讲到第三章我又压力山大了，相信各位看官都是冲着这一章和下一章来的，如果写不好的话肯定要被各位看官拍死。好，废话不多说，转入正题。在开始正式讲 HOOK 之前，先把 WIN64 的系统调用说清楚。

WIN64 的系统调用比 WIN32 要复杂很多，原因很简单，因为 WIN64 系统可以运行两种 EXE，而且 WIN32EXE 的执行效率并不差（据我本人用 3DMARK06 实测，在一台电脑上分别安装 WIN7X86 和 WIN7X64，使用同样版本的显卡驱动，3DMARK06 在 WIN7X86 的系统得分比在 WIN7X64 系统的得分高 3%左右，性能损失还算少），因此判断出 WIN32EXE 在 WIN64 系统上绝对不是模拟执行的，而是经过了某种转换后直接执行。在本文中，先讲解 WIN64 进程（或称 64 位进程）的系统函数的执行过程，再讲解 WOW64 进程（或称 32 位进程）的系统函数的执行过程。

一、WIN64 进程的系统函数执行流程

以 NtCreateFile 为例，首先对 ntdll!NtCreateFile 进行反汇编：

```
lkd> uf NtCreateFile
ntdll!NtCreateFile:
00000000`77250400 4c8bd1          mov     r10,rcx
00000000`77250403 b852000000      mov     eax,52h
00000000`77250408 0f05            syscall
00000000`7725040a c3              ret
```

ntdll!NtCreateFile 没有像 Win32 一样经过 ntdll!KiFastSystemCall 等麻烦步骤，直接通过 syscall 指令进入内核（多说一句，ntdll!ZwCreateFile 和 ntdll!NtCreateFile 的反汇编代码是一样的，也就是说跟 Win32 一样，ntdll!ZwCreateFile 和 ntdll!NtCreateFile 是同一个函数，除了名字不相同外）。在第二句反汇编代码中，0x52 是 ZwCreateFile 在 SSDT 中的编号。接下来看看内核中的 ZwCreateFile 调用了什么：

```
lkd> uf nt!ZwCreateFile
nt!ZwCreateFile:
fffff800`040cdf00 488bc4          mov     rax,rsp
fffff800`040cdf03 fa              cli
fffff800`040cdf04 4883ec10        sub     rsp,10h
fffff800`040cdf08 50              push    rax
fffff800`040cdf09 9c              pushfq
fffff800`040cdf0a 6a10            push    10h
fffff800`040cdf0c 488d05dd270000  lea     rax,[nt!KiServiceLinkage (fffff800`040d06f0)]
fffff800`040cdf13 50              push    rax
fffff800`040cdf14 b852000000      mov     eax,52h
fffff800`040cdf19 e9225f0000      jmp     nt!KiServiceInternal (fffff800`040d3e40)

nt!KiServiceInternal:
fffff800`040d3e40 4883ec08        sub     rsp,8
fffff800`040d3e44 55              push    rbp
fffff800`040d3e45 4881ec58010000  sub     rsp,158h
```

```
fffff800`040d3e4c 488dac2480000000 lea     rbp,[rsp+80h]
fffff800`040d3e54 48899dc0000000  mov     qword ptr [rbp+0C0h],rbx
fffff800`040d3e5b 4889bdc8000000  mov     qword ptr [rbp+0C8h],rdi
fffff800`040d3e62 4889b5d0000000  mov     qword ptr [rbp+0D0h],rsi
fffff800`040d3e69 fb              sti
fffff800`040d3e6a 65488b1c2588010000 mov   rbx,qword ptr gs:[188h]
fffff800`040d3e73 0f0d8bd8010000  prefetchw [rbx+1D8h]
fffff800`040d3e7a 0fb6bbf6010000  movzx   edi,byte ptr [rbx+1F6h]
fffff800`040d3e81 40887da8        mov     byte ptr [rbp-58h],dil
fffff800`040d3e85 c683f601000000  mov     byte ptr [rbx+1F6h],0
fffff800`040d3e8c 4c8b93d8010000  mov     r10,qword ptr [rbx+1D8h]
fffff800`040d3e93 4c8995b8000000  mov     qword ptr [rbp+0B8h],r10
fffff800`040d3e9a 4c8d1d3d010000  lea     r11,[nt!KiSystemServiceStart (fffff800`040d3fde)]
fffff800`040d3ea1 41ffe3          jmp     r11
```

ZwCreateFile 调用了 KiServiceLinkage，把系统服务序号放进 eax 后又调用了 KiServiceInternal，KiServiceInternal 又调用了 KiSystemServiceStart。KiServiceLinkage 和 KiServiceInternal 是初始化系统服务的，KiSystemServiceStart 则是开始执行系统服务。再看看 KiSystemServiceStart 干了什么：

```
lkd> uf KiSystemServiceStart
nt!KiSystemServiceStart:
fffff800`040d3fde 4889a3d8010000  mov     qword ptr [rbx+1D8h],rsp
fffff800`040d3fe5 8bf8            mov     edi,eax
fffff800`040d3fe7 c1ef07          shr     edi,7
fffff800`040d3fea 83e720          and     edi,20h
fffff800`040d3fed 25ff0f0000      and     eax,0FFFh
nt!KiSystemServiceRepeat:
fffff800`040d3ff2 4c8d1547782300  lea     r10,[nt!KeServiceDescriptorTable
(fffff800`0430b840)]
fffff800`040d3ff9 4c8d1d80782300  lea     r11,[nt!KeServiceDescriptorTableShadow
(fffff800`0430b880)]
fffff800`040d4000 f7830001000080000000 test dword ptr [rbx+100h],80h
fffff800`040d400a 4d0f45d3        cmovne  r10,r11
fffff800`040d400e 423b441710      cmp     eax,dword ptr [rdi+r10+10h]
fffff800`040d4013 0f83e9020000    jae     nt!KiSystemServiceExit+0x1a7 (fffff800`040d4302)

//以下反汇编代码省略
```

KiSystemServiceStart 调用了 KiSystemServiceRepeat，KiSystemServiceRepeat 根据系统服务序号来选择 SSDT 还是 Shadow SSDT（到了 KiSystemServiceRepeat 才真正开始调用 Nt***函数）。KiSystemServiceRepeat 执行完毕后，调用了 KiSystemServiceExit(系统服务调用完毕，它会带上 Nt***函数的返回值等信息)：

```
lkd> u KiSystemServiceExit
nt!KiSystemServiceExit:
fffff800`040d415b 488b9dc0000000  mov     rbx,qword ptr [rbp+0C0h]
fffff800`040d4162 488bbdc8000000  mov     rdi,qword ptr [rbp+0C8h]
fffff800`040d4169 488bb5d0000000  mov     rsi,qword ptr [rbp+0D0h]
fffff800`040d4170 654c8b1c2588010000 mov   r11,qword ptr gs:[188h]
fffff800`040d4179 f685f000000001  test    byte ptr [rbp+0F0h],1
fffff800`040d4180 0f844f010000    je      nt!KiSystemServiceExit+0x17a (fffff800`040d42d5)
fffff800`040d4186 440f20c1        mov     rcx,cr8
fffff800`040d418a 410a8bf0010000  or      cl,byte ptr [r11+1F0h]
```

总结一下，WIN64 进程系统函数的执行流程是：ntdll!ZwXXX ->(syscall 进内核)-> nt!ZwXXX -> nt!KiServiceInternal -> nt!KiSystemServiceStart -> nt!NtXXX -> KiSystemServiceExit ->(返回)-> nt!ZwXXX ->(返回)-> ntdll!ZwXXX。

二、WOW64 进程的系统函数执行流程

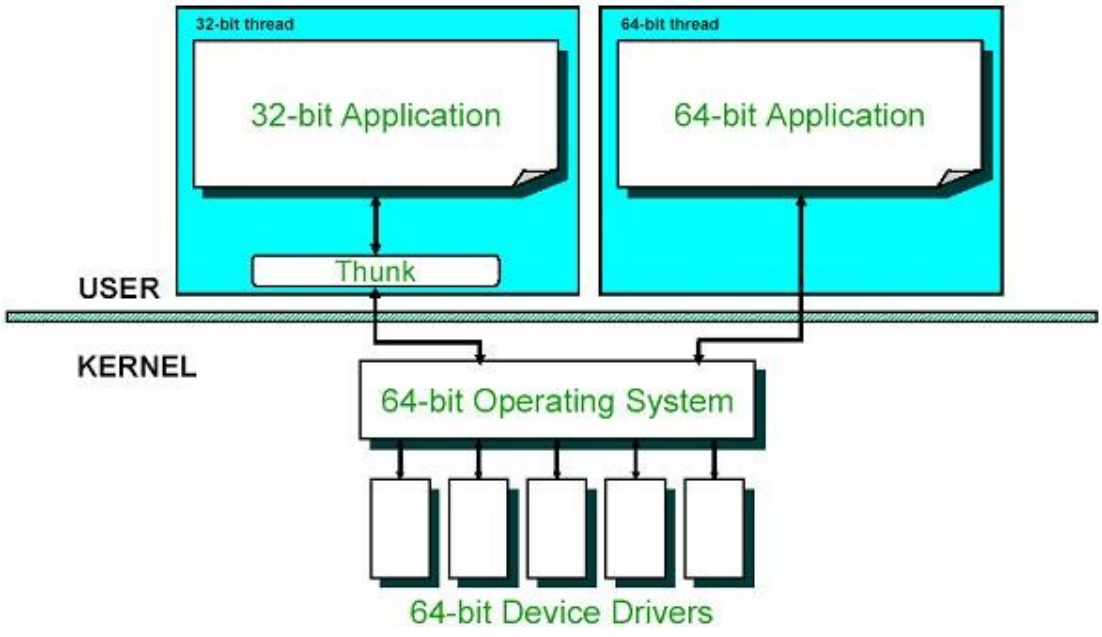
首先用 WINDBG 打开任意 WIN32EXE 进行调试，看看模块加载列表：

```
Executable search path is:
ModLoad: 00000000`008f0000 00000000`009b0000   calc.exe
ModLoad: 00000000`770c0000 00000000`77269000   ntdll.dll
ModLoad: 00000000`772a0000 00000000`77420000   ntdll32.dll
ModLoad: 00000000`74d00000 00000000`74d3f000   C:\Windows\SYSTEM32\wow64.dll
ModLoad: 00000000`74ca0000 00000000`74cfc000   C:\Windows\SYSTEM32\wow64win.dll
ModLoad: 00000000`74c90000 00000000`74c98000   C:\Windows\SYSTEM32\wow64cpu.dll
(1190.1124): Break instruction exception - code 80000003 (first chance)
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll.dll -
ntdll!CsrSetPriorityClass+0x40:
00000000`7716cb60 cc              int     3
0:000> g
ModLoad: 00000000`76fa0000 00000000`770bf000   WOW64_IMAGE_SECTION
ModLoad: 00000000`75440000 00000000`75550000   WOW64_IMAGE_SECTION
ModLoad: 00000000`76fa0000 00000000`770bf000   NOT_AN_IMAGE
ModLoad: 00000000`76ea0000 00000000`76f9a000   NOT_AN_IMAGE
ModLoad: 00000000`75440000 00000000`75550000   C:\Windows\syswow64\kernel32.dll
ModLoad: 00000000`750f0000 00000000`75136000   C:\Windows\syswow64\KERNELBASE.dll
ModLoad: 00000000`76250000 00000000`76e9a000   C:\Windows\syswow64\SHELL32.dll
ModLoad: 00000000`75550000 00000000`755fc000   C:\Windows\syswow64\msvcrt.dll
ModLoad: 00000000`75840000 00000000`75897000   C:\Windows\syswow64\SHLWAPI.dll
ModLoad: 00000000`75c50000 00000000`75ce0000   C:\Windows\syswow64\GDI32.dll
ModLoad: 00000000`76110000 00000000`76210000   C:\Windows\syswow64\USER32.dll
ModLoad: 00000000`753a0000 00000000`75440000   C:\Windows\syswow64\ADVAPI32.dll
//以下内容省略
```

可以看出，加载了 CALC.EXE 之后，首先就是加载 64 位的 NTDLL，然后才加载 32 位的 NTDLL，然后加载几个模式转换的 DLL：WOW64.DLL、WIN64WIN.DLL、WOW64CPU.DLL。关于这几个 DLL，网上没有什么公开的研究资料，唯一能找到的资料就是一篇叫做《Mixing x86 with x64 code》的博文。另外，如果大家对这几个 DLL 感兴趣，可以去看一下这篇文章原作者的博客：http://blog.rewolf.pl。接下来，对 ntdll32!NtCreateFile 进行反汇编：

```
ntdll32!ZwCreateFile:
00000000`772c00a4 b852000000      mov     eax,52h
00000000`772c00a9 33c9            xor     ecx,ecx
00000000`772c00ab 8d542404        lea     edx,[rsp+4]
00000000`772c00af  64ff15c0000000    call        qword  ptr  fs:[ntdll32!ZwCancelIoFile+0xa
(00000000`772c0176)]
00000000`772c00b6 83c404          add     esp,4
00000000`772c00b9 c22c00          ret     2Ch
```

结果让人失望，我的电脑上无法把符号表加载完全，看到了一串莫名其妙的东西：ntdll32!ZwCancelIoFile+0xa，但据查资料，知道这货是一个函数：wow64cpu!X86SwitchTo64BitMode。这就是说从这里开始，线程从 32 位切换到了 64 位，那么接下来，应该执行 ntdll!NtCreateFile。执行完毕后，又调用 wow64cpu!CpupReturnFromSimulatedCode，把线程从 64 位切换到了 32 位。这也就解释了为什么 32 位进程在 WIN64 系统上有性能损耗的原因：浪费的性能都用在转换上了。最后附上一张图作为总结（Thunk 其实就是刚才提到的那三个 DLL）：



三、WOW64 进程与 WIN64 进程在运行时的不同之处

一言以蔽之，就是 WOW64 进程在对某些目录和注册表项进行读写时，会被系统重定位。WOW64 进程只有访问两个目录才会被重定向：%Program Files%和%System32%。简而言之，就是一个 WOW64 进程试图在 C:\Program Files 和 C:\Windows\System32 下创建文件或文件夹时，会被重定向到 C:\Program Files

(x86)和 C:\Windows\SysWow64 创建。而注册表的重定位就太多了，具体请参见：http://msdn.microsoft.com/en-us/library/aa384253(v=VS.85).aspx。比如访问 HKEY_LOCAL_MACHINE\Software\test 的时候，会重定向到 HKEY_LOCAL_MACHINE\Software\Wow6432Node\test。当然，你可以选择拒绝被重定向。关闭文件重定向可以使用 Wow64EnableWow64FsRedirection，关闭注册表重定向更加简单，只要调用 RegCreateKeyEx 或 RegOpenKeyEx 时在 samDesired 参数上增加一个常量 KEY_WOW64_64KEY 即可。示例代码如下：

```
//注册表重定向例子
VOID RegRedirectionTest()
{
    HKEY hkey,us;
    printf("Create key [test] in [HKEY_LOCAL_MACHINE\\Software\\Wow6432Node]!\n");
    RegOpenKeyExA(HKEY_LOCAL_MACHINE,"Software",0,KEY_ALL_ACCESS, &hkey);
    RegCreateKeyA(hkey,"test",&us);
    getchar();
    printf("Create key [test] in [HKEY_LOCAL_MACHINE\\Software]!\n");
    RegOpenKeyExA(HKEY_LOCAL_MACHINE,"Software",0,KEY_ALL_ACCESS | KEY_WOW64_64KEY,
&hkey);
    RegCreateKeyA(hkey,"test",&us);
    getchar();
}
```

```
//文件重定向例子
VOID FsRedirectionTest()
{
    Wow64EnableWow64FsRedirection =
(WOW64ENABLEWOW64FSREDIRECTION)GetProcAddress(LoadLibraryA("kernel32.dll"),"Wow64EnableWo
w64FsRedirection");
    printf("Create folder [test] in [C:\\WINDOWS\\SYSWOW64]!\n");
    _mkdir("c:\\windows\\system32\\test");
    getchar();
    Wow64EnableWow64FsRedirection(FALSE); //Close Redirection
    printf("Create folder [test] in [C:\\WINDOWS\\SYSTEM32]!\n");
    _mkdir("c:\\windows\\system32\\test");
    getchar();
}
```

在使用 Wow64EnableWow64FsRedirection(FALSE)前，文件夹实际创建到了 C:\WINDOWS\SysWow64 目录；使用后，文件夹才创建到 C:\WINDOWS\System32 目录。当使用 Wow64EnableWow64FsRedirection(TRUE)后，会再次出现重定向效果。

接下来说说 32 位程序怎么检测自己是否运行在 WIN64 系统上。微软的官方方案是使用 kernel32!IsWow64Process（这个函数在 XP SP2 以后才有）：

```
//代码来自：http://msdn.microsoft.com/en-us/library/ms684139(VS.85).aspx
#include <windows.h>
```

```
#include <tchar.h>

typedef BOOL (WINAPI *LPFN_ISWOW64PROCESS) (HANDLE, PBOOL);

LPFN_ISWOW64PROCESS fnIsWow64Process;

BOOL IsWow64()
{
    BOOL bIsWow64 = FALSE;

    //IsWow64Process is not available on all supported versions of Windows.
    //Use GetModuleHandle to get a handle to the DLL that contains the function
    //and GetProcAddress to get a pointer to the function if available.

    fnIsWow64Process = (LPFN_ISWOW64PROCESS) GetProcAddress(
        GetModuleHandle(TEXT("kernel32")),"IsWow64Process");

    if(NULL != fnIsWow64Process)
    {
        if (!fnIsWow64Process(GetCurrentProcess(),&bIsWow64))
        {
            //handle error
        }
    }
    return bIsWow64;
}

int main( void )
{
    if(IsWow64())
        _tprintf(TEXT("The process is running under WOW64.\n"));
    else
        _tprintf(TEXT("The process is not running under WOW64.\n"));

    return 0;
}
```

但我也有自己的一套检测方法，就是分别调用 GetSystemInfo 和 GetNativeSystemInfo。这两个函数的参数都是一样的（LPSYSTEM_INFO），但是返回的某些信息会有所不同。微软对 GetNativeSystemInfo 的解释是：

> Retrieves information about the current system to an application running under WOW64. If the function is called from a 64-bit application, it is equivalent to the GetSystemInfo function.

其中一个明显的不同就是 SYSTEM_INFO 结构体中 wProcessorArchitecture

成员的值。根据 MSDN 的解释，此值为 0 时是 IA32 体系，此值为 9 是 AMD64 体系。如果 WIN32 程序在 WIN32 系统运行，则两个值都是 0，如果在 WIN64 系统运行，则第一个函数返回 0，第二个函数返回 9。简而言之，当这两个的值不同时就认为程序是在 WIN64 系统上运行，否则认为是在 WIN32 系统上运行：

```
typedef VOID (WINAPI *GETNATIVESYSTEMINFO) (LPSYSTEM_INFO);
GETNATIVESYSTEMINFO GetNativeSystemInfo;

BOOL IsWow64_TA()
{
    SYSTEM_INFO si1;
    SYSTEM_INFO si2;

GetNativeSystemInfo=(GETNATIVESYSTEMINFO)GetProcAddress(LoadLibraryA("kernel32.dll"),"GetNativeSystemIn
fo");
    memset(&si1,0,sizeof(SYSTEM_INFO));
    memset(&si2,0,sizeof(SYSTEM_INFO));
    GetSystemInfo(&si1);
    GetNativeSystemInfo(&si2);
    if(si1.wProcessorArchitecture != si2.wProcessorArchitecture)
    {
        printf("The process is running under WOW64.\n");
        return TRUE;
    }
    else
    {
        printf("The process is not running under WOW64.\n");
        return FALSE;
    }
}
```

四、兼容模式

兼容模式与 WOW64 不是一回事，但有点类似，兼容模式是关于旧 WINDOWS 程序在新 WINDOWS 平台上运行的。通过兼容模式，十几年前的 OFFICE97 可以在 WINDOWS 7 上运行（但反过来 OFFICE2007 不能在 WINDOWS 97 上运行）。可以想象，如果没有兼容模式，将会有多少旧程序无法在新系统上运行，而新系统又会损失多少用户。

先说说兼容模式的实现。当一个程序运行在兼容模式时，系统就会给它加载不同的 DLL，保证此程序的正常运行。随便运行一个应用程序，在兼容模式与非兼容模式下，会有不同的 DLL 加载（如下图所示）。这些 DLL 藏身于 C:\WINDOWS\WINSXS 文件夹里，这个文件夹非常不引人注目，但是它非常庞大，而且我感觉它是 WINDOWS 的幕后仓库。说一个关于 WINSXS 的秘密，也许会让很多人震惊，WINDOWS 目录的 EXPLORER.EXE 和 NOTEPAD.EXE 不过是一个硬链接而已，它真身实际上在 WINSXS 文件夹里。不信？用 WINHEX 看看就知道了。

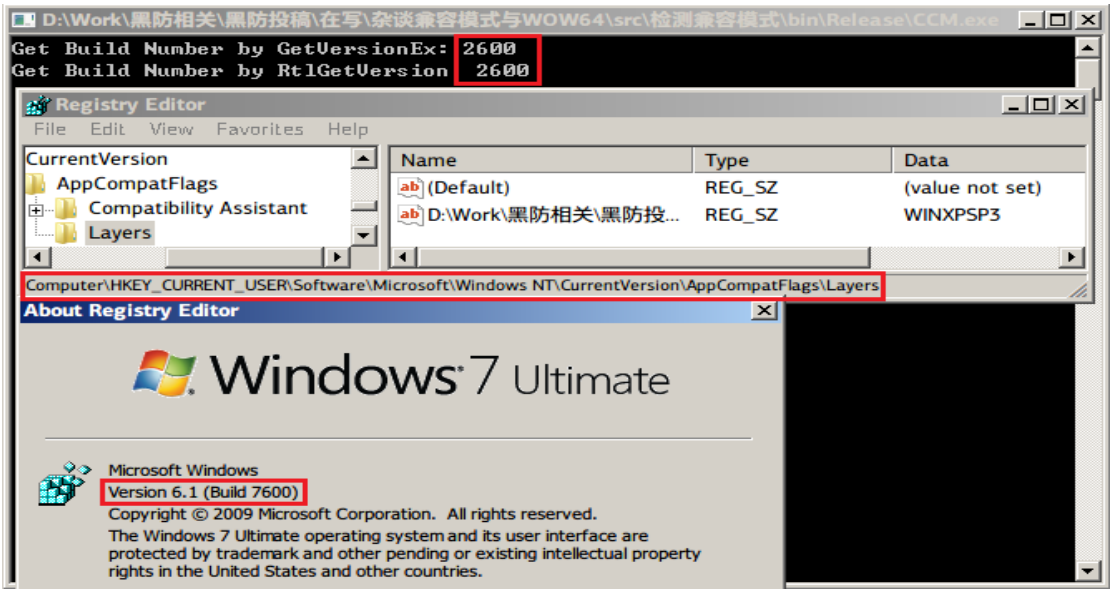
非兼容模式

Name	Description	Company Na
ADVAPI32.dll	高级 Windows 32 基	
CLBCatQ.DLL	COM+ Configuration	
COMCTL32.dll	用户体验控件库 Mic	
comdlg32.dll	Common Dialogs DLL	
CRYPTBASE.dll	Base cryptographic	
dwmapi.dll	Microsoft 桌面窗口	
GDI32.dll	GDI Client DLL Mi	
GetTrueVersion.exe	TODO: <Fil	
IMM32.DLL	Multi-User Windows	
kernel32.dll	Windows NT 基本 AP	
KERNELBASE.dll	Windows NT 基本 AP	
locale.nls		
LPK.dll	Language Pack	Microsoft
MSCTF.dll	MSCTF 服务器 DLL	

兼容模式

Name	Description	Company Name	Vers
AcGenral.DLL	Windows Compatibility DLL		
AcLayers.DLL	Windows Compatibility DLL		
AcXtrnal.DLL	Windows Compatibility DLL		
ADVAPI32.dll	高级 Windows 32 基本 API		
apphelp.dll	应用程序兼容性客户端库		M
CFGMGR32.dll	Configuration Manager DLL		
CLBCatQ.DLL	COM+ Configuration Catalog		
COMCTL32.dll	用户体验控件库 Microsoft Cor		
comdlg32.dll	Common Dialogs DLL		Micr
CRYPT32.dll	加密 API32	Microsoft	
CRYPTBASE.dll	Base cryptographic API DLL		
DEVOBJ.dll	Device Information Set DLL		
dwmapi.dll	Microsoft 桌面窗口管理器 API		
GDI32.dll	GDI Client DLL Microsoft Co		
GetTrueVersion.exe	TODO: <File descript		
iertutil.dll	Run time utility for Interne		
IMM32.DLL	Multi-User Windows IMM32 API		
kernel32.dll	Windows NT 基本 API 客户端 D		
KERNELBASE.dll	Windows NT 基本 API 客户端 D		
locale.nls			
LPK.dll	Language Pack	Microsoft Corporatio	
MPR.dll	多提供程序路由器 DLL	Microsoft Co	
MSACM32.dll	Microsoft ACM 音频筛选器		
MSASN1.dll	ASN.1 Runtime APIs		Micr
MSCTF.dll	MSCTF 服务器 DLL	Microsoft	

要设置某个程序的兼容性，就打开此程序文件的“属性”对话框，切换到“兼容性”选项卡，勾选“用兼容模式运行这个程序”复选框并选择系统版本即可。实际上，设置程序兼容性就是在注册表的[HKCU/Software/Microsoft/Windows NT/CurrentVersion/AppCompatFlags/Layers]下面建立一个键值。新建这个键值会使 kernel32!GetVersionEx 和 ntdll!RtlGetVersion 获得兼容模式中设置的系统的版本号。比如说某程序明明是在 WIN7 下运行的，但是设置了在 XP 的兼容模式下运行，结果调用 kernel32!GetVersionEx 和 ntdll!RtlGetVersion 都会得到版本号为 2600 而不是 7600。

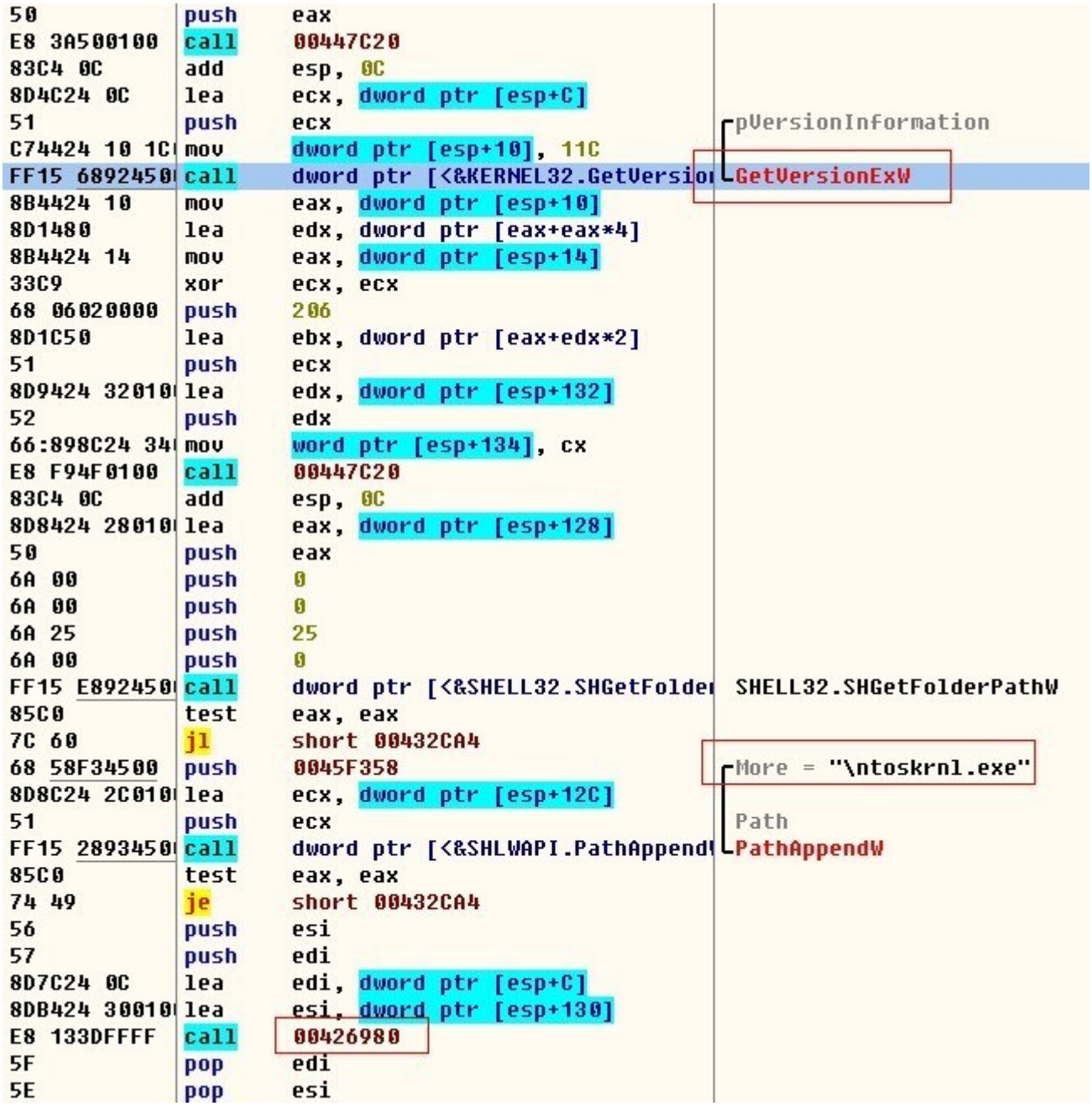


```
#include <stdio.h>
#include <windows.h>

typedef long (__stdcall *RTLGETVERSION)(POSVERSIONINFO);

int main()
{
    RTLGETVERSION
RtlGetVersion=(RTLGETVERSION)GetProcAddress(LoadLibrary("ntdll.dll"),"RtlGetVersion");
    OSVERSIONINFO osv1={0},osv2={0};
    //way 1
    osv1.dwOSVersionInfoSize=sizeof(OSVERSIONINFO);
    GetVersionEx(&osv1);
    printf("Get Build Number by GetVersionEx: %ld\n",osv1.dwBuildNumber);
    //way 2
    osv2.dwOSVersionInfoSize=sizeof(OSVERSIONINFO);
    RtlGetVersion(&osv2);
    printf("Get Build Number by RtlGetVersion: %ld\n",osv2.dwBuildNumber);
    //show info
    getchar();
    return 0;
}
```

设置程序兼容性能对抗不少安全类软件，因为不同的系统有不同的硬编码，所以安全类软件启动后的第一件事就是获取 Build Number 来判定该使用哪一套硬编码，如果 Build Number 不是任何已知的 Build Number，就退出程序。大约在两年前，通过“程序兼容性”能让 360 和微点无法启动。当然，360 和微点很快就封了这个漏洞。判断自己的程序是否运行在兼容模式下，是个比较有意义的问题。有位网友通过逆向 360，发现 360 有个巧妙的办法来判断自己是否运行在兼容模式下：先调用 GetVersionEx 得到一个版本号，譬如是 5.0（Win2000），算出一个值 50（5 * 10 + 0）；然后取出 ntoskrnl.exe 的版本号，譬如是 5.1（Win XP），再算出一个值 51（5 * 10 + 1）；对比前后两个值，如果不相等则认为自身运行在兼容模式下：



IDA 的反汇编代码如下：

```
bool __cdecl sub_432BC0()
{
  int v0; // ebx@1
  char v2; // [sp+4h] [bp-330h]@1
  unsigned int v3; // [sp+330h] [bp-4h]@1
  struct _OSVERSIONINFOW VersionInformation; // [sp+Ch] [bp-328h]@1
  WCHAR pszPath; // [sp+128h] [bp-20Ch]@1
  char v6; // [sp+12Ah] [bp-20Ah]@1
  unsigned int v7; // [sp+8h] [bp-32Ch]@4

  v3 = (unsigned int)&v2 ^ dword_46A3C4;
  sub_447C20((int)&VersionInformation.dwMajorVersion, 0, 280);
  VersionInformation.dwOSVersionInfoSize = 284;
  GetVersionExW(&VersionInformation);
  v0 = VersionInformation.dwMinorVersion + 10 * VersionInformation.dwMajorVersion;
  pszPath = 0;
  sub_447C20((int)&v6, 0, 518);
  return SHGetFolderPathW(0, 37, 0, 0, &pszPath) >= 0
      && PathAppendW(&pszPath, L"\\ntoskrnl.exe")
      && sub_426980()
      && v0 < (signed int)((unsigned __int16)v7 + 10 * (v7 >> 16));
}
```

但是在测试的过程中，发生了一件极其诡异的事情，不得不说一下。拥有计算机基础知识的人都知道，**如果把一份代码“按原样”翻译成另外一种语言的代码，编译后的执行结果应该是相同的**。但是我把上述代码翻译成 VB 代码并编译出 EXE 后，无论怎么设置兼容性，得到的结果都是真实系统的 Build Number，而不是被兼容系统的 Build Number。这件事情实在太过离奇，我绞尽脑汁也没搞明白是怎么回事。

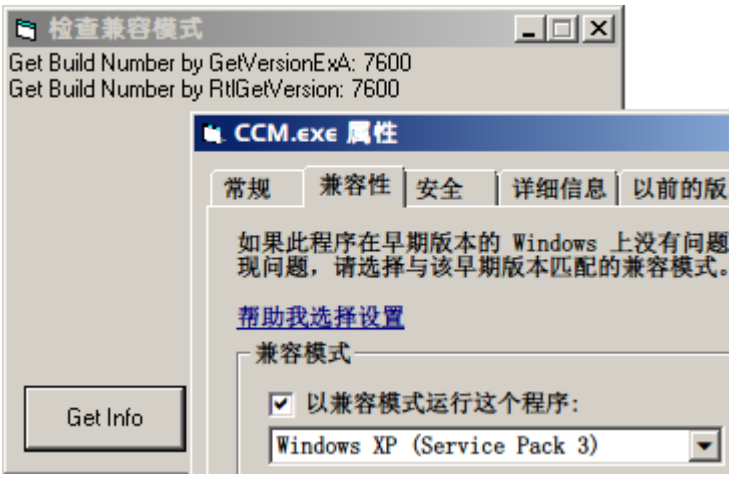
```
Option Explicit


Private Type OSVERSIONINFO
    dwOSVersionInfoSize As Long
    dwMajorVersion As Long
    dwMinorVersion As Long
    dwBuildNumber As Long
    dwPlatformId As Long
    szCSDVersion As String * 128 ' Maintenance string for PSS usage
End Type


Private Declare Function GetVersionExA Lib "kernel32.dll" (lpVersionInformation As OSVERSIONINFO) As
Long
Private Declare Function RtlGetVersion Lib "ntdll.dll" (lpVersionInformation As OSVERSIONINFO) As Long


Private Sub Command1_Click()
    Dim osv1 As OSVERSIONINFO, osv2 As OSVERSIONINFO
    'way 1
    osv1.dwOSVersionInfoSize = Len(osv1)
    Call GetVersionExA(osv1)
```

```
    Print "Get Build Number by GetVersionExA:"; osv1.dwBuildNumber
    'way 2
    osv2.dwOSVersionInfoSize = Len(osv2)
    Call RtlGetVersion(osv2)
    Print "Get Build Number by RtlGetVersion:"; osv2.dwBuildNumber
End Sub
```



本篇到此结束。作业：无。