**Windows平台高效Shellcode编程技术实战**

# 第一讲：什么是shellcode

shellcode本质上就是一段代码，但是与普通代码不同：

独立的存在，无需任何文件格式的包装。

普通的程序：编写代码→编译生成PE格式的可执行文件→执行PE格式文件。

任意进程内存中运行，无需固定指定的宿主进程。

shellcode的基本特点：

短小精悍：shellcode就是代码本身，没有经过格式包装，直接执行核心功能

灵活多变：因为不依赖与任何文件格式的约束，所以可以以任何方式加密或变形。

shellcode生成方式：2大类

编程语言编写：汇编语言、C语言

可以得到任意灵活的shellcode，前提是有一套合理的设计方法或生成框架

第一种shellcode生成方案：将shellcode全部写入程序的代码段，然后从代码段将shellcode部分提取出来；

shellcode生成器：shell storm、Msfvenom

建立在shellcode库的基础上，生成器将事先存在shellcode库中的模块取出来给我们。优点：方便快捷；缺点：模块都是通用型的，功能比较单一，无法获得私有的，针对某一特定问题的复杂的shellcode。

# 第二讲：基本环境的搭建

第一种shellcode生成方案：

开发平台：VC6.0 、VS2005、VS2008、VS2010、VS2012、VS2013、VS2015。

工程创建：win32控制台程序

建立一个win32控制台的空程序（取消SDL检查），添加main.cpp文件，创建main函数:

int main()

{

return 0;

}

选择release版本，编译生成exe文件。

通过IDA查看exe文件，可以看出生成的exe文件不仅加入了PE格式，VS还向其加入了很多初始化代码（IDA的Function window中很多函数名）。

VS自动加入的初始化代码很影响我们生成shellcode，要将这些额外代码去掉（两步）：

1.修改入口点

项目→配置属性→链接器→高级→入口点→输入任意自定义名字（eg：EntryMain）→将源文件中的main函数名对应改为入口点名字（eg：EntryMain）→编译生成exe文件→使用IDA打开exe文件→查看IDA的Function window中只有start函数（对应入口函数），可能还有个把其它函数（VS版本不同），继续进行第二步；

2.关闭缓冲区安全检查

项目→配置属性→C/C++→代码生成→安全检查→禁用→编译生成exe文件→使用IDA打开exe文件→查看IDA的Function window中必定只有start函数→在Function window中双击start函数，鼠标在IDA汇编代码中定位到start函数→鼠标左键按在汇编代码start函数处同时按F5，可以跳转到start函数对应的伪代码(C代码)处；

注意：关闭缓冲区安全检查可以取消VS生成的exe文件中的.data段和. reloc段，前提是程序中没有用到静态的或者全局或者常量的数据。

完成上述两步之后，我们就可以将shellcode写在入口函数处，为了让shellcode框架更完美，还可以进一步优化。

3.设置工程兼容windows XP

VS2015编译生成的exe文件是不支持在window XP下运行的。

项目→配置属性→常规→平台工具集→选择支持window XP的→C/C++→代码生成→运行库→多线程（MT）→编译生成exe文件；

运行库有四个选择：

多线程（MT）：对应release版本，不含调试信息（Debug），将DLL中的调用复制到源代码中（不再依赖DLL文件）；

多线程调试（MTd）：Debug版本，包含调试信息，不依赖DLL文件

多线程DLL（MD）：release版本，依赖DLL文件。

多线程调试DLL（MDd）：Debug版本，包含调试信息，依赖DLL文件。

4.关闭生成清单

通过PE文件查看器，查看exe文件有：.text代码段、.rdata只读数据段、.rsrc资源段。理想的是要让程序只有.text代码段。.rsrc资源段就VS默认添加的清单数据。

项目→配置属性→链接器→清单文件→生成清单→否→编译生成exe文件；

注意：如果解决方案中包含了资源文件（resource.h），那么即使关闭生成清单文件的话编译生成的exe文件依然有.rsrc段。

5.关闭调试信息

对应没有去除的.rdata数据段；

项目→配置属性→链接器→调试→生成调试信息→否；理论上可以去掉.rdata段（VS2013可以），但VS2015不能去除，不过没去除也不影响生成shellcode；

# 第三讲：shellcode编写原则一

本规则主要适用于VC和VS平台。其它的平台可能有各自特殊的规则和处理。这里讲的是通用型的原则。

首先要阻止VS编译器对程序的优化（比如：未使用的变量直接被视为冗余代码去掉了）：项目→配置属性→C/C++→优化→已禁用；

1.杜绝双引号字符串的直接使用：VS开发平台中，双引号字符串会被编译到只读数据段，以引用绝对地址的方式使用。而在shellcode编程中，要尽量避免一切绝对地址的使用。

eg1：

int EntryMain()

{

char szData[] = "123456";

return 0;

}

编译生成exe文件，使用PE文件查看器，可以看到此时的exe文件包含：.text段、.rdata段（存放“123456”）、.reloc段（重定位段）。

eg2：

int EntryMain ()

{

char szData[] = {'1', '2', '3', '4', '5', '6', '\0'};

return 0;

}

编译生成exe文件，使用PE文件查看器，可以看到此时的exe文件只包含：.text段。可知字符串“123456”被隐藏在代码代码段中了。

eg3：

int EntryMain ()

{

char szData[10] ;

memcpy(szData, "123456");

sprintf(szData, "%s", pszData);

return 0;

}

上述绿颜色的都是直接使用双引号字符串，在编写shellcode的时候是不行，可以改为如下：

eg4：

int EntryMain ()

{

char szData[10] ;

char szBuf[] = {'1', '2', '3', '4', '5', '6', '\0'};

char formate[] = {'%', 's', '\0'};

memcpy(szData, szBuf);

sprintf(szData, formate, pszData);

return 0;

}

shellcode中字符串定义的第二种方式：内嵌汇编的方式

eg5:

int EntryMain ()

{

\_\_asm

{

\_EMIT '1'

\_EMIT '2'

\_EMIT '3'

\_EMIT '4'

\_EMIT '5'

\_EMIT '6'

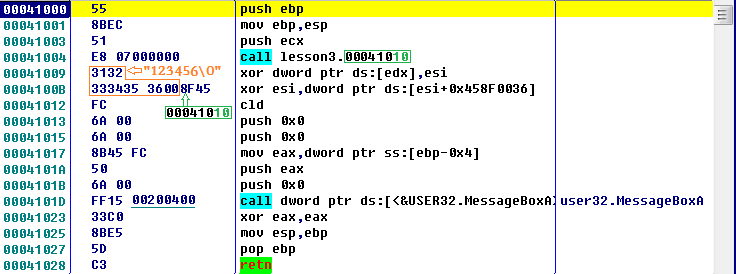
\_EMIT 0

}

return 0;

}

上述内嵌汇编中的”\_EMIT”相当于伪指令”db”，表示在当前内存中存一个字符。但是上述代码有缺陷，因为它在代码段，会被当做指令执行，但是它不是有效指令无法执行，所以会导致程序崩溃，改进如下：

eg6:

int EntryMain ()

{

char \*pszData;

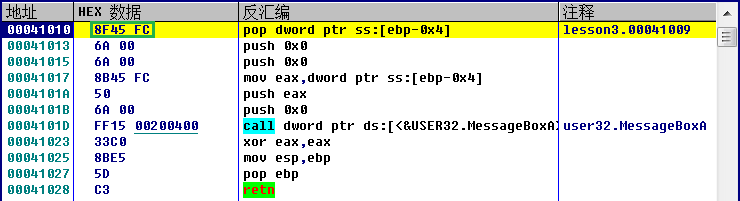
\_\_asm

{

call \_pszData

\_EMIT '1'

\_EMIT '2'

 \_EMIT '3'

\_EMIT '4'

\_EMIT '5'

\_EMIT '6'

\_EMIT 0

\_pszData:

pop pszData

}

return 0;

}

# 第四讲：shellcode编写原则二

2.函数动态调用：杜绝一切绝对地址的直接引用，不能引用函数的绝对地址，否则shellcode就无法实现通用性，必须动态地获取函数入口地址；

所有的工程的配置都与第二讲中的配置一致

eg7：

#include < windows.h>

#pragma comment(linker, "/entry:EntryMain")

int EntryMain ()

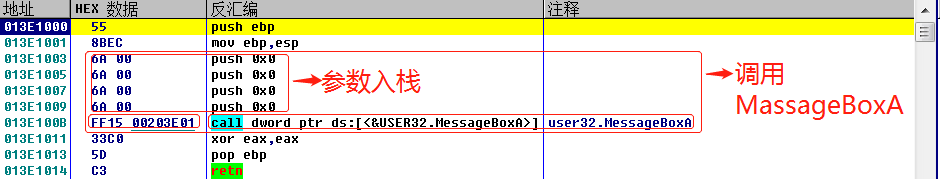
{

MessageBoxA(NULL, NULL, NULL, NULL);

return 0;

}

编译生成exe文件，使用OD打开exe文件



可以从上图中看出来，在调用MessageBoxA时，VS在编译的过程中将其计算替换为MessageBoxA函数的绝对入口地址。如果换了别的运行环境可能地址会发生变化，所以不满足shellcode的通用性。

eg8：动态地获取API函数入口地址（GetProcAdderss、LoadLibraryA）

#include < windows.h>

#pragma comment(linker, "/entry:EntryMain")

int EntryMain ()

{

// MessageBoxA(NULL, NULL, NULL, NULL);

LPVOID lp = GetProcAddress (LoadLibraryA("user32.dll"), "MessageBoxA");

char \*pszData = "hello, world";

\_\_asm

{

push 0

push 0

push pszData # push 0

push 0

call lp

}

return 0;

}

使用汇编动态调用函数是可行的，但是比较繁琐，呆板，还可以有更好的方法。

eg9：正规的函数动态调用代码（以CreateFileA和printf来演示）

#include < windows.h>

#pragma comment(linker, "/entry:EntryMain")

int EntryMain ()

{

// CreateFileA("hello.txt", GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, 0, NULL);

typedef HANDLE (WINAPI\*FN\_CreateFileA)(

\_\_in LPCSTR lpFileName,

\_\_in DWORD dwDesiredAccess,

\_\_in DWORD dwShareMode,

\_\_in\_opt LPSECURITY\_ATTRIBUTES lpSecurityAttributes,

\_\_in DWORD dwCreationDisposition,

\_\_in DWORD dwFlagsAndAttributes,

\_\_in\_opt HANDLE hTemplateFile

);

FN\_CreateFileA fn\_CreateFileA;

fn\_CreateFileA = (FN\_CreateFileA)GetProcAddress (LoadLibraryA("kernel32.dll"), "CreateFileA");

fn\_CreateFileA("hello.txt", GENERIC\_WRITE, 0, NULL, CREATE\_ALWAYS, 0, NULL);

typedef int (\_\_cdecl \*FN\_printf)(\_In\_z\_ \_Printf\_format\_string\_ const char \* \_Format, ...);

FN\_printf fn\_printf = (FN\_printf) GetProcAddress (LoadLibraryA("msvcrt.dll"), "printf");

fn\_printf("%s","hello, world !");

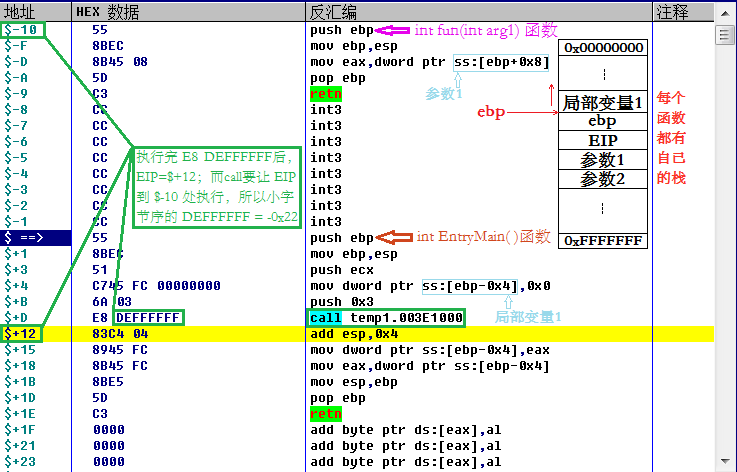
return 0;

}

先找到函数原型，然后使用typedef定义函数指针，通过函数指针动态调用函数。可以看出使用GetProcAdderss和LoadLibraryA两个函数的可以动态调用所有dll文件中的导出函数，但是，它们自己本身的地址怎么获取呢？

常用的动态链接库（可以通过PE文件查看导出表）：kernel32.dll（CreateFileA）、user32.dll（MassageBoxA）、gdi32.dll、msvcrt.dll（strcpy）、shell32.dll（ShellExecuteA）、

关于逻辑地址、虚拟地址、线性地址、物理地址

int fun(int a)

{

return a;

}

int EntryMain ()

{

int iRet = 0;

iRet = fun(3);

return iRet;

}

当调用自己写的函数数，call指令是 E8，后面跟的地址是相对偏移，当调用的是系统API函数的时，call指令是 FF15，后面跟的地址是在程序运行的时候，系统动态填进去的。DLL文件中的函数没有私有空间，系统将其物理地址映射到线程的私有空间（虚拟地址）中。举例，当调用MessageBoxA( )函数时，该函数在系统的物理地址是不变的，但在不同程序里调用它时，call 指令 FF15后面跟的地址不一样，说明不是物理地址，应该是线性地址。

call可以分为：（对应的十六进制操作码不同）

（1）near relative call：使用相对地址，**段内直接近调用**

（2）near absolute call：**段内间接近调用**

（3）...

# 第五讲：shellcode编写原则三

3.获取GetProcAddress和LoadLibrary两个函数自身的地址

GetProcAddress地址获取；

在获取到GetProcAddress函数地址之前，不能直接调用任何的dll库函数。

有些人会发现，假设已经获取到了GetProcAddress函数的地址，其实LoadLibraryA函数的地址可以通过下面的方式来获取。

LPVOID lp = GetProcAddress(LoadLibraryA("Kernel32.dll"), "LoadLibraryA");

但其实，仔细想一想这种方式在shellcode中是行不通的，是自相矛盾。

kernel32.dll基址的获取：

在windows操作系统中，不管我们的程序是否有加载kernel32.dll、。。。，系统在启动的时候都会将它们加载到内存中。

PCB：进程控制块，保存的是进程内的数据

PEB：

TCB：

TEB：

32位操作系统中，每个进程（不是线程）独享4G虚拟内存（通过映射表映射到实际的物理内存），同样的一个物理内存地址，在不同的进程中的虚拟内存地址可能是不一样的。

# 第六讲：shellcode编写原则四

# 第七讲：第一种shellcode编写实例

# 第八讲：单文件函数生成位置规律

# 第九讲：多文件函数生成位置规律

# 第十讲：编写shellcode加载器