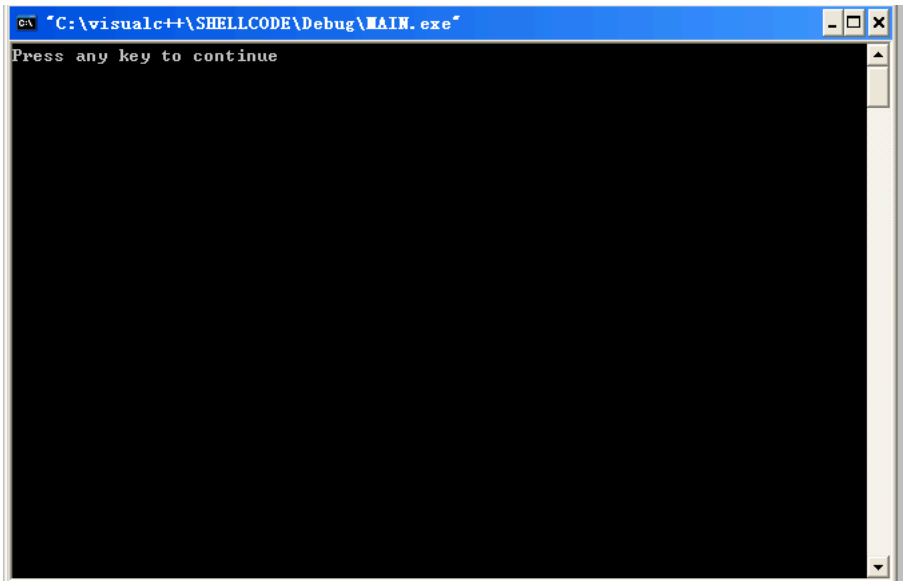
```
33\xDB\x53\x53\x53\x53\xB8\xEA\x07\xD5\x77\xFF\xD0
```



编写测试程序，填充shellcode的机器码：

```
#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;
}
```

运行程序可得下列弹窗，证明我们的shellcode代码是正确的：

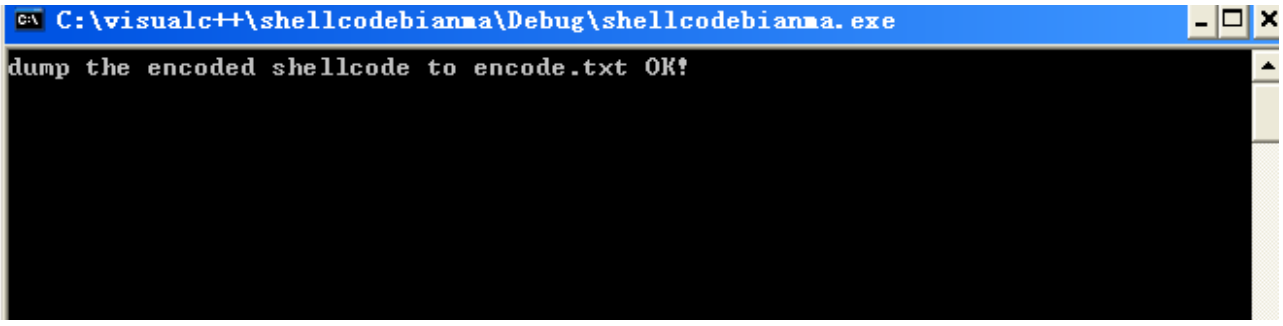


3.2 shellcode的编码：

异或编码程序：

```
#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; //输入的每一个字节异或密钥
        fp = fopen("encode.txt", "w+");//输出写入一个文件中
        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()
{   //输入编码对象
    char sc[] =
"\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\x90";
    encoder(sc, 0x44);//进行异或编码，并输入密钥
    getchar();
    return 0;
}
```

运行测试显示输出成功：



生成编码后的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"
""
```

编写解码程序（后接shellcode）：

```
void main()
{
    __asm
    {
    add eax, 0x14 ;前提为：eax为当前指令的起始地址，加上0x14即越过 decoder 记录 shellcode
起始地址eax指向此段代码下一条指令，即指向上述shellcode
    xor ecx, ecx ;ecx=0
    decode_loop:
    mov bl, [eax + ecx];从头依次表示shellcode的字节
    xor bl, 0x44 ;用 0x44 作为 key，异或
    mov [eax + ecx], bl;在放回去
    inc ecx
    cmp bl, 0x90 ;末尾放一个 0x90 作为结束符
    jne decode_loop
    }
}
```

解码器流程图：

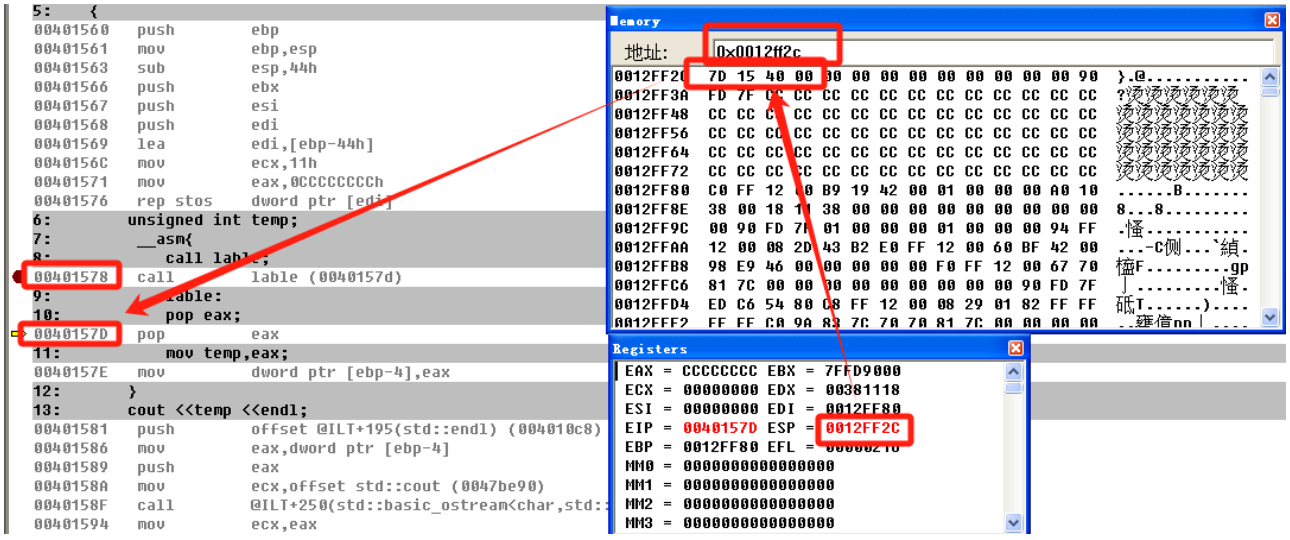
```
flowchart TD
    A[Shellcode 起始] --> B[call label]
    B --> C[pop eax ← 当前地址]
    C --> D[add eax, 0x15]
    D --> E[ECX ← 0]
    E --> F[取 shellcode 字节]
    F --> G[异或 key (0x44)]
    G --> H[是否等于 0x90?]
    H -- 否 --> F
    H -- 是 --> I[跳出，执行原始 shellcode]
```

编写如下程序，通过思考如下程序的原理来实现让`eax`记录shellcode的当前起始代码：

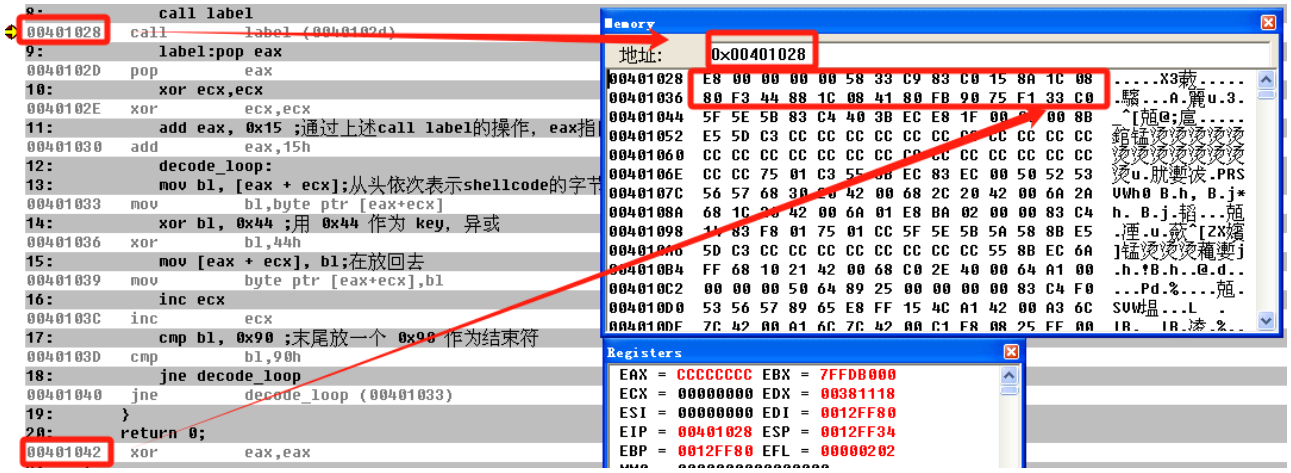
```
#include <iostream>
using namespace std;
int main(int argc, char const *argv[])
{
     unsigned int temp;
     __asm{
```

```
        call lable;//跳转到label标签
        lable:
        pop eax;//返回地址入栈
         mov temp,eax;
    }
     cout <<temp <<endl;
     return 0;
}
```

通过`call label;lael:pop eax;`语句，之后`eax`即为当前指令的地址，如下图展示：




```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
int main()
{
    __asm
    {
        call label
        label:pop eax
        xor ecx,ecx
        add eax, 0x15 ;通过上述call label的操作， eax指向上一条指令的地址，因此此处为0x15
        decode_loop:
        mov bl, [eax + ecx];从头依次表示shellcode的字节
        xor bl, 0x44 ;用 0x44 作为 key，异或
        mov [eax + ecx], bl;在放回去
        inc ecx
        cmp bl, 0x90 ;末尾放一个 0x90 作为结束符
        jne decode_loop
    }
    return 0;
}
```



提取机器码`E8 00 00 00 00 58 33 C9 83 C0 15 8A 1C 08 80 F3 44 88 1C 08 41 80 FB 90 75 F1`

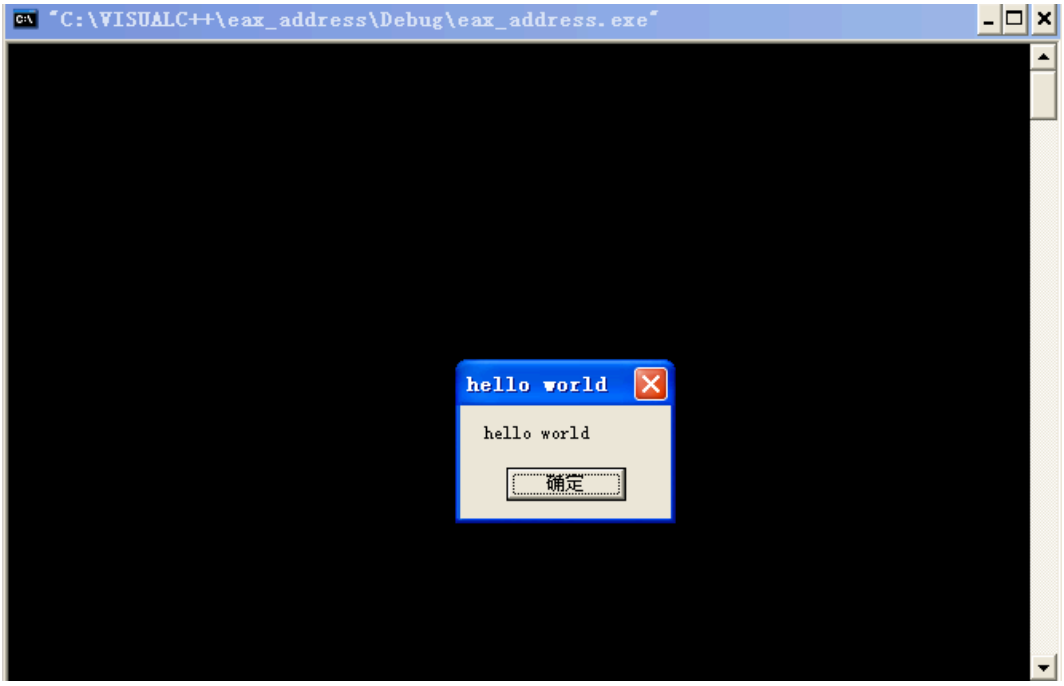
与`shellcode`的机器码连接起来就可以完成完整的机器码程序：

```
\xE8\x00\x00\x00\x00\x58\x33\xC9\x83\xC0\x15\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\x
17\x14\x14\x17\xfc\xae\x43\x91\x33\xbb\x94\xd4
```

运行下述完整的代码：

```
#include <stdio.h>
#include <windows.h>
char
ourshellcode[]="\xE8\x00\x00\x00\x00\x58\x33\xC9\x83\xC0\x15\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\x
28\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;
}
```

得到结果，说明程序正确：



4 心得体会:

本实验系统地从“零基础”出发，完成了一个 Shellcode 的提取、编写、编码与利用测试 的全过程在实验过程中首先对 Shellcode 本质的认识更深刻Shellcode 本质上是一段高度精简、可控、特定用途的机器码，可以在特定上下文中直接执行。它往往被放置在程序内存中，通过修改控制流（如覆盖返回地址）执行，达到绕过程序逻辑甚至系统调用的目的。

通过实验更加深入理解shellcode编码目的:

1. 避开坏字符：如 `\x00`（NULL），`\x0a`（换行），`\x20`（空格）等字符会在某些函数（如 `strcpy`）中截断字符串或被过滤。
2. 绕过安全检测：通过变形后的编码 Shellcode，能绕过防病毒软件、IDS/IPS 的签名检测。
3. 适应平台：某些平台限制了可以使用的字符集（如网页传输、数据库注入时只能用可打印字符）。

在实际攻击中，shellcode 的“隐蔽性”和“可执行性”同等重要。编码就是为了增加其生存能力。

实验编写并验证了 Shellcode 的生成流程从 C 语言编译出机器码；使用汇编进行改写，解决坏字符问题；提取指令字节（即 shellcode）；测试执行是否生效；编码 + 解码器嵌套到 shellcode 中；最后验证“带壳”的 shellcode 是否可用。让我明白 “攻击的成功不只取决于 payload 本身，更依赖其传输、解码和执行策略的设计”。同时我还掌握了掌握了编码器和解码器的编写方式异或编码器的实现简洁易懂（比如用 0x44 做异或 key）；解码器通过 `EAX` 指向 shellcode 起始地址进行解密；以及学会了如何通过 `call-label → pop eax` 方式取得当前 EIP（写通用解码器时非常关键）。这一部分让我对汇编语言、寄存器使用、栈帧结构有了更实战性的理解。