

## 2021/05/07\_Windows32位内核\_第12课\_ShadowSSDT、HOOK SSDT\_API

笔记本: Windows32位内核  
创建时间: 2021/5/7 星期五 14:59  
作者: ileemi

---

- [UI相关API](#)
- [分析 KeServiceDescriptorTableShadow](#)
- [HOOK SSDT API](#)

SSDT 支持显示函数名:

- 函数名写死 (为了兼容性, 需要判断操作系统的版本)
- 通过ntdll.dll的导出函数, 通过SSDT获取到的下标, 遍历导出函数, 函数实现中会用到这个下标。
- 兼容所有版本的操作系统, 按照对应的操作系统版本去和.pdb文件进行匹配 (类似火绒剑)

ARK (Anti RootKit) 工具: 火绒剑、pchunter

Hook操作系统API的常用方法:

- SSDT Hook
- Inline Hook
- MSR Hook
- Hook KiFastCallEntry: 判断EAX, 就知道调用了哪些API

可实现的工具:

- 病毒行为监控: Hook操作系统所有API, 通过API的调用来判断其主要行为。
- 主动防御软件: 在内核中Hook所有含有恶意行为的API。对危险的操作进行过滤, 将具有危险的操作通知给对应的Ring3程序, Ring3程序将处理结果返回给驱动, 驱动根据Ring3返回的结果进行处理 (拒绝恶意操作, 直接API进行返回)。

主动防御软件现在主流的做法就是通过事件对象进行Ring0与Ring3之间的消息传递。

Ring3程序通过CreateEvent创建事件对象, 产生对应的句柄。在内核中需要将Ring3产生的句柄转换为对象 (ObReferenceObjectByHandle) 后, 即可得到对应的同步对象。Ring0中通过KeSetEvent设置事件对象。由于对象由Ring3产生 (存在伪造的风险)。所以就改为Ring0创建对象 (IoCreateNotificationEvent), Ring3来打开事件对象 (OpenEvent)。

通过 ObReferenceObjectByHandle 可以将句柄转为对象, 同时会增加引用计数。不使用时, 调用 ObDereferenceObject 来减去增加的引用计数。

驱动编写中的链表: LIST\_ENTRY。InitializeListHead、InsertTailList、RemoveHeadList、RemoveTailList等。

# UI相关API

UI相关的API在单独的驱动中（win32k.sys）。相关API保存在 ShadowSDDT（是一个隐藏表，在内核代码中没有导出）表（KeServiceDescriptorTableShadow）中。

```
kd> dd KeServiceDescriptorTableShadow
83fb6a00 83ecad9c 00000000 00000191 83ecb3e4
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
```

包含了SSDT

UI线程跑到 KeServiceDescriptorTableShadow 中的表项才有效:

```
kd> u
83e8a159 03bebc000000 add edi,dword ptr [esi+0BCh]
83e8a15f 8bd8 mov ebx,eax
83e8a161 25ff0f0000 and eax,0FFFh
83e8a166 3b4708 cmp eax,dword ptr [edi+8]
83e8a169 0f8333fdffff jae nt!KiBbTUnexpectedRange (83e89ea2)
83e8a16f 83f910 cmp ecx,10h
83e8a172 751a jne nt!KiFastCallEntry+0xcce (83e8a18e)
83e8a174 8b8e88000000 mov ecx,dword ptr [esi+88h]
kd> bp 83e8a159
kd> g
Break instruction exception - code 80000003 (first chance)
nt!KiFastCallEntry+0x99:
83e8a159 03bebc000000 add edi,dword ptr [esi+0BCh]
kd> t
nt!KiFastCallEntry+0x9f:
83e8a15f 8bd8 mov ebx,eax
kd> dd 0x83fb6a10
ReadVirtual: 83fb6a10 not properly sign extended
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
kd> dd 948a6000
ReadVirtual: 948a6000 not properly sign extended
948a6000 94833d37 9484bc23 946a71ac 94842c5d
948a6010 9484d369 94834554 948345e8 9475dad1
948a6020 9484cb94 94711965 94711882 9484eead
948a6030 9484d085 9484bc97 947528cb 9484cfd8
948a6040 9484fc51 9484bb9e 94784a88 9484d10f
948a6050 9484f645 94712069 947b88bf 947dc7bc
948a6060 9485063d 94845659 9477358b 9484d075
948a6070 947c76c3 9484f508 9484f8d2 9474cf2e
```

此时表项才有效

```
946a0000 948ec000 win32k (pdb symbols) d:\windbg_symbol\win32k.pdb\A03753B3B9274F8E848233:
kd> dds 948a6000 L339
948a6000 94833d37 win32k!NtGdiAbortDoc
948a6004 9484bc23 win32k!NtGdiAbortPath
948a6008 946a71ac win32k!NtGdiAddFontResourceW
948a600c 94842c5d win32k!NtGdiAddRemoteFontToDC
948a6010 9484d369 win32k!NtGdiAddFontMemResourceEx
948a6014 94834554 win32k!NtGdiRemoveMergeFont
948a6018 948345e8 win32k!NtGdiAddRemoteMMInstanceToDC
948a601c 9475dad1 win32k!NtGdiAlphaBlend
948a6020 9484cb94 win32k!NtGdiAngleArc
948a6024 94711965 win32k!NtGdiAnyLinkedFonts
948a6028 94711882 win32k!NtGdiFontIsLinked
948a602c 9484eead win32k!NtGdiArcInternal
948a6030 9484d085 win32k!NtGdiBeginGdiRendering
948a6034 9484bc97 win32k!NtGdiBeginPath
948a6038 947528cb win32k!NtGdiBitBlt
948a603c 9484cfd8 win32k!NtGdiCancelDC
948a6040 9484fc51 win32k!NtGdiCheckBitmapBits
948a6044 9484bb9e win32k!NtGdiCloseFigure
948a6048 94784a88 win32k!NtGdiClearBitmapAttributes
948a604c 9484d10f win32k!NtGdiClearBrushAttributes
948a6050 9484f645 win32k!NtGdiColorCorrectPalette
948a6054 94712069 win32k!NtGdiCombineRgn
```

## 分析 KeServiceDescriptorTableShadow

在 "ntkrnlpa.exe" 中，通过 KiFastCallEntry 进行分析：

```
.text:0043E14F      mov     edi, eax
.text:0043E14F      shr     edi, 8
.text:0043E151      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
.text:0043E174      mov     ecx, [esi+88h]
.text:0043E17A      xor     esi, esi

kd> dd KeServiceDescriptorTableShadow
83fb6a00 83ecad9c 00000000 00000191 83ecb3e4
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
```

通过表示来判断要查询的表项，有UI操作就查询KeServiceDescriptorTableShadow表首地址+10H

API 序号低12位有效，可有4096项

SSDT +10H

上图中，add edi, [esi+0BCh] // 表示 SSDT 在 KTHREAD 0BCh偏移处。

+0x0bc ServiceTable : Ptr32 Void

0x1000，高位是标志位（1，ShadowSSDT；0，SSDT），低12位为下标。

分析Ring3 API最终如何进入Ring0：以 FindWindowExW 为例

```
内核源码\Win7\user32.dll
mina Options Windows Help
No debugger
Instruction Data Unexplored External symbol Lumina function
IDA View-A Hex View-1 Structures Enums Imports Exports
.text:77D1DA9B ; __stdcall NtUserFindWindowEx(x, x, x, x, x)
.text:77D1DA9B ; _NtUserFindWindowEx@20 proc near
.text:77D1DA9B ; CODE XREF: InternalFindWindowExA(x,x,x,x,x,x)+581p
.text:77D1DA9B ; InternalFindWindowExW(x,x,x,x,x,x)+574p
.text:77D1DA9B      mov     eax, 118Ch
.text:77D1DAA0      mov     edx, 7FFE0300h
.text:77D1DAA5      call    dword ptr [edx]
.text:77D1DAA7      retn     14h
.text:77D1DAA7 ; _NtUserFindWindowEx@20 endp
.text:77D1DAA7 ;
.text:77D1DAA7 ;
.text:77D1DAAA      db 5 dup(90h)

kd> dds 948a6000+18c*4
948a6630 9471c0db win32k!NtUserFindWindowEx
948a6634 947cb642 win32k!NtUserFlashWindowEx
948a6638 94800073 win32k!NtUserFrostCrashedWindow
948a663c 947fbe2e win32k!NtUserGetAltTabInfo
```

进入Ring0

0x118C  
0x1000: ShadowSSDT  
0x018C: API下标

对于 "ntdll.dll" 中的API来说，其根据标志就会查询 "SSDT" 表：

```
ktop\操作系统内核源码\ntdll.dll.idb
Lumina Options Windows Help
No debugger
Instruction Data Unexplored External symbol Lumina function
IDA View-A Hex View-1 Structures Enums Imports Exports
.text:77F05D88 ; __stdcall ZwOpenProcess(x, x, x, x)
.text:77F05D88 ; public _ZwOpenProcess@16
.text:77F05D88 ; _ZwOpenProcess@16 proc near
.text:77F05D88 ; CODE XREF: RtlQueryProcessDebugInformation(x,x,x)+8F1p
.text:77F05D88 ; RtlpChangeQueryDebugBufferTarget(x,x,x,x)+64C3A1p ...
.text:77F05D88 ; NtOpenProcess
.text:77F05D88      mov     eax, 0BEh
.text:77F05D8D      mov     edx, 7FFE0300h
.text:77F05D92      call    dword ptr [edx]
.text:77F05D94      retn     10h
.text:77F05D94 ; _ZwOpenProcess@16 endp
.text:77F05D94 ;
```

会查询 SSDT 表



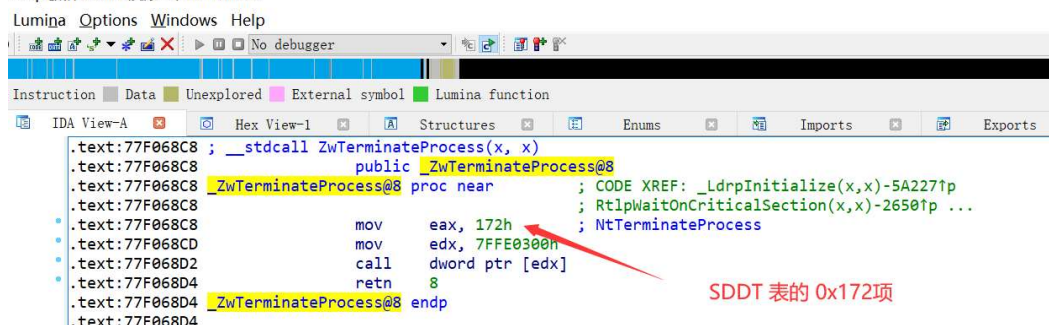
# HOOK SDDT API

主动防御软件实现：防止软件自身被别的进程关闭。

根据需求进行分析，分析操作会调用的API。进程结束任务时会调用 "TerminateProcess" 函数。HookAPI一般从所需API的源头开始Hook，由于 TerminateProcess 函数的第一个参数为目标进程的句柄，所以应该Hook OpenProcess，或者OpenThread。

Hook TerminateProcess，就需要分析该API在 SDDT 表还是在ShadowSDDT表中。

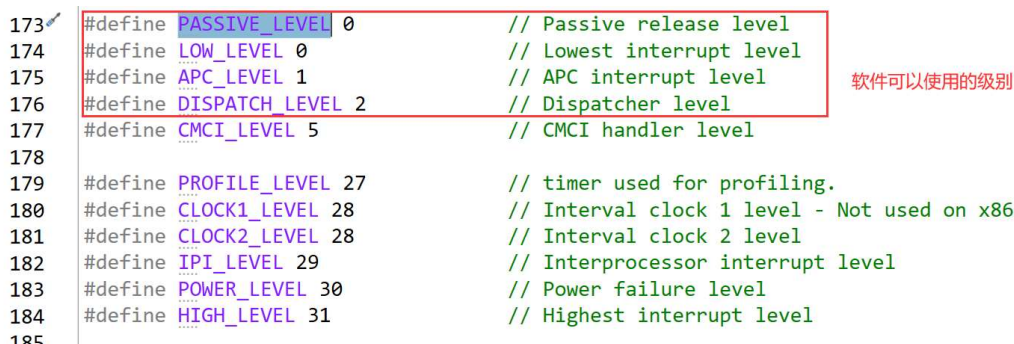
iktop\操作系统内核源码\ntdll.dll.idb



```
.text:77F068C8 ; __stdcall ZwTerminateProcess(x, x)
.text:77F068C8 public _ZwTerminateProcess@8
.text:77F068C8 _ZwTerminateProcess@8 proc near
.text:77F068C8 ; CODE XREF: _LdrpInitialize(x,x)-5A227?p
.text:77F068C8 ; RtlpWaitOnCriticalSection(x,x)-2650?p ...
.text:77F068C8 ; NtTerminateProcess
.text:77F068CD mov     eax, 172h
.text:77F068CD mov     edx, 7FFE0300h
.text:77F068D2 call    dword ptr [edx]
.text:77F068D4 retn     8
.text:77F068D4 _ZwTerminateProcess@8 endp
```

```
kd> dds 83ecad9c+172*4
83ecb364 840a89bf nt!NtTerminateProcess
83ecb368 840c6334 nt!NtTerminateThread
83ecb36c 840bdafa nt!NtTestAlert
83ecb370 83f2075f nt!NtThawRegistry
83ecb374 84141478 nt!NtThawTransactions
83ecb378 8409d9bb nt!NtTraceControl
```

LRQL：中断请求级别（每个线程一个，每个线程都会运行在对应的中断请求级别上），widnows线程调度机制是**抢占式调度机制**。Windows操作系统同时还设计了**动态优先级**机制。可创建进程时所赋予其优先权。



```
173 #define PASSIVE_LEVEL 0 // Passive release level
174 #define LOW_LEVEL 0 // Lowest interrupt level
175 #define APC_LEVEL 1 // APC interrupt level
176 #define DISPATCH_LEVEL 2 // Dispatcher level
177 #define CMCI_LEVEL 5 // CMCI handler level
178
179 #define PROFILE_LEVEL 27 // timer used for profiling.
180 #define CLOCK1_LEVEL 28 // Interval clock 1 level - Not used on x86
181 #define CLOCK2_LEVEL 28 // Interval clock 2 level
182 #define IPI_LEVEL 29 // Interprocessor interrupt level
183 #define POWER_LEVEL 30 // Power failure level
184 #define HIGH_LEVEL 31 // Highest interrupt level
185
```

API操作有对应的中断请求级别：

## Comments

**RtlCopyMemory** runs faster than **RtlMoveMemory**. However, the (Source) passed in to **RtlCopyMemory**.

Callers of **RtlCopyMemory** can be running at any IRQL if both memory buffers are at **IRQL < DISPATCH\_LEVEL**.

## Requirements

**Versions:** Available in Windows 2000 and later versions of Windows.

**IRQL:** PASSIVE\_LEVEL (see Comments section)

**Headers:** Declared in *Wdm.h*. Include *Wdm.h*, *Ntddk.h*, or *Ntifs.h*.

**Library:** Contained in *Ntoskrnl.lib*.

在编写驱动时，需要使用 "KeGetCurrentIrql" 来获取当前的 IRQL 级别。同时可以通过 "KeRaiseIrql、KeLowerIrql" 操作IRQL的级别（0~2）。可用来防止出现同步操作。

驱动中的每个函数都应该检查中断请求级别（一般为0级），级别不能大于 APC\_LEVEL。

```
KIRQL irql = KeGetCurrentIrql();
if (irql >= APC_LEVEL) {
    return 0;
}
// 上述的代码可使用 PAGED_CODE(); 进行代替
#define PAGED_CODE() PAGED_ASSERT(KeGetCurrentIrql() <= APC_LEVEL);
```

SSDT 表默认的内存属性是不可写的。修改表项前需要修改内存属性。

```
__asm {
    // 屏蔽当前核心中断，反之出现线程切换
    cli

    //关闭写保护标志
    mov eax, cr0
    and eax, not 10000h
    mov cr0, eax
}

// ...

__asm {
    //恢复写保护
    mov eax, cr0
    or eax, 10000h
    mov cr0, eax

    //恢复中断
    sti
}
```

ObReferenceObjectByHandle: 通过句柄获取对象，引用计数加1。

ObDereferenceObject: 减少对象引用计数。

PsGetProcessImageFileName: 获取进程名 (在\_EPROCESS 0x16C项, 类型为UCHAR)。

Zw... 开头的函数基本对应的都有 Nt... 开头的函数, 不推荐直接使用 Nt... 开头的函数是因为 Zw... 开头的函数会负责参数的验证。调用 Nt... 开头的函数 需要自己检测参数。

以 ZwTerminateProcess 为例:

```
kd> u ZwTerminateProcess
nt!ZwTerminateProcess:
83e8943c b872010000 mov     eax,172h
83e89441 8d542404   lea     edx,[esp+4]
83e89445 9c         pushfd
83e89446 6a08      push    8
83e89448 e8a10b0000 call    nt!KiSystemService (83e89fee)
83e8944d c20800    ret     8
nt!ZwTerminateThread:
83e89450 b873010000 mov     eax,173h
83e89455 8d542404   lea     edx,[esp+4]
```

会调用  
KeServiceDescriptorTableShadow  
的0x172项: NtTerminateProcess

进入Ring0

```
kd> dds 83ecad9c+172*4
83ecb364 840a89bf nt!NtTerminateProcess
83ecb368 840c6334 nt!NtTerminateThread
83ecb36c 840bdafa nt!NtTestAlert
```

在Hook内核API时需要注意:

- 重入问题: Hook的内核API, 自己的进程是否在使用还函数, 所以在HookAPI前需要判断是否是自己的进程在操作。防止出现蓝屏。  
return g\_pfnOld(ProcessHandle, ExitStatus);

- 卸载问题: 卸载Hook时, 需要保证Hook的API没有线程在占用。

解决方法:

1. 在驱动卸载回调中卸载Hook回调, 使用 KeDelayExecutionThread 函数等待线程结束 (类似), 等待时间就要考虑, 写该方法不靠谱。
2. 使用引用计数, 在进入Hook回调时引用计数加一, 出Hook回调时引用计数减一。在卸载时判断引用计数的值是否等于0 (需要注意同步问题)。