2021/05/06_Windows32位内核_第11课_中断门、SSDT

笔记本: Windows32位内核
创建时间: 2021/5/6 星期四 15:05
作者: ileemi

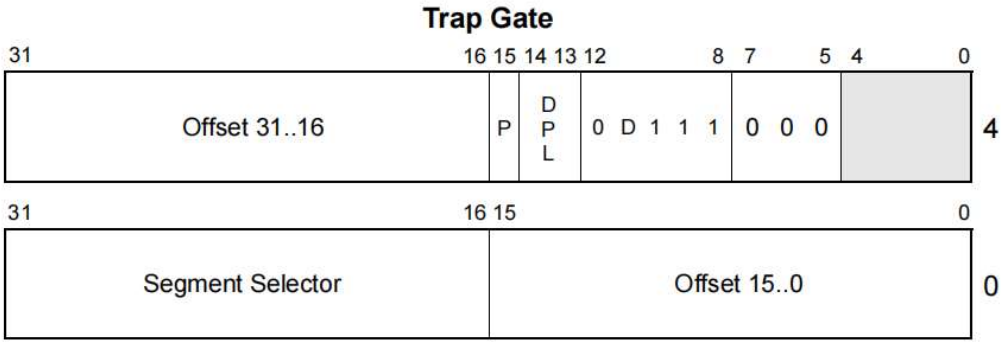
内核漏洞的利用

> 内核代码存在漏洞，就由机会操作GDT表（GDT表的地址在操作系统中是固定的），通过GDT表可以写入一个门描述符。门描述符的地址代码段指向Ring3程序的代码段。`dg 8` -- 表项权限只能在Ring0。`dg 1b` -- 表项权限可以在Ring3中使用。调用门的偏移地址填写Ring3中的函数名。Ring3程序调用调用门，CPU会切换权限，此时之前填写的Ring3中的函数就拥有Ring0权限。这样做，就可以在WinXP上无驱动进入Ring0环。

中断门编写API

> 在IDT表中找到一个空项（在中断描述表中找到保留项使用）。中断门和调用门的区别在于门的类型不同。

中断门（标志为14），表的格式如下：



> 系统段和门描述符类型：

Table 3-2. System-Segment and Gate-Descriptor Types

Type Field					Description	
Decimal	11	10	9	8	32-Bit Mode	IA-32e Mode
0	0	0	0	0	Reserved	Reserved
1	0	0	0	1	16-bit TSS (Available)	Reserved
2	0	0	1	0	LDT	LDT
3	0	0	1	1	16-bit TSS (Busy)	Reserved
4	0	1	0	0	16-bit Call Gate	Reserved
5	0	1	0	1	Task Gate	Reserved
6	0	1	1	0	16-bit Interrupt Gate	Reserved
7	0	1	1	1	16-bit Trap Gate	Reserved
8	1	0	0	0	Reserved	Reserved
9	1	0	0	1	32-bit TSS (Available)	64-bit TSS (Available)
10	1	0	1	0	Reserved	Reserved
11	1	0	1	1	32-bit TSS (Busy)	64-bit TSS (Busy)
12	1	1	0	0	32-bit Call Gate	64-bit Call Gate
13	1	1	0	1	Reserved	Reserved
14	1	1	1	0	32-bit Interrupt Gate	64-bit Interrupt Gate
15	1	1	1	1	32-bit Trap Gate	64-bit Trap Gate

中断返回不能使用 retf（retf 只适合调用门），需要使用 iretd 进行权限切换。中断门不需要拷贝参数。
Ring3切换Ring0的指令示例：int 21h ; 21h为IDT表中对应的中断门下标项。

分析win操作系统API的实现方式

微软操作系统的API没有做到GDT表中，而是通过中断门来实现API。通过IDT表的进行分析：

> !idt -a
> **2e**: 83e89fee nt!**KiSystemService**

```
kd> u KiSystemService
nt!KiSystemService:
83e89fee 6a00            push    0
83e89ff0 55              push    ebp
83e89ff1 53              push    ebx
83e89ff2 56              push    esi
83e89ff3 57              push    edi
83e89ff4 0fa0            push    fs
83e89ff6 bb30000000      mov     ebx,30h
83e89ffb 668ee3          mov     fs,bx
kd> u
nt!KiSystemService+0x10:
83e89ffe bb23000000      mov     ebx,23h
83e8a003 8edb            mov     ds,bx
83e8a005 8ec3            mov     es,bx
83e8a007 648b3524010000  mov     esi,dword ptr fs:[124h]
83e8a00e 64ff3500000000  push    dword ptr fs:[0]
83e8a015 64c70500000000ffffffff mov dword ptr fs:[0],0FFFFFFFFh
83e8a020 ffb63a010000    push    dword ptr [esi+13Ah]
83e8a026 83ec48          sub     esp,48h
```

通过操作系统加载的内核代码进行静态分析：ntkrnlpa.exe -- Win7不支持单核CPU，所以该文件就是一个多核的

```
kd> lm
start    end        module name
83e4c000 8425e000   nt         (pdb symbols)          d:\windbg_symbol\ntkrpamp.pdb\68
```

获取 KiSystemService 在模块（ntkrpamp.exe）中的偏移：

0x83e89fee - 0x83e4c000 = 0x3DFEE

通过IDA静态分析：

定位 KiSystemService 在模块（ntkrpamp.exe）中的位置：

0x00400000 + 0x3DFEE = 0x0043DFEE

```
text:0043DFEE
text:0043DFEE _KiSystemService proc near               ; CODE XREF: ZwAcceptConnectPort(x,x,x,x,x,x)+C↑p
text:0043DFEE                                          ; ZwAccessCheck(x,x,x,x,x,x,x,x)+C↑p ...
text:0043DFEE
text:0043DFEE arg_0           = dword ptr  4
text:0043DFEE
text:0043DFEE                 push    0
text:0043DFF0                 push    ebp
text:0043DFF1                 push    ebx
text:0043DFF2                 push    esi
text:0043DFF3                 push    edi
text:0043DFF4                 push    fs
text:0043DFF6                 mov     ebx, 30h ; '0'
text:0043DFFB                 mov     fs, bx
text:0043DFFE                 mov     ebx, 23h ; '#'
text:0043E003                 mov     ds, ebx
text:0043E005                 mov     es, ebx
text:0043E007                 mov     esi, large fs:124h
text:0043E00E                 push    large dword ptr fs:0
text:0043E015                 mov     large dword ptr fs:0, 0FFFFFFFFh
text:0043E020                 push    dword ptr [esi+13Ah]
text:0043E026                 sub     esp, 48h
text:0043E029                 mov     ebx, [esp+68h+arg_0]
text:0043E02D                 and     ebx, 1
text:0043E030                 mov     [esi+13Ah], bl

.text:0043E19C                mov     edi, [edi]
.text:0043E19E                mov     cl, [eax+edx]
.text:0043E1A1                mov     edx, [edi+eax*4]
.text:0043E1A4                sub     esp, ecx
.text:0043E1A6                shr     ecx, 2
.text:0043E1A9                mov     edi, esp
.text:0043E1AB                cmp     esi, ds:_MmUserProbeAddress
.text:0043E1B1                jnb     loc_43E3E5
.text:0043E1B7
.text:0043E1B7 loc_43E1B7:                             ; CODE XREF: _KiFastCallEntry+329↓j
.text:0043E1B7                                         ; DATA XREF: _KiTrap0E:loc_441448↓o
.text:0043E1B7                rep movsd
.text:0043E1B9                test    byte ptr [ebp+6Ch], 1
.text:0043E1BD                jz      short loc_43E1D5
.text:0043E1BF                mov     ecx, large fs:124h
.text:0043E1C6                mov     edi, [esp+7Ch+var_7C]
.text:0043E1C9                mov     [ecx+13Ch], ebx
.text:0043E1CF                mov     [ecx+12Ch], edi
.text:0043E1D5
.text:0043E1D5 loc_43E1D5:                             ; CODE XREF: _KiFastCallEntry+FD↑j
.text:0043E1D5                mov     ebx, edx
.text:0043E1D7                test    byte ptr ds:dword_537908, 40h
.text:0043E1DE                setnz   byte ptr [ebp+12h]
.text:0043E1E2                jnz     loc_43E574
.text:0043E1E8
.text:0043E1E8 loc_43E1E8:                             ; CODE XREF: _KiFastCallEntry+4BB↓j
.text:0043E1E8                call    ebx
.text:0043E1EA
.text:0043E1EA loc_43E1EA:                             ; CODE XREF: _KiFastCallEntry+334↓j
```

IDT表首地址

调用API

> 获取 _KPCR 结构体的首地址：
>
> ~~lea eax, fs:[0]~~ -- 无效指令
>
> mov eax, fs:[1CH]

```
.text:0043E0F0                push    ebp
.text:0043E0F1                push    ebx
.text:0043E0F2                push    esi
.text:0043E0F3                push    edi
.text:0043E0F4                mov     ebx, large fs:1Ch
.text:0043E0FB                push    3Bh ; ';'
.text:0043E0FD                mov     esi, [ebx+124h]
.text:0043E103                push    dword ptr [ebx]
.text:0043E105                mov     dword ptr [ebx], 0FFFFFFFFh
kd> dt _kpcr
nt!_KPCR
   +0x000 NtTib            : _NT_TIB
   +0x000 Used_ExceptionList : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Used_StackBase   : Ptr32 Void
   +0x008 Spare2           : Ptr32 Void
   +0x00c TssCopy          : Ptr32 Void
   +0x010 ContextSwitches  : Uint4B
   +0x014 SetMemberCopy    : Uint4B
   +0x018 Used_Self        : Ptr32 Void
   +0x01c SelfPcr          : Ptr32 _KPCR
   +0x020 Prcb             : Ptr32 _KPRCB
```

获取 _KPCR结构体的首地址

> 获取当前线程：mov esi, [ebx+124h]

```
   +0x120 PrcbData         : _KPRCB
kd> dt _KPRCB
nt!_KPRCB
   +0x000 MinorVersion     : Uint2B
   +0x002 MajorVersion     : Uint2B
   +0x004 CurrentThread    : Ptr32 _KTHREAD
   +0x008 NextThread       : Ptr32 _KTHREAD
   +0x00c IdleThread       : Ptr32 _KTHREAD
   +0x010 LegacyNumber     : UChar
```

+124

> 获取初始化栈的地址：mov ebp, [esi+28h]

```
.text:0043E0F4                mov     ebx, large fs:1Ch
.text:0043E0FB                push    3Bh ; ';'
.text:0043E0FD                mov     esi, [ebx+124h]
.text:0043E103                push    dword ptr [ebx]
.text:0043E105                mov     dword ptr [ebx], 0FFFFFFFFh
.text:0043E10B                mov     ebp, [esi+28h]
kd> dt _KTHREAD
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 CycleTime        : Uint8B
   +0x018 HighCycleTime    : Uint4B
   +0x020 QuantumTarget    : Uint8B
   +0x028 InitialStack     : Ptr32 Void
   +0x02c StackLimit       : Ptr32 Void
```

> 获取系统服务描述表（SSDT） _KTHREAD --> ServiceTable

```
.text:0043E14F loc_43E14F:                            ; CODE XREF: _KiBBTUnexpectedRange+18↑j
.text:0043E14F                                        ; _KiSystemService+7F↑j
.text:0043E14F                mov     edi, eax
.text:0043E151                shr     edi, 8
.text:0043E154                and     edi, 10h
.text:0043E157                mov     ecx, edi
.text:0043E159                add     edi, [esi+0BCh]
.text:0043E15F                mov     ebx, eax
.text:0043E161                and     eax, 0FFFh
.text:0043E166                cmp     eax, [edi+8]
.text:0043E169                jnb     _KiBBTUnexpectedRange
.text:0043E16F                cmp     ecx, 10h
.text:0043E172                jnz     short loc_43E18E
   +0x0b8 ReservedFlags    : Pos 11, 21 Bits
   +0x0b8 ThreadFlags      : Int4B
   +0x0bc ServiceTable     : Ptr32 Void
   +0x0c0 WaitBlock        : [4] _KWAIT_BLOCK
   +0x120 QueueListEntry   : _LIST_ENTRY
```

> 定位存储API表数组的首地址：
> 使用 dds 948a6000 L339 可显示存储API函数指针数组中的数据（显示对应的函数名）

```
.text:0043E19C                mov     edi, [edi]
.text:0043E19E                mov     cl, [eax+edx]
.text:0043E1A1                mov     edx, [edi+eax*4]
.text:0043E1A4                sub     esp, ecx
.text:0043E1A6                shr     ecx, 2
.text:0043E1A9                mov     edi, esp
.text:0043E1AB                cmp     esi, ds:_MmUserProbeAddress
.text:0043E1B1                jnb     loc_43E3E5
.text:0043E1B7
.text:0043E1B7 loc_43E1B7:                            ; CODE XREF: _KiFastCallEntry+329↓j
.text:0043E1B7                                        ; DATA XREF: _KiTrap0E:loc_441448↓o
.text:0043E1B7                rep movsd
.text:0043E1B9                test    byte ptr [ebp+6Ch], 1
.text:0043E1BD                jz      short loc_43E1D5
.text:0043E1BF                mov     ecx, large fs:124h
.text:0043E1C6                mov     edi, [esp+7Ch+var_7C]
.text:0043E1C9                mov     [ecx+13Ch], ebx
.text:0043E1CF                mov     [ecx+12Ch], edi
.text:0043E1D5
.text:0043E1D5 loc_43E1D5:                            ; CODE XREF: _KiFastCallEntry+FD↑j
.text:0043E1D5                mov     ebx, edx
.text:0043E1D7                test    byte ptr ds:dword_537908, 40h
.text:0043E1DE                setnz   byte ptr [ebp+12h]
.text:0043E1E2                jnz     loc_43E574
.text:0043E1E8
.text:0043E1E8 loc_43E1E8:                            ; CODE XREF: _KiFastCallEntry+4BB↓j
.text:0043E1E8                call    ebx
```

```
nt!KiFastCallEntry+0xce:
83e8a18e 64ff05b0060000  inc     dword ptr fs:[6B0h]
83e8a195 8bf2            mov     esi,edx
83e8a197 33c9            xor     ecx,ecx
83e8a199 8b570c          mov     edx,dword ptr [edi+0Ch]
83e8a19c 8b3f            mov     edi,dword ptr [edi]
83e8a19e 8a0c10          mov     cl,byte ptr [eax+edx]
83e8a1a1 8b1487          mov     edx,dword ptr [edi+eax*4]
83e8a1a4 2be1            sub     esp,ecx
83e8a1a6 c1e902          shr     ecx,2
83e8a1a9 8bfc            mov     edi,esp
83e8a1ab 3b351c67fb83    cmp     esi,dword ptr [nt!MmUserProbeAddress (83fb671c)]
83e8a1b1 0f832e020000    jae     nt!KiSystemCallExit2+0xa5 (83e8a3e5)
```

下断点

```
kd> bp 83e8a19c
breakpoint 3 redefined
kd> g
Breakpoint 2 hit
nt!KiFastCallEntry+0xd9:
83e8a199 8b570c          mov     edx,dword ptr [edi+0Ch]
kd> dd edi
83fb6a10  948a6000 00000000 00000339 948a702c
83fb6a20  00000000 00000000 83fb6a24 00000340
83fb6a30  00000340 865f0a38 00000007 00000000
83fb6a40  865f0970 865f0718 865f08a8 865f07e0
83fb6a50  00000000 865f0650 00000000 00000000
83fb6a60  83ec4809 83ed1eed 83ee03a5 00000003
83fb6a70  80783000 80784000 00000120 ffffffff
83fb6a80  00000001 00000000 00000000 00000000
```

存储API地址的函数指针数组

API表项的个数

存储API对应参数的结构

Win操作系统调用API的方式：
mov eax, 0 // 0 为对应的API函数指针在表中的下标
int 2EH // 中断门

> .reload /f kernel32.dll -- 加载指定模块的pdb文件，本地没有就从服务器进行下载。加载Ring3层的模块符号需要在当前层加载。

```
kd> lm
start    end        module name
77880000 77954000   kernel32   (pdb symbols)          d:\windbg_symbol\kernel32.pdb\D4AC3906D49D487B9F69F9326A7D2
77960000 77a9c000   ntdll      (pdb symbols)          d:\windbg_symbol\ntdll.pdb\120028FA453F4CD5A6A404EC37396A58
83e4c000 8425e000   nt         (pdb symbols)          d:\windbg_symbol\ntkrpamp.pdb\684DA42A30CC450F81C535B4D1894
kd> bp kernel32!CreateFileA
kd> g
Breakpoint 0 hit
kernel32!CreateFileA:
001b:778ccee8 8bff            mov     edi,edi
```

kernel32.dll 中的 CreateFileA 最终会调用 ntdll.dll 模块中的ZwCreateFile函数（新版本使用ZwCreateFile，老版本使用NtCreateFile，两个函数的导出地址在模块中一

致，为了兼容老版本）。由于参数类型以及参数个数不一致，所以在kernel32中的CreateFileA函数实现中会进行参数的转换=，如下图所示：

```
.text:77E2CEE8 hTemplateFile   = dword ptr  20h
.text:77E2CEE8
.text:77E2CEE8 ; FUNCTION CHUNK AT .text:77E4A818 SIZE 00000008 BYTES
.text:77E2CEE8
.text:77E2CEE8                 mov     edi, edi
.text:77E2CEEA                 push    ebp
.text:77E2CEEB                 mov     ebp, esp
.text:77E2CEED                 push    ecx
.text:77E2CEEE                 push    ecx
.text:77E2CEEF                 push    [ebp+lpFileName] ; SourceString
.text:77E2CEF2                 lea     eax, [ebp+UnicodeString]
.text:77E2CEF5                 push    eax             ; int
.text:77E2CEF6                 call    _Basep8BitStringToDynamicUnicodeString@8 ; Basep8BitStringToDynamicUnicodeString(x,x)
.text:77E2CEFB                 test    eax, eax
.text:77E2CEFD                 jz      loc_77E4A818
.text:77E2CF03                 push    esi
.text:77E2CF04                 push    [ebp+hTemplateFile] ; hTemplateFile
.text:77E2CF07                 push    [ebp+dwFlagsAndAttributes] ; dwFlagsAndAttributes
.text:77E2CF0A                 push    [ebp+dwCreationDisposition] ; dwCreationDisposition
.text:77E2CF0D                 push    [ebp+lpSecurityAttributes] ; lpSecurityAttributes
.text:77E2CF10                 push    [ebp+dwShareMode] ; dwShareMode
.text:77E2CF13                 push    [ebp+dwDesiredAccess] ; dwDesiredAccess
.text:77E2CF16                 push    [ebp+UnicodeString.Buffer] ; lpFileName
.text:77E2CF19                 call    _CreateFileWImplementation@28 ; CreateFileWImplementation(x,x,x,x,x,x,x)
.text:77E2CF1E                 mov     esi, eax
.text:77E2CF20                 lea     eax, [ebp+UnicodeString]
.text:77E2CF23                 push    eax             ; UnicodeString
.text:77E2CF24                 call    ds:__imp__RtlFreeUnicodeString@4 ; RtlFreeUnicodeString(x)
.text:77E2CF2A                 mov     eax, esi
.text:77F055C8 ; NTSTATUS __stdcall NtCreateFile(PHANDLE FileHandle, ACCESS_MASK DesiredAccess, POBJECT_ATTRIBUTES Ob
.text:77F055C8                 public _NtCreateFile@44
.text:77F055C8 _NtCreateFile@44 proc near              ; CODE XREF: RtlpFileIsWin32WithRCManifest(x)+86↑p
.text:77F055C8                                         ; LdrpNtCreateFileUnredirected(x,x,x)+21↑p ...
.text:77F055C8
.text:77F055C8 FileHandle      = dword ptr  4
.text:77F055C8 DesiredAccess   = dword ptr  8
.text:77F055C8 ObjectAttributes= dword ptr  0Ch
.text:77F055C8 IoStatusBlock   = dword ptr  10h
.text:77F055C8 AllocationSize  = dword ptr  14h
.text:77F055C8 FileAttributes  = dword ptr  18h
.text:77F055C8 ShareAccess     = dword ptr  1Ch
.text:77F055C8 CreateDisposition= dword ptr  20h
.text:77F055C8 CreateOptions   = dword ptr  24h
.text:77F055C8 EaBuffer        = dword ptr  28h
.text:77F055C8 EaLength        = dword ptr  2Ch
.text:77F055C8
.text:77F055C8                 mov     eax, 42h ; 'B'  ; NtCreateFile
.text:77F055CD                 mov     edx, 7FFE0300h
.text:77F055D2                 call    dword ptr [edx]
.text:77F055D4                 retn    2Ch ; ','
.text:77F055D4 _NtCreateFile@44 endp
.text:77F055D4
.text:77F055D4 ; ---------------------------------------------------------------------------
.text:77F055D7                 align 4
```

在SSDT表的第42项

函数指针用来解决不同调用约定的问题，会调用Kixxx函数

```
.text:77F070C0
.text:77F070C0 ; _DWORD __stdcall KiIntSystemCall()
.text:77F070C0                 public _KiIntSystemCall@0
.text:77F070C0 _KiIntSystemCall@0 proc near            ; DATA XREF: .text:off_77EF61B8↑o
.text:77F070C0
.text:77F070C0 arg_4           = byte ptr  8
.text:77F070C0
.text:77F070C0                 lea     edx, [esp+arg_4]
.text:77F070C4                 int     2Eh             ; DOS 2+ internal - EXECUTE COMMAND
.text:77F070C4                                         ; DS:SI -> counted CR-terminated command string
.text:77F070C6                 retn
.text:77F070C6 _KiIntSystemCall@0 endp
.text:77F070C6
```

> SSDT 表在内核模块（ntkrxxx）中被导出：KeServiceDescriptorTable

```
kd> dd KeServiceDescriptorTable
83fb69c0  83ecad9c 00000000 00000191 83ecb3e4
83fb69d0  00000000 00000000 00000000 00000000
83fb69e0  83f296af 00000000 01ac7834 000000bb
83fb69f0  00000011 00000100 5385d2ba d717548f
83fb6a00  83ecad9c 00000000 00000191 83ecb3e4
83fb6a10  948a6000 00000000 00000339 948a702c
83fb6a20  00000000 00000000 83fb6a24 00000340
83fb6a30  00000340 865f0a38 00000007 00000000
kd> dds 83ecad9c+42*4
83ecaea4  8409d1e4 nt!NtCreateFile
83ecaea8  840a8667 nt!NtCreateIoCompletion
83ecaeac  8403f977 nt!NtCreateJobObject
83ecaeb0  8412d6de nt!NtCreateJobSet
83ecaeb4  8404ee2a nt!NtCreateKey
83ecaeb8  8405dd1e nt!NtCreateKeyedEvent
```

> 对指定进程设置线程条件断点（条件为：nt!NtCreateFile）
> bp /p 87ced228 nt!NtCreateFile // 87ced228 -- EPROCESS

```
kd> bl
 0 e 8409d1e4     0001 (0001) nt!NtCreateFile
     Match process data 87ced228

kd> kb
ChildEBP RetAddr  Args to Child
9a1aec00 83e8a1ea 0016f8c4 80100080 0016f868 nt!NtCreateFile
9a1aec00 779a70b4 0016f8c4 80100080 0016f868 nt!KiFastCallEntry+0x12a
0016f824 779a55d4 75b3aa21 0016f8c4 80100080 ntdll!KiFastSystemCallRet
0016f828 75b3aa21 0016f8c4 80100080 0016f868 ntdll!ZwCreateFile+0xc
0016f8cc 778ccca0 00000060 80100080 00000001 KERNELBASE!CreateFileW+0x35e
0016f8f8 778ccf1e 00300cc8 80000000 00000001 kernel32!CreateFileWImplementation+0x69
0016f928 01136d7e 011daf30 80000000 00000001 kernel32!CreateFileA+0x37
0016fa24 01137583 00000001 002fffb0 00300db8 CreateFile+0x56d7e
0016fa44 011373d7 0fd4b80b 00000000 00000000 CreateFile+0x57583
0016faa0 0113726d 0016fab0 01137608 0016fabc CreateFile+0x573d7
0016faa8 01137608 0016fabc 778d3c45 7ffdd000 CreateFile+0x5726d
0016fab0 778d3c45 7ffdd000 0016fafc 779c37f5 CreateFile+0x57608
0016fabc 779c37f5 7ffdd000 7785d80b 00000000 kernel32!BaseThreadInitThunk+0xe
0016fafc 779c37c8 0113012c 7ffdd000 00000000 ntdll!__RtlUserThreadStart+0x70
0016fb14 00000000 0113012c 7ffdd000 00000000 ntdll!_RtlUserThreadStart+0x1b
```

老版本的CPU会通过 KiIntSystemCall 函数进行权限切换（从Ring3进入Ring0），新版本的CPU则通过 KiFastSystemCall（内部使用了 sysenter 指令） 函数进行权限切换。

```
.text:77F070B0
.text:77F070B0
.text:77F070B0 ; _DWORD __stdcall KiFastSystemCall()
.text:77F070B0                 public _KiFastSystemCall@0
.text:77F070B0 _KiFastSystemCall@0 proc near           ; DATA XREF: .text:off_77EF61B8↑o
.text:77F070B0                 mov     edx, esp
.text:77F070B2                 sysenter
.text:77F070B2 _KiFastSystemCall@0 endp
.text:77F070B2
.text:77F070B4 ; Exported entry 107. KiFastSystemCallRet
```

> softice：是DOS及Windows 2000及之前的内核级调试工具。

> kdcom.dll：负责双击调试

模式指定寄存器（MSR）

> MSR是CPU的一组64位寄存器，可以分别通过RDMSR和WRMSR 两条指令进行读和写的操作，前提要在ECX中写入MSR的地址

相关指令：

- sysenter：进入Ring0
- sysexit：回到Ring3
- wrmsr
- rdmsr

sysenter（Fast System Call）：快速调用特权0级系统过程。通过这条指令可以快速调用一个API。
中断门切换权限指令：int 2e

```
// int 2e -- 访问效率较低，都浪费在了内存访问上
// 调用一次API最少访问8次内存
push cs
push eip
push ss
push esp
```

```
...
iret
pop esp
pop ss
pop eip
pop cs
```

SYSENTER—Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 34	SYSENTER	ZO	Valid	Valid	Fast call to privilege level 0 system procedures.

sysexit：从快速系统呼叫中快速返回。

SYSEXIT—Fast Return from Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 35	SYSEXIT	ZO	Valid	Valid	Fast return to privilege level 3 user code.
REX.W + 0F 35	SYSEXIT	ZO	Valid	Valid	Fast return to 64-bit mode privilege level 3 user code.

保存Ring3寄存器，恢复Ring0寄存器。

rdmsr、wrmsr 用于操作下面的寄存器值：

- IA32_SYSENTER_CS（MSR address 174H）：此MSR的下16位是特权0级代码段的段选择器。该值还用于确定特权级别0堆栈段的段选择器（请参见操作部分）。此值不能表示一个为空的选择器。
- IA32_SYSENTER_EIP（MSR address 176H）：此MSR的值被加载到RIP中（因此，该值引用所选操作过程或例程的第一个指令）。在受保护模式下，仅加载第31：0位。
- IA32_SYSENTER_ESP（MSR address 175H）：此MSR的值将加载到RSP中（因此，此值包含权限级别为0的堆栈的堆栈指针）。此值不能表示一个非规范的地址。在受保护模式下，仅加载第31：0位。

> 上述的三个寄存器并不能满足权限切换时寄存器环境的保存，SS寄存器的值也需要进行切换。操作系统需要将Ring0的堆栈段设计到Ring0代码段相应的位置。Ring3的代码段和Ring0的堆栈段相邻，Ring3的堆栈段和Ring3的代码段相邻时就可以省去三个寄存器。
> sysenter：
> cs0 = 8, ss0 = cs0+8 = 10
> cs3 = cs0 + 16, ss3 = cs0 + 24

对于Ring3的 EIP、ESP，CPU使用ECX、EDX来取代。执行 sysexit 指令时，将ECX、EDX中的值还原到Ring3的 EIP、ESP中。

rdmsr：从特定型号的寄存器中读取

RDMSR—Read from Model Specific Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 32	RDMSR	ZO	Valid	Valid	Read MSR specified by ECX into EDX:EAX.

NOTES:
* See IA-32 Architecture Compatibility section below.

寄存器值的读取：

```
mov ecx, 174h
rdmsr
mov ecx, 175h
rdmsr
mov ecx, 176h
rdmsr
```

寄存器值的修改：

```
mov ecx, 174h
mov eax, 100h
wrmsr
```

在Windbg中可通过 rdmsr 指令读取模式指定寄存器的值：

```
nt!RtlpBreakWithStatusInstruction:
83ec7110 cc              int     3
kd> rdmsr 174H
msr[174] = 00000000`00000008
kd> rdmsr 175H
msr[175] = 00000000`80792000
kd> rdmsr 176H
msr[176] = 00000000`83e8a0c0
```

sysenter流程

1. 装载 SYSENTER_CS_MSG 到 CS 寄存器，设置目标代码段
2. 装载 SYSENTER_EIP_MSG 到 EIP 寄存器，设置目标指令
3. 装载 SYSENTER_CS_MSG + 8 到 SS 寄存器，设置堆栈段
4. 装载 SYSENTER_ESP_MSG 到 ESP 寄存器，设置栈帧
5. 切换 Ring0
6. 清除 EFLAGS 的 VM 标志
7. 执行 RING0 例程

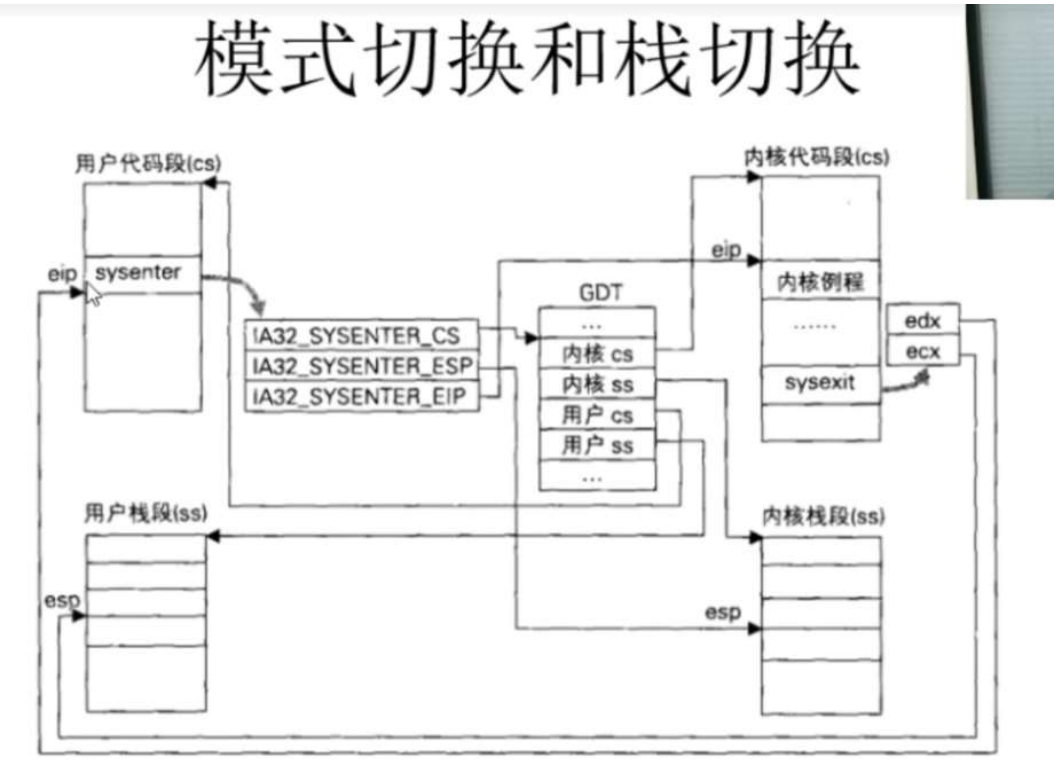
sysexit流程

1. 装载 SYSENTER_CS_MSG + 16 到 CS 寄存器
2. 将 EDX 的值送入 EIP
3. SYSENTER_CS_MSG + 24 装载到SS寄存器
4. 将 ECX 的值送入 ESP
5. 切换回 Ring3
6. 执行 EIP 处的Ring3指令

在R3进R0之前会将R3的EIP和ESP分别存入EDX和ECX，而后R0将ECX压栈，由于R3是call 进来的，栈上有返回地址，故在R0返回R3时，执行POP ECX; POP EDX即可。

在新版本中，基本上都使用此方式进入Ring0、回Ring3。在64位上，以 syscall 和 sysret 来完成进进入Ring0、回Ring3（在64位下对SSDT进行了加密，对于每个项拿

出 来右移4位+表基址，主要是为了防止SSDT Hook）。swapgs 交换gs寄存器（在64位下代替fs寄存器）。KPP保护：对内核模块代码进行保护，修改，会直接蓝屏。



MSR HOOK

修改 IA32_SYSENTER_EIP 的值，接管所有API的调用。Hook MSR、IDT表都有API漏掉的风险。

系统服务分派表的数据结构（SDT）

```
// extern PVOID KeServiceDescriptorTable;
// KeServiceDescriptorTable 只在32位操作系统的内核代码中进行导出

typedef struct ServiceDescriptorEntry {
  unsigned int*  KiServiceTable; // 服务表地址
  unsigned int*  Res; // 保留
  unsigned int   CountOfServices; // 服务数量
  unsigned char* KiArgumentTable; // API 参数数组指针
}  SSDTEntry;

extern SSDTEntry* KeServiceDescriptorTable;
```

```
kd> dds KeServiceDescriptorTable
83fb69c0  83ecad9c nt!KiServiceTable
83fb69c4  00000000
83fb69c8  00000191
83fb69cc  83ecb3e4 nt!KiArgumentTable
```


```
kd> dds KeServiceDescriptorTable
83fb69c0  83ecad9c nt!KiServiceTable
83fb69c4  00000000
83fb69c8  00000191
83fb69cc  83ecb3e4 nt!KiArgumentTable
```