Windows环境下shellcode编写

本文讲述怎么在Windows下编写shellcode，以弹出一个简单的消息框shellcode为例.分别针对特殊环境下的shellcode以及通用shellcode.

# 1.特殊环境下编写shellcode

|  |  |  |
| --- | --- | --- |
|  | 环境 | 备注 |
| 操作系统 | Windows XP Pro sp3 | 简体中文专业版 |
| 调试器 | Ollydbg |  |
| 编译器 | Visual C++ 6.0 |  |

先来认识两个实验所需要的函数：

1. MessageBox

MSDN 对这个函数的解释如下：  
int MessageBox(  
HWND hWnd, // handle to owner window  
LPCTSTR lpTex, // text in message box  
LPCTSTR lpCaption, // message box title  
UINT uType // message box style  
);

hWnd：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

1. exit

exit(0)表示正常退出

本文shellcode功能就是弹出一个显示I am moonagirl的消息框然后正常退出。

写出的 shellcode 的源代码如下所示：

#include <windows.h>

int main()

{

HINSTANCE LibHandle0,LibHandle1;

char dllbuf0[11] = "user32.dll";

char dllbuf1[13] = "kernel32.dll";

LibHandle0 = LoadLibrary(dllbuf0);

LibHandle1 = LoadLibrary(dllbuf1);

\_asm{

xor ebx,ebx

push ebx // cut string

push 0x20206c72

push 0x6967616e

push 0x6f6f6d20

push 0x6d612069 //push i am moonagirl

mov eax,esp

push ebx // Type

push eax //

push eax // Text

push ebx // hWnd

mov eax,0x77D507EA // address of messageboxA

call eax

push ebx

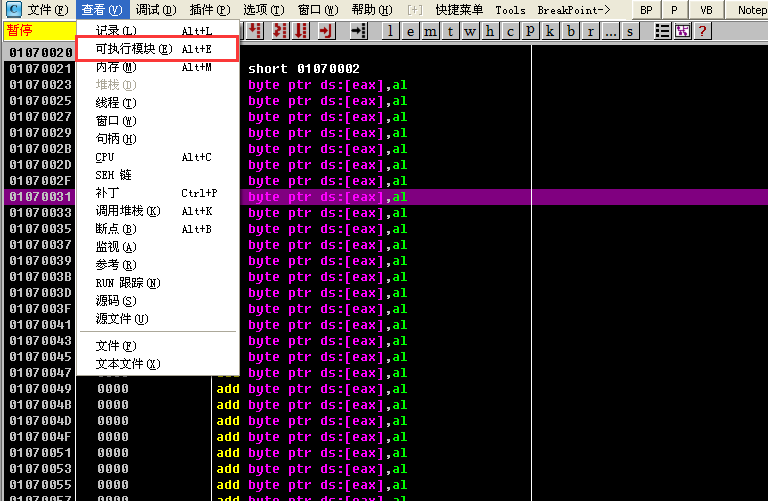
mov eax,0x7c81CAFA //address of ExitProcess

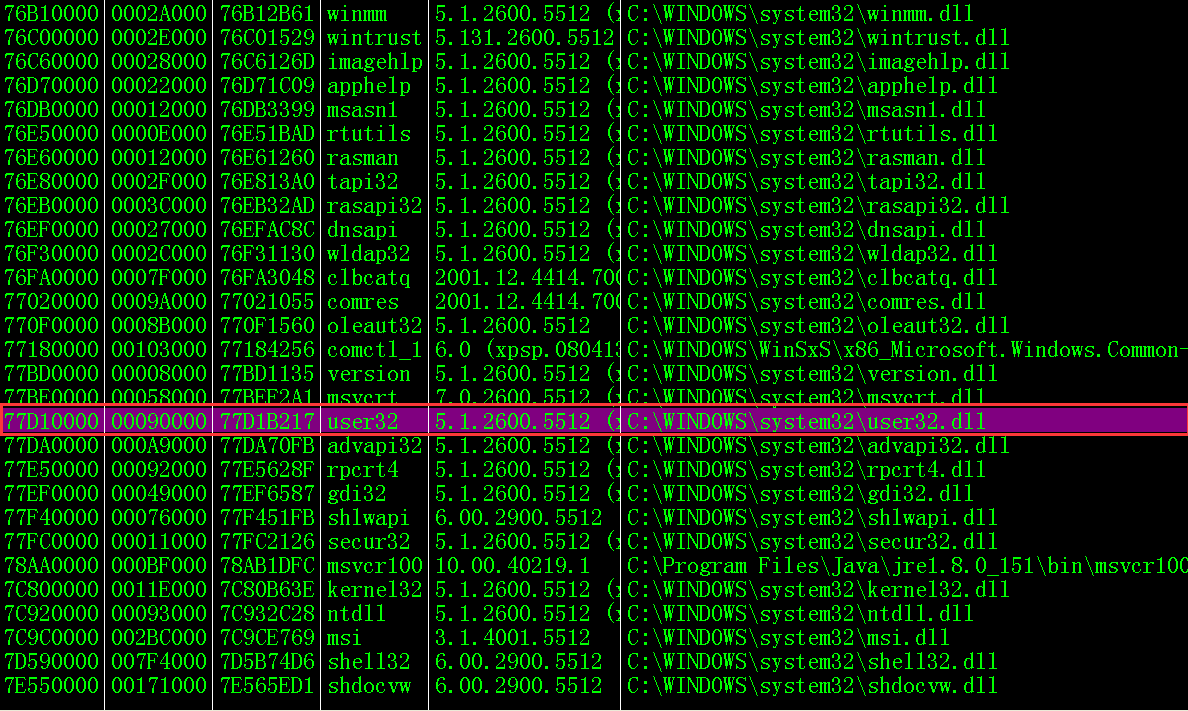
call eax // FFD0

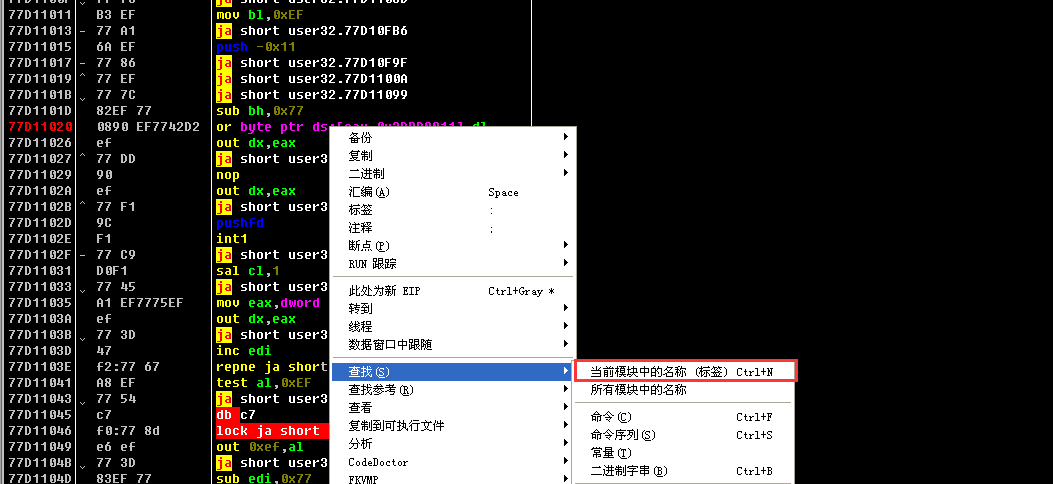
}

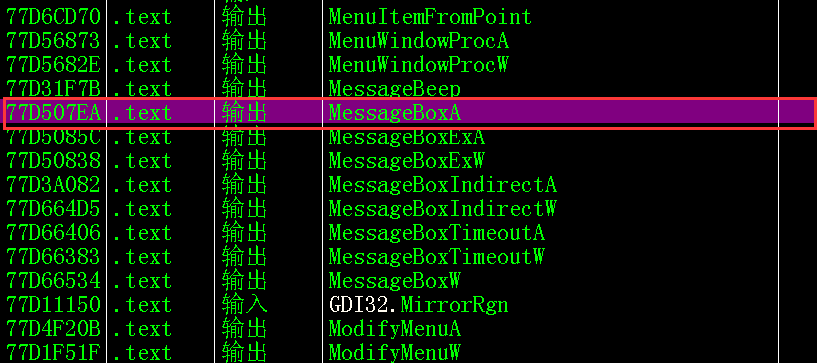
}

其中messageboxA的地址是=可以通过Ollydbg找到，先用Ollydbg附加IE浏览器进程，然后在可执行模块user32.dll里找到messageboxA函数地址，如下图



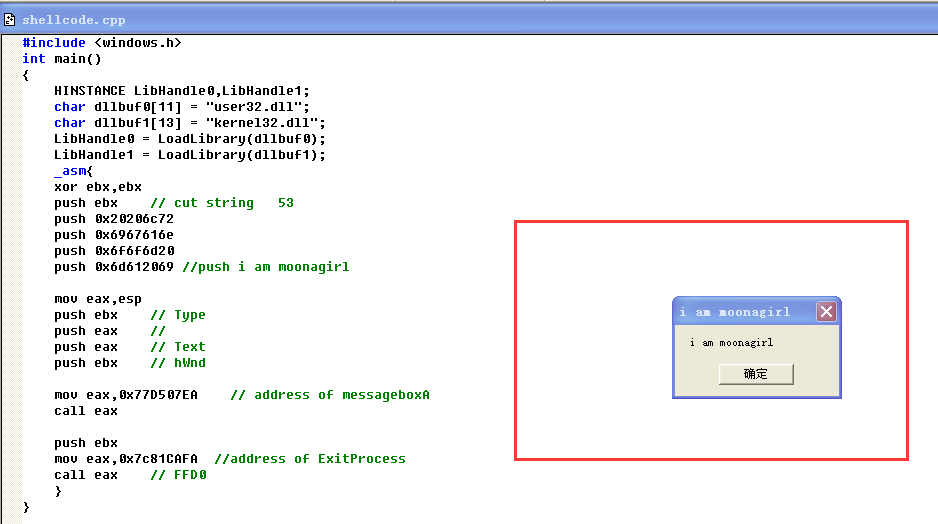






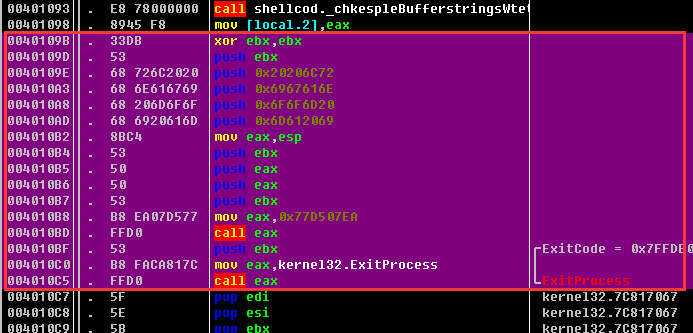
同理ExitProcess函数地址可在可执行模块kernel32.dll中找到。

编译源程序



找到可执行文件shellcode.exe用Ollydbg打开，找到弹出消息框部分汇编代码

将机器码保存下来就是我们要的shellcode了。



Shellcode =

“\x33\xDB\x53\x68\x72\x6C\x20\x20\x68\x6E\x61\x67\x69\x68\x20\x6D\x6F\x6F\x68\x69\x20\x61\x6D\x8B\xC4\x53\x50\x50\x53\xB8\xEA\x07\xD5\x77\xFF\xD0\x53\xB8\xFA\xCA\x81\x7C\xFF\xD0”

用下面这段c++程序验证shellcode

#include<windows.h>

unsigned char Shellcode[] =

"\x33\xDB\x53\x68\x72\x6C\x20\x20\x68\x6E\x61\x67\x69\x68\x20\x6D\x6F\x6F\x68\x69\x20\x61\x6D\x8B\xC4\x53\x50\x50\x53\xB8\xEA\x07\xD5\x77\xFF\xD0\x53\xB8\xFA\xCA\x81\x7C\xFF\xD0";

int main()

{

HINSTANCE LibHandle0,LibHandle1;

char dllbuf0[11] = "user32.dll";

char dllbuf1[13] = "kernel32.dll";

LibHandle0 = LoadLibrary(dllbuf0);

LibHandle1 = LoadLibrary(dllbuf1);

((void(\*)(void))&Shellcode)();

return 0;

}

编译运行，证明shellcode可行



# 2.未知环境下编写通用shellcode

不同的操作系统版本，不同的补丁版本的API 地址存在差异，使用静态函数地址来调用 API 会使 exploit的通用性受到很大限制。所以，实际中使用的 shellcode必须还要能动态地获得自身所需的 API 函数地址。

Windows 的 API 是通过动态链接库中的导出函数来实现的，例如，内存操作等函数在  
kernel32.dll 中实现；大量的图形界面相关的 API 则在 user32.dll 中实现。

Win\_32 平台下的shellcode 使用最广泛的方法，就是通过从进程环境块中找到动态链接库的导出表，并搜索出所需的 API 地址，然后逐一调用。

所有 win\_32 程序都会加载 ntdll.dll 和 kernel32.dll 这两个最基础的动态链接库。如果想要在 win\_32 平台下定位 kernel32.dll 中的 API 地址，可以采用如下方法。

（ 1）首先通过段选择字 FS 在内存中找到当前的线程环境块 TEB。  
（ 2）线程环境块偏移位置为 0x30 的地方存放着指向进程环境块 PEB 的指针。  
（ 3）进程环境块中偏移位置为 0x0C 的地方存放着指向 PEB\_LDR\_DATA 结构体的指针，  
其中，存放着已经被进程装载的动态链接库的信息。  
（ 4） PEB\_LDR\_DATA 结构体偏移位置为 0x1C 的地方存放着指向模块初始化链表的头指  
针 InInitializationOrderModuleList。  
（ 5）模块初始化链表 InInitializationOrderModuleList 中按顺序存放着 PE 装入运行时初始化  
模块的信息，第一个链表结点是 ntdll.dll，第二个链表结点就是 kernel32.dll。  
（ 6）找到属于 kernel32.dll 的结点后，在其基础上再偏移 0x08 就是 kernel32.dll 在内存中的  
加载基地址。  
（ 7）从 kernel32.dll 的加载基址算起，偏移 0x3C 的地方就是其 PE 头。  
（ 8） PE 头偏移 0x78 的地方存放着指向函数导出表的指针。  
（ 9）至此，我们可以按如下方式在函数导出表中算出所需函数的入口地址

z 导出表偏移 0x1C 处的指针指向存储导出函数偏移地址（ RVA）的列表。  
z 导出表偏移 0x20 处的指针指向存储导出函数函数名的列表。  
z 函数的 RVA 地址和名字按照顺序存放在上述两个列表中，我们可以在名称列表中定位  
到所需的函数是第几个，然后在地址列表中找到对应的 RVA。  
z 获得 RVA 后，再加上前边已经得到的动态链接库的加载基址，就获得了所需 API 此刻在内存中的虚拟地址，这个地址就是我们最终在 shellcode 中调用时需要的地址。

下面我们将给 shellcode 加入自动定位 API 的功能。为了实现弹出消息框并显示“I am moonagirl”的功能，需要使用如下 API 函数。  
（ 1） MessageBoxA位于 user32.dll 中，用于弹出消息框。  
（ 2） ExitProcess 位于 kernel32.dll 中，用于正常退出程序。  
（ 3） LoadLibraryA 位于 kernel32.dll 中。并不是所有的程序都会装载 user32.dll，所以在我们调用 MessageBoxA 之前，应该先使用 LoadLibrary(“ user32.dll” )装载其所属的动态链接库。

由于 shellcode 最终是要放进缓冲区的，为了让 shellcode 更加通用，能被大多数缓冲区容纳，我们总是希望 shellcode 尽可能短。因此，在函数名导出表中搜索函数名的时候，一般情况下并不会用“ MessageBoxA”这么长的字符串去进行直接比较。通常情况下，我们会对所需的 API 函数名进行 hash 运算，在搜索导出表时对当前遇到的函数名也进行同样的 hash，这样只要比较 hash 所得的摘要（ digest）就能判定是不是我们所需的 API了。虽然这种搜索方法需要引入额外的 hash 算法，但是可以节省出存储函数名字符串的代码。

本文中所用 hash 函数的 C 代码如下。

#include <stdio.h>  
#include <windows.h>  
DWORD GetHash(char \*fun\_name)  
{  
 DWORD digest=0;  
 while(\*fun\_name)  
 {  
 digest=((digest<<25)|(digest>>7)); //循环右移 7 位  
 digest+= \*fun\_name ; //累加  
 fun\_name++;  
 }  
 return digest;  
}  
main()  
{  
 DWORD hash;  
 hash= GetHash("AddAtomA");  
 printf("result of hash is %.8x\n",hash);  
}

在将 hash 压入栈中之前，注意先将增量标志 DF 清零。因为当 shellcode 是利用异常处理  
机制而植入的时候，往往会产生标志位的变化，使 shellcode 中的字串处理方向发生变化而产生错误（如指令 LODSD）。如果您在堆溢出利用中发现原本身经百战的 shellcode 在运行时出错，很可能就是这个原因。总之，一个字节的指令可以大大增加 shellcode 的通用性。

;store hash  
push 0x1e380a6a ;hash of MessageBoxA  
push 0x4fd18963 ;hash of ExitProcess  
push 0x0c917432 ;hash of LoadLibraryA  
mov esi,esp ;esi = addr of first function hash  
lea edi,[esi-0xc] ;edi = addr to start writing function

然后我们需要抬高栈顶，保护 shellcode 不被入栈数据破坏。

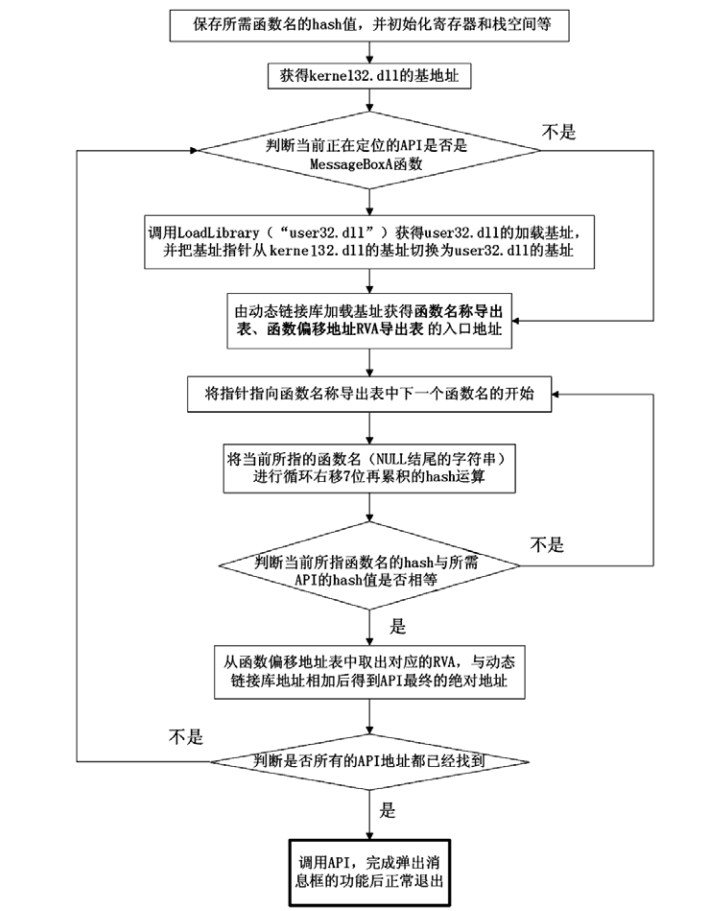
;make some stack space  
xor ebx,ebx  
mov bh, 0x04  
sub esp, ebx

定位 kernel32.dll 的代码如下

;find base addr of kernel32.dll  
mov ebx, fs:[edx + 0x30] ;ebx = address of PEB  
mov ecx, [ebx + 0x0c] ;ecx = pointer to loader data  
mov ecx, [ecx + 0x1c] ;ecx = first entry in initialisation  
;order list  
mov ecx, [ecx] ;ecx = second entry in list  
;(kernel32.dll)

mov ebp, [ecx + 0x08] ;ebp = base address of kernel32.dll

在导入表中搜索 API 的逻辑可以设计如下图所示



最终的代码实现如下

#include<windows.h>

int main()

{

\_asm{

CLD ;clear flag DF

;store hash

push 0x1e380a6a ;hash of MessageBoxA

push 0x4fd18963 ;hash of ExitProcess

push 0x0c917432 ;hash of LoadLibraryA

mov esi,esp ;esi = addr of first function hash

lea edi,[esi-0xc] ;edi = addr to start writing function

;make some stack space

xor ebx,ebx

mov bh, 0x04

sub esp, ebx

;push a pointer to "user32" onto stack

mov bx, 0x3233 ;rest of ebx is null

push ebx

push 0x72657375

push esp

xor edx,edx

;find base addr of kernel32.dll

mov ebx, fs:[edx + 0x30] ;ebx = address of PEB

mov ecx, [ebx + 0x0c] ;ecx = pointer to loader data

mov ecx, [ecx + 0x1c] ;ecx = first entry in initialization

;order list

mov ecx, [ecx] ;ecx = second entry in list

;(kernel32.dll)

mov ebp, [ecx + 0x08] ;ebp = base address of kernel32.dll

find\_lib\_functions:

lodsd ;load next hash into al and increment esi

cmp eax, 0x1e380a6a ;hash of MessageBoxA - trigger

;LoadLibrary("user32")

jne find\_functions

xchg eax, ebp ;save current hash

call [edi - 0x8] ;LoadLibraryA

xchg eax, ebp ;restore current hash, and update ebp

;with base address of user32.dll

find\_functions:

pushad ;preserve registers

mov eax, [ebp + 0x3c] ;eax = start of PE header

mov ecx, [ebp + eax + 0x78] ;ecx = relative offset of export table

add ecx, ebp ;ecx = absolute addr of export table

mov ebx, [ecx + 0x20] ;ebx = relative offset of names table

add ebx, ebp ;ebx = absolute addr of names table

xor edi, edi ;edi will count through the functions

next\_function\_loop:

inc edi ;increment function counter

mov esi, [ebx + edi \* 4] ;esi = relative offset of current

;function name

add esi, ebp ;esi = absolute addr of current

;function name

cdq ;dl will hold hash (we know eax is

;small)

hash\_loop:

movsx eax, byte ptr[esi]

cmp al,ah

jz compare\_hash

ror edx,7

add edx,eax

inc esi

jmp hash\_loop

compare\_hash:

cmp edx, [esp + 0x1c] ;compare to the requested hash (saved on

;stack from pushad)

jnz next\_function\_loop

mov ebx, [ecx + 0x24] ;ebx = relative offset of ordinals

;table

add ebx, ebp ;ebx = absolute addr of ordinals

;table

mov di, [ebx + 2 \* edi] ;di = ordinal number of matched

;function

mov ebx, [ecx + 0x1c] ;ebx = relative offset of address

;table

add ebx, ebp ;ebx = absolute addr of address table

add ebp, [ebx + 4 \* edi] ;add to ebp (base addr of module) the

;relative offset of matched function

xchg eax, ebp ;move func addr into eax

pop edi ;edi is last onto stack in pushad

stosd ;write function addr to [edi] and

;increment edi

push edi

popad ;restore registers

;loop until we reach end of last hash

cmp eax,0x1e380a6a

jne find\_lib\_functions

function\_call:

xor ebx,ebx

push ebx ;cut string

push 0x20206c72

push 0x6967616e

push 0x6f6f6d20

push 0x6d612069 //push i am moonagirl

mov eax,esp ;load address of failwest

push ebx

push eax

push eax

push ebx

call [edi - 0x04] ;call MessageboxA

push ebx

call [edi - 0x08] ;call ExitProcess

nop

nop

nop

nop

}

}

这样，一段考虑了跨平台、健壮性、稳定性、通用性等各方面因素的高质量 shellcode 就生  
成了。



可以看出即使是经验丰富的汇编程序员，想要写出高质量的 shellcode也得着实花一翻工夫。事实上，若非真的有特殊需要，即使是经验丰富的 hacker 也不会总是自己编写 shellcode。大多数情况下，从 Internet 上可以得到许多经典的 shellcode。另外MetaSploit通用漏洞测试架构 3.0 下的 payload 库中，目前已经包含了包括绑定端口、网马 downloader、远程 shell、任意命令执行等在内的 104 种不同功能的经典 shellcode。通过简单的参数配置，可以轻易导出 C 语言格式、 Perl 语言格式、 ruby 语言格式、原始十六进制格式等形式的 shellcode。

char popup\_general[]=  
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"  
"\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"  
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"  
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"  
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"  
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"  
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"  
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"  
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"  
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"  
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8";  
void main()  
{  
\_\_asm  
{  
lea eax, popup\_general  
push eax  
ret  
}  
}