《软件安全》实验报告

姓名：蒋枘言　学号：2313546　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，并将产生的编码后的 shellcode 在示例 5-4 中进行验证，阐述 shellcode 提取的思想、shellcode 编码的原理。

三、实验过程

（1）提取 shellcode 代码

打开 VC6，输入以下代码：

利用调试功能，找到其对应的汇编代码，如下图所示。

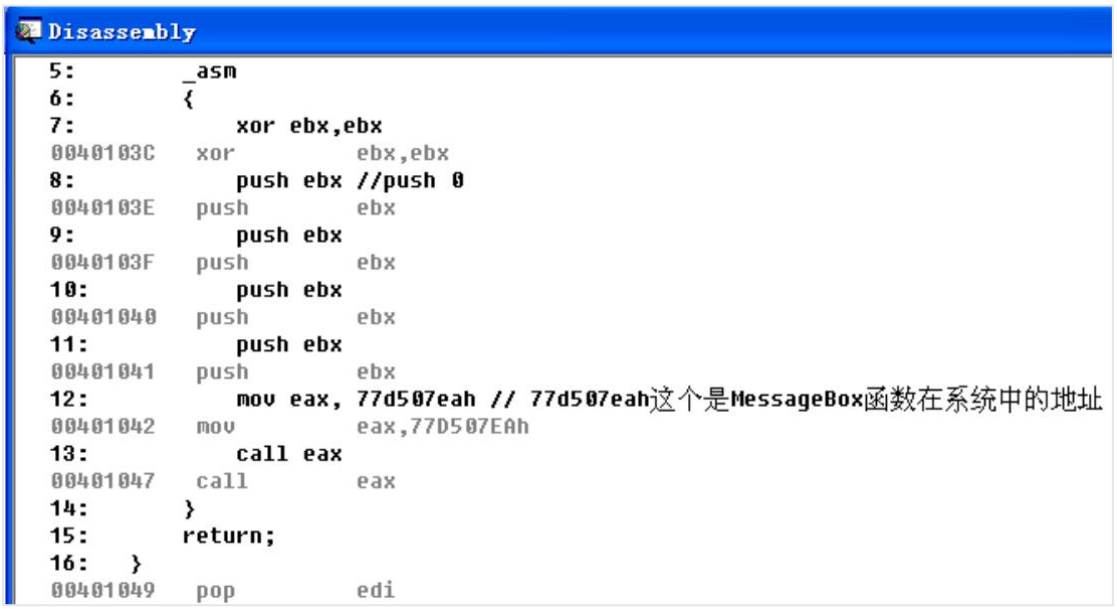
```
5:        MessageBox(NULL,NULL,NULL,0);
00401028  mov         esi,esp
0040102A  push        0
0040102C  push        0
0040102E  push        0
00401030  push        0
00401032  call        dword ptr [__imp__MessageBoxA@16 (0042428c)]
```

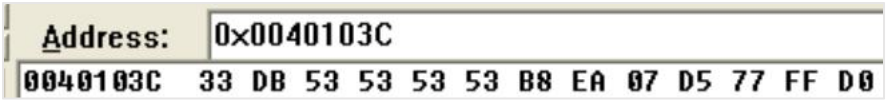
由于 push 0 的二进制机器码中会出现 00（即出现空字节），为了避免 shellcode 被截断，这一般是不允许的，所以我们还要对代码进行加工，改为以下的代码：

```
#include <stdio.h>
#include <windows.h>
void main() {
    LoadLibrary("user32.dll"); //  加载 user32.dll
    __asm
    {
        xor ebx, ebx //  使用 xor 清零
```

```
        push ebx // 代替原来的 push 0
        push ebx
        push ebx
        push ebx
        mov eax, 77D507EAh // 0x77D507EA 是 MessageBoxA 函数在系统中的地址
                           // 获得该地址的方法详见教材 5.1.3 节，此处不详细介绍
        call eax
    }
    return;
```

在 VC6 里调试以上程序，转到反汇编，查看_asm 里的汇编代码的地址，发现为 0040103C-00401048。查看该地址的二进制机器码，得到：33 DB 53 53 53 53 B8 EA 07 D5 77 FF D0。





现在已经完成了对 shellcode 的提取，可以编写以下程序进行测试：

```
#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;
    // 声明一个整型指针 ret，未初始化，此时 ret 指向随机内存地址。
```

```
    ret = (int*)&ret + 2;
    // &ret：获取指针变量 ret 自身的地址（即指针的指针）。
    // (int*)&ret：将& ret 强制转换为 int* 类型，此时它是一个指向 int* 类型变量的指针。
    // + 2：指针算术运算。由于 int* 的步长为 sizeof(int)（4 字节）， + 2 会移动 2 * 4 = 8 字节。

    (*ret) = (int)ourshellcode;
    // &ret 是 ret 变量的地址，(int*)&ret + 2 会指向当前栈帧中返回地址的位置。
    // 此操作的目的是让 ret 指向函数的返回地址。
    return;
}
```

在 VC6 里运行以上程序，发现成功弹出对话框（显示错误是因为目前还没有传入要显示的参数）。

说明 shellcode 的提取成功。但是现在想要对话框显示“hello world”则需要将以上代码进行修改来得到想要的 shellcode。

```
#include <stdio.h>
#include <windows.h>
void main()
{
    LoadLibrary("user32.dll");
    __asm
    {
        xor ebx, ebx
        push ebx
// “hello world”对应的 ASCII 码为：\x68\x65\x6C\x6C\x6F\x20\x77\x6F\x72\x6C\x64\x20。
// 考虑小端序，在内存中的实际存储为 6C6C6568 6F77206F 20646C72（地址由小到大）。
// 入栈需要倒着来，大地址先入栈。
        push 20646C72h
        push 6F77206Fh
        push 6C6C6568h
        mov eax, esp
        push ebx
        push eax
        push eax
        push ebx
        mov eax, 77D507EAh
        call eax
    }
    return;
}
```

在 VC6 里运行以上程序，发现此时弹出的对话框成功显示了“hello world”。

（2）shellcode 编码

shellcode 通常都需要进行编码，目的是避免坏字（如空字符、换行、空格等等）、符绕过检测、适配协议限制等。

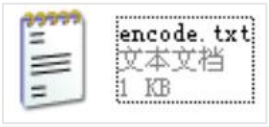
通过（1）中的方法可以提取到弹出显示“hello world”的对话框的 shellcode 为：\x33\xDB\x53\x68\x72\x6C\x64\x20\x68\x6F\x20\x77\x6F\x68\x68\x65\x6C\x6C\x8B\xC4\x53\x50\x50\x53\xB8\xEA\x07\xD5\x77\xFF\xD0\x90（在结尾添加 90，机器码 90 对应的指令是 NOP（空操作），便于后续编码解码操作）。

shellcode 编码的方式有很多，这里采用**异或编码**。通过以下程序即可得到编码后的 shellcode，储存在 encode.txt 中。

```
#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); // 计算输入字符串长度（直到遇到'\0'）
    unsigned char* output = (unsigned char*)malloc(len + 1); // 分配内存来存储编码后的结果

    for (i = 0; i < len; i++){
        output[i] = input[i] ^ key;   // 对每个字符进行 XOR 编码（异或密钥为 key）
    } // 加密过程

    fp = fopen("encode.txt", "w+"); // 创建文件 encode.txt
    fprintf(fp, "\""); // 写入起始双引号（注意转义字符）

    for (i = 0; i < len; i++) {
        fprintf(fp, "\\x%0.2x", output[i]); // 将每个字节格式化为 16 进制转义字符
        if ((i + 1) % 16 == 0) // 每 16 字节换行并添加新双引号
            fprintf(fp, "\"\n\"");
    }
    fprintf(fp, "\""); // 写入结束双引号
    fclose(fp);
    printf("dump the encoded shellcode to encode.txt OK!\n");
    free(output);   // 释放内存
}
```

```
int main() {
    // 原始 shellcode
    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";

    // 调用编码函数，密钥为 0x44
    encoder(sc, 0x44);

    getchar();
    return 0;
}
```

运行程序，得到如下结果：



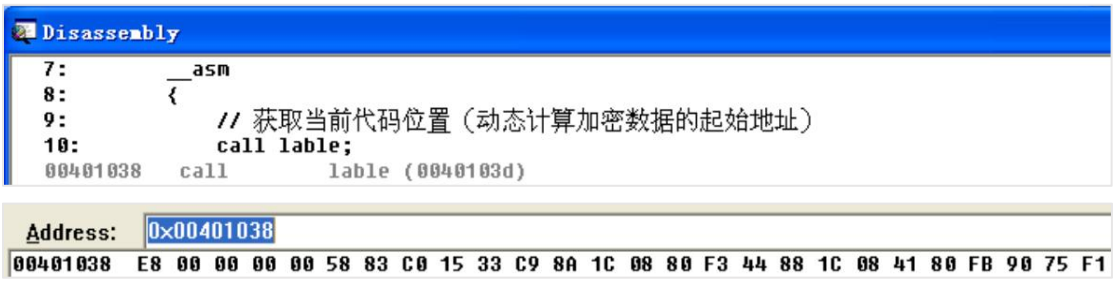
（3）shellcode 解码

编码后的 shellcode 必须经过解码才能正确运行，我们需要编写一个解码程序，解码程序会与编码后的 shellcode 联合执行。解码程序如下：

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main()
{
    __asm
    {
        // 获取当前代码位置（解码程序的关键是动态计算加密 shellcode 的起始地址）
        call lable;
    lable:
        pop eax // 弹出 call 指令压入的返回地址到 EAX（此时 EAX 指向 lable 标签地
址）
        add eax, 0x15 // 计算加密数据的起始地址（跳过后续汇编指令，定位到加密数
据区）
        xor ecx, ecx
```

```
        // 动态解码循环（异或解码）
        decode_loop :
            mov bl, [eax + ecx] // 读取加密数据的 1 个字节到 BL 寄存器
            xor bl, 0x44 // 用密钥 0x44 进行异或解码
            mov[eax + ecx], bl   // 将解码后的字节写回原地址（覆盖加密数据）
            inc ecx // 计数器+1，处理下一个字节
            cmp bl, 0x90 // 检查解码后的字节是否为 0x90（NOP 指令）
            jne decode_loop // 如果不是 0x90，继续循环
    }
    return 0;
}
```

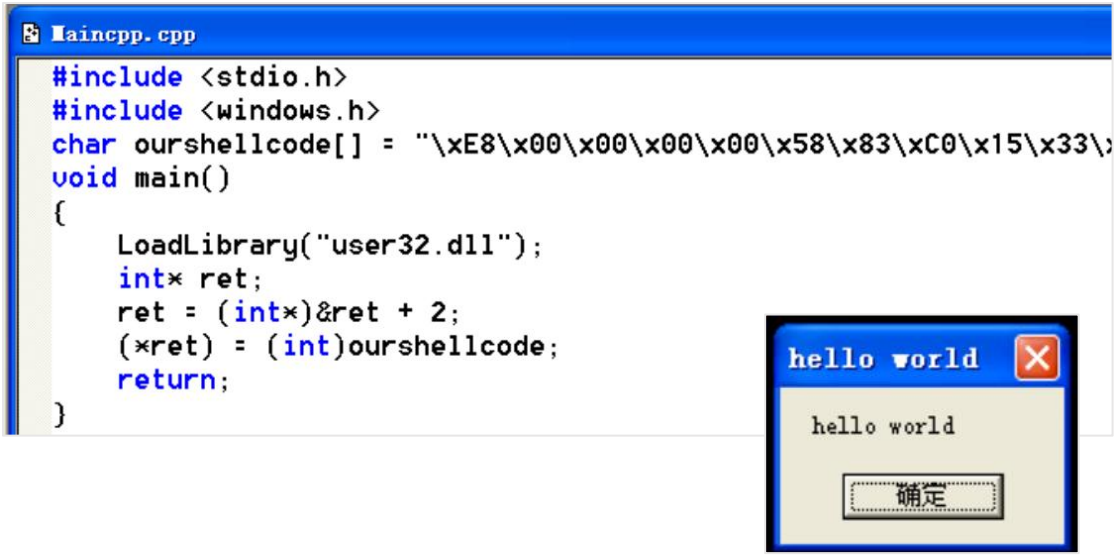
通过同样的方法提取到的机器码如下图内存显示所示：



最后，将现在得到的 shellcode 解码程序的机器码与（2）中得到的加密后的 shellcode 连接，就得到了完整的 shellcode：

\xE8\x00\x00\x00\x00\x58\x83\xC0\x15\x33\xC9\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\x17\x14\x14\x17\xFC\xAE\x43\x91\x33\xBB\x94\xD4

现在进行验证：在 VC6 里运行以下程序，可以发现成功弹出了显示“hello world”的对话框，故完整的 shellcode 编码成功！



四、心得体会

（1）shellcode 提取的思想

1.核心目标：提取可独立运行的二进制机器码，确保其不依赖外部环境，且能直接注入目标程序执行。

2.关键步骤

①避免坏字符：使用汇编指令替代会产生空字节的操作。例如，通过 xor ebx, ebx 清零寄存器，替代 push 0（机器码含 0x00），防止输入函数截断。

②动态获取函数地址：直接调用系统函数（如 MessageBoxA）需硬编码地址（如 0x77D507EA），但实际应用中需通过动态解析绕过 ASLR。

③调试提取机器码：利用调试工具查看汇编指令对应的二进制代码，手动提取为十六进制字符串。

④栈帧布局控制：通过修改返回地址（如 ret = (int*)&ret + 2）劫持程序流程，使返回地址指向 shellcode 起始位置，触发执行。

（2）shellcode 编码的原理

1.异或（XOR）编码实现：逐字节与固定密钥异或（如 0x44）。数学上有 data ^ key ^ key = data，确保运行时可还原。

2.终止标识：在 shellcode 末尾添加 0x90（NOP 指令），解码时检测到 0x90 停止循环。

3.解码程序：需要动态确定 shellcode 起始位置，**这里使用 call + pop 技巧获取当前指令地址**，通过偏移计算定位加密数据区。