

在 WIN64 上 HOOK SSDT 和 UNHOOK SSDT 在原理上跟 WIN32 没什么不同，甚至说 HOOK 和 UNHOOK 在本质上也没有不同，都是在指定的地址上填写一串数字而已（填写代理函数的地址时叫做 HOOK，填写原始函数的地址时叫做 UNHOOK）。不过实现起来还是很大不同的。废话不多说，开始分点讲解 HOOK 和 UNHOOK。

一、HOOK SSDT

要挂钩 SSDT，必然要先得到 ServiceTableBase 的地址。和 SSDT 相关的两个结构体 SYSTEM_SERVICE_TABLE 以及 SERVICE_DESCRIPTOR_TABLE 并没有发生什么的变化（除了整个结构体的长度胖了一倍）：

```
typedef struct _SYSTEM_SERVICE_TABLE{
    PVOID    ServiceTableBase;
    PVOID    ServiceCounterTableBase;
    SIZE_T   NumberOfServices;
    PVOID    ParamTableBase;
} SYSTEM_SERVICE_TABLE, *PSYSTEM_SERVICE_TABLE;

typedef struct _SERVICE_DESCRIPTOR_TABLE{
    SYSTEM_SERVICE_TABLE ntoskrnl; // ntoskrnl.exe (native api)
    SYSTEM_SERVICE_TABLE win32k;   // win32k.sys (gdi/user)
    SYSTEM_SERVICE_TABLE Table3;   // not used
    SYSTEM_SERVICE_TABLE Table4;   // not used
}SERVICE_DESCRIPTOR_TABLE,*PSERVICE_DESCRIPTOR_TABLE;
```

得到 ServiceTableBase 的地址后，就能得到每个服务函数的地址了。但和 WIN32 不一样，这个表存放的并不是 SSDT 函数的完整地址，而是其相对于 ServiceTableBase[Index]>>4 的数据（我称它为偏移地址），每个数据占四个字节，所以计算指定 Index 函数完整地址的公式是：ServiceTableBase[Index]>>4 + ServiceTableBase。代码如下：

```
ULONGLONG GetSSDTFuncCurAddr(ULONG id)
{
    LONG dwtmp=0;
    PULONG ServiceTableBase=NULL;
    ServiceTableBase=(PULONG)KeServiceDescriptorTable->ServiceTableBase;
    dwtmp=ServiceTableBase[id];
    dwtmp=dwtmp>>4;
    return dwtmp + (ULONGLONG)ServiceTableBase;
}
```

反之，从函数的完整地址获得函数偏移地址的代码也就出来了：

```
ULONG GetOffsetAddress(ULONGLONG FuncAddr)
{
    LONG dwtmp=0;
    PULONG ServiceTableBase=NULL;
    ServiceTableBase=(PULONG)KeServiceDescriptorTable->ServiceTableBase;
    dwtmp=(LONG)(FuncAddr-(ULONGLONG)ServiceTableBase);
```

```

    return dwtmp<<4;
}

```

知道了这一套机制，HOOK SSDT 就很简单了，首先获得待 HOOK 函数的序号 Index，然后通过公式把自己的代理函数的地址转化为偏移地址，然后把偏移地址的数据填入 ServiceTableBase[Index]。也许有些读者看到这里，已经觉得胜利在望了，我当时也是如此。但实际上我在这里栽了个大跟头，整整郁闷了很长时间！因为我低估了设计这套算法的工程师的智商，我没有考虑一个问题，为什么 WIN64 的 SSDT 表存放地址的形式这么奇怪？只存放偏移地址，而不存放完整地址？难道是为了节省内存？这肯定是不可能的，要知道现在内存白菜价。那么不是为了节省内存，唯一的可能性就是要给试图挂钩 SSDT 的人制造麻烦！要知道，WIN64 内核里每个驱动都不在同一个 4GB 里，而 4 字节的整数只能表示 4GB 的范围！所以无论你怎么修改这个值，都跳不出 ntoskrnl 的手掌心。如果你想通过修改这个值来跳转到你的代理函数，那是绝对不可能的。**因为你的驱动的地址不可能跟 ntoskrnl 在同一个 4GB 里。**然而，这位工程师也低估了我们中国人的智商，在中国有两句成语，这位工程师一定没听过，叫“明修栈道，暗渡陈仓”以及“上有政策，下有对策”。虽然不能直接用 4 字节来表示自己的代理函数所在的地址，但是还是可以修改这个值的。要知道，ntoskrnl 虽然有很多地方的代码通常是不会被执行的，比如 KeBugCheckEx。所以我的办法是：**修改这个偏移地址的值，使之跳转到 KeBugCheckEx，然后在 KeBugCheckEx 的头部写一个 12 字节的 mov - jmp，这是一个可以跨越 4GB 的跳转，跳到我们的函数里！**代码如下：

```

VOID FuckKeBugCheckEx()
{
    KIRQL irql;
    ULONGLONG myfun;
    UCHAR jmp_code[]="\x48\xB8\xff\xff\xff\xff\xff\xff\xff\x00\xff\xe0";
    myfun=(ULONGLONG)Fake_NtTerminateProcess;
    memcpy(jmp_code+2,&myfun,8);
    irql=WPOFFx64();
    memset(KeBugCheckEx,0x90,15);
    memcpy(KeBugCheckEx,jmp_code,12);
    WPONx64(irql);
}

VOID HookSSDT()
{
    KIRQL irql;
    ULONGLONG dwtmp=0;
    PULONG ServiceTableBase=NULL;
    //get old address
    NtTerminateProcess=(NTTERMINATEPROCESS)GetSSDTFuncCurAddr(41);
    dprintf("Old_NtTerminateProcess: %llx", (ULONGLONG)NtTerminateProcess);
    //set kebugcheckex
    FuckKeBugCheckEx();
}

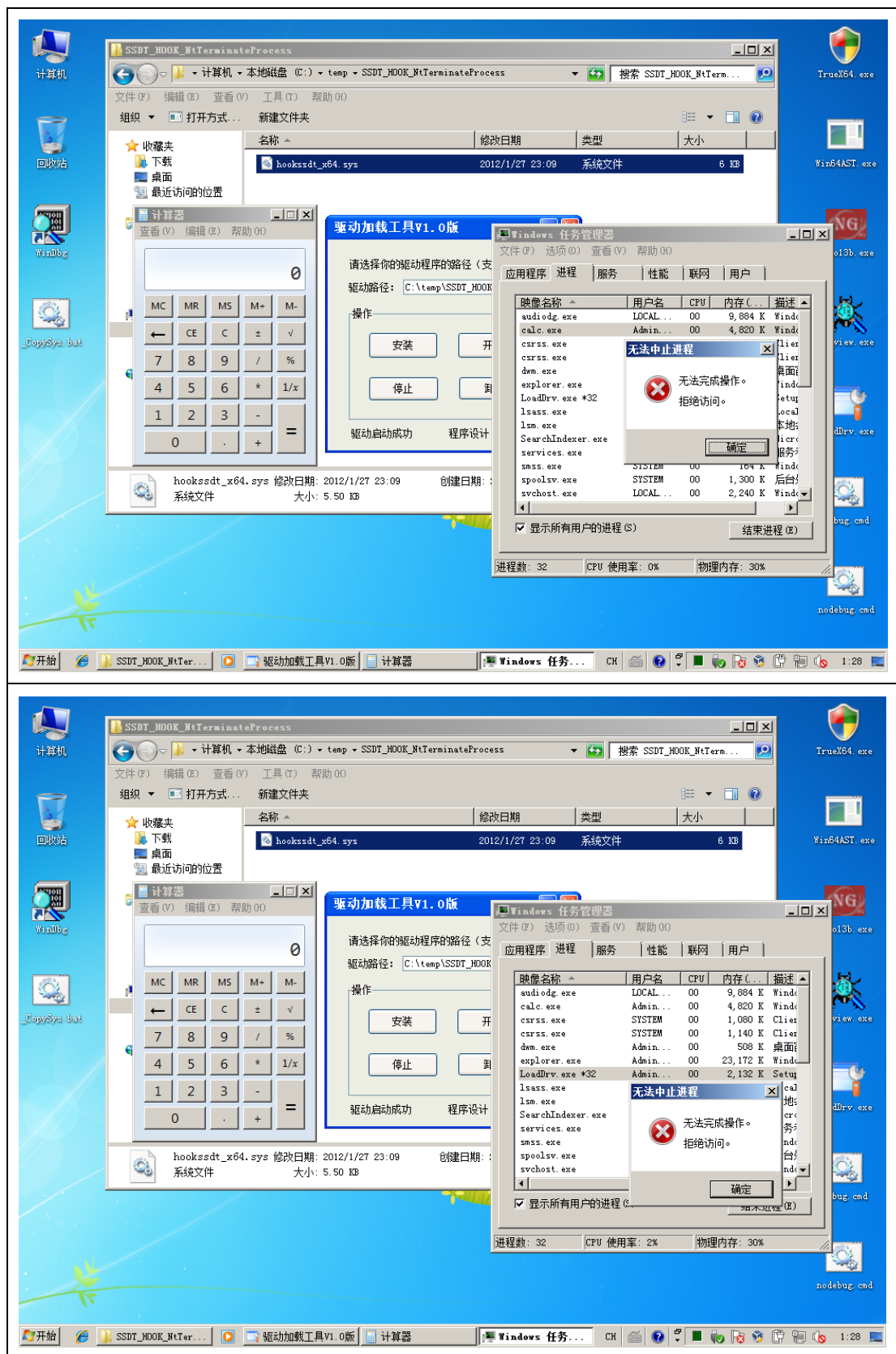
```

```
//show new address
ServiceTableBase=(PULONG)KeServiceDescriptorTable->ServiceTableBase;
OldTpVal=ServiceTableBase[41]; //record old offset value
irq1=WPOFFx64();
ServiceTableBase[41]=GetOffsetAddress((ULONGLONG)KeBugCheckEx);
WPONx64(irq1);
dprintf("KeBugCheckEx: %llx", (ULONGLONG)KeBugCheckEx);
dprintf("New_NtTerminateProcess: %llx", GetSSDTFuncCurAddr(41));
}
```

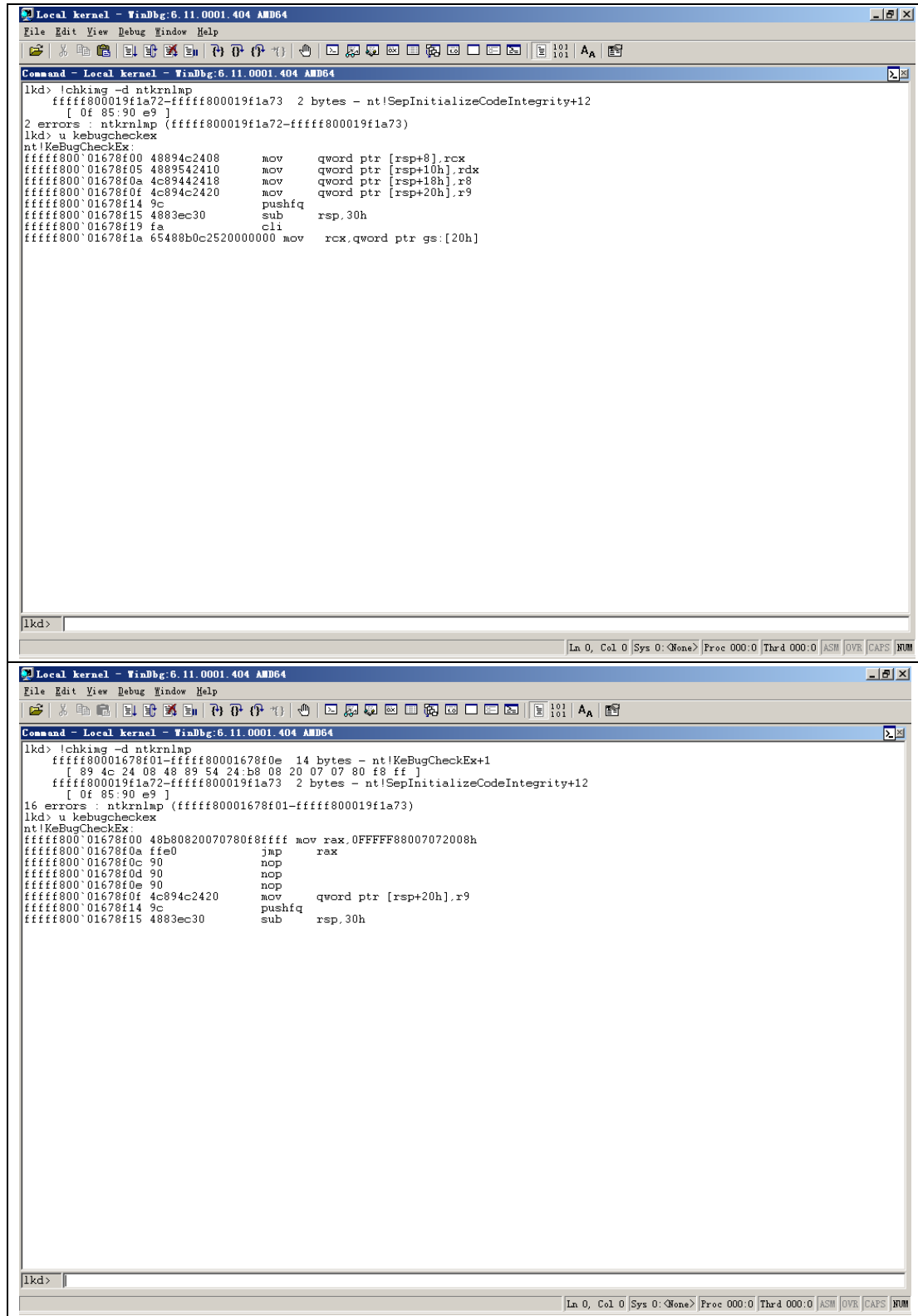
在代理函数里这么写,保护名为 calc.exe 和 loaddrv.exe 的程序不被结束:

```
NTSTATUS __fastcall Fake_NtTerminateProcess(IN HANDLE ProcessHandle, IN NTSTATUS
ExitStatus)
{
    PEPROCESS Process;
    NTSTATUS st = ObReferenceObjectByHandle (ProcessHandle, 0, *PsProcessType, KernelMode,
&Process, NULL);
    DbgPrint("Fake_NtTerminateProcess called!");
    if (NT_SUCCESS(st))
    {
        if (!_stricmp(PsGetProcessImageFileName(Process), "loaddrv.exe") || !_stricmp(PsGetProcessImageFileName(Process), "calc.exe"))
            return STATUS_ACCESS_DENIED;
        else
            return NtTerminateProcess(ProcessHandle, ExitStatus);
    }
    else
        return STATUS_ACCESS_DENIED;
}
```

注意在代理函数一定要注明是__fastcall, 否则会出问题。测试效果如下:



给大家看一下 WINDBG 里的反汇编结果（挂钩前和挂钩后）：



Local kernel - WinDbg:6.11.0001.404 AMD64

File Edit View Debug Window Help

Command - Local kernel - WinDbg:6.11.0001.404 AMD64

```
lkd> !chkimg -d nt!ntkrnlmp
fffff800019f1a72-fffff800019f1a73  2 bytes - nt!SepInitializeCodeIntegrity+12
[ 0f 85 90 e9 ]
2 errors : nt!ntkrnlmp (fffff800019f1a72-fffff800019f1a73)
lkd> u kebugcheckex
nt!KeBugCheckEx:
fffff800`01678f00 48894c2408      mov     qword ptr [rsp+8],rcx
fffff800`01678f05 4889542410      mov     qword ptr [rsp+10h],rdx
fffff800`01678f0a 4c89442418      mov     qword ptr [rsp+18h],r8
fffff800`01678f0f 4c894c2420      mov     qword ptr [rsp+20h],r9
fffff800`01678f14 9c             pushfq
fffff800`01678f15 4883ec30       sub     rsp,30h
fffff800`01678f19 fa             cli
fffff800`01678f1a 65488b0c2520000000 mov     rcx,qword ptr gs:[20h]
```

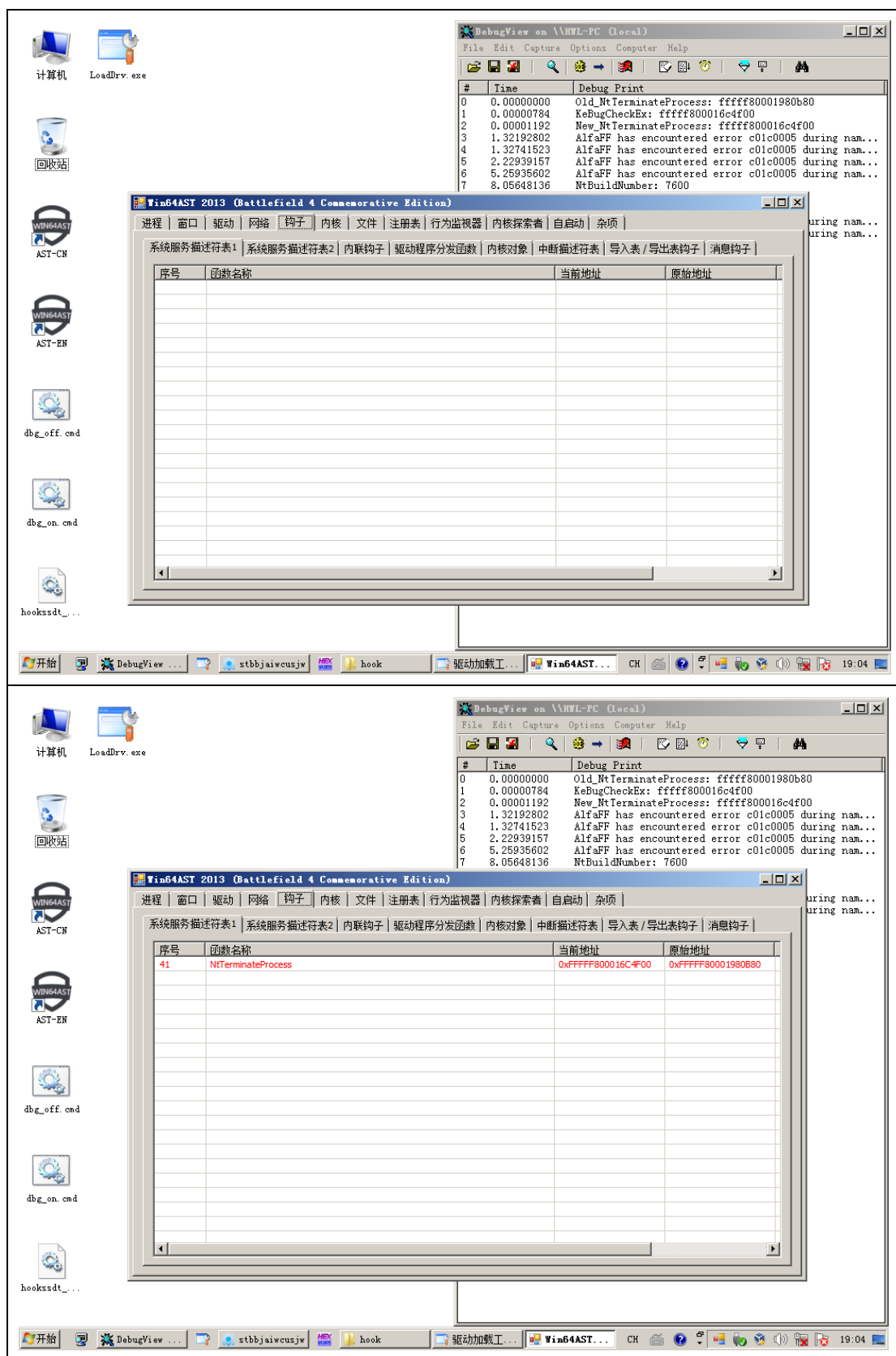
Local kernel - WinDbg:6.11.0001.404 AMD64

File Edit View Debug Window Help

Command - Local kernel - WinDbg:6.11.0001.404 AMD64

```
lkd> !chkimg -d nt!ntkrnlmp
fffff80001678f01-fffff80001678f0e  14 bytes - nt!KeBugCheckEx+1
[ 89 4c 24 08 48 89 54 24 b8 08 20 07 80 f8 ff ]
fffff800019f1a72-fffff800019f1a73  2 bytes - nt!SepInitializeCodeIntegrity+12
[ 0f 85 90 e9 ]
16 errors : nt!ntkrnlmp (fffff80001678f01-fffff800019f1a73)
lkd> u kebugcheckex
nt!KeBugCheckEx:
fffff800`01678f00 48b80820070780f8ffff mov     rax,0FFFFFFF88007072008h
fffff800`01678f0a ffe0          jmp     rax
fffff800`01678f0c 90             nop
fffff800`01678f0d 90             nop
fffff800`01678f0e 90             nop
fffff800`01678f0f 4c894c2420      mov     qword ptr [rsp+20h],r9
fffff800`01678f14 9c             pushfq
fffff800`01678f15 4883ec30       sub     rsp,30h
```

用 WIN64AST 查看的效果如下（挂钩前和挂钩后）：



接下来给出取消 SSDT HOOK 的代码,在这个代码里我没有复原 KeBugCheckEx 的原始内容,因为执行到 KeBugCheckEx 就意味着蓝屏,所以是否恢复 KeBugCheckEx 的原始机器码都无所谓了:

```

VOID UnhookSSDT()
{
    KIRQL irql;
    PULONG ServiceTableBase=NULL;
    ServiceTableBase=(PULONG)KeServiceDescriptorTable->ServiceTableBase;
    //set value
    irql=WPOFFx64();
    ServiceTableBase[41]=GetOffsetAddress((ULONGLONG)NtTerminateProcess);
    WPONx64(irql);
    //没必要恢复 KeBugCheckEx 的内容了，反正执行到 KeBugCheckEx 时已经完蛋了。
    dprintf("NtTerminateProcess: %llx",GetSSDTFuncCurAddr(41));
}

```

网上 SSDT HOOK 的代码，动辄几百行的代码，感觉简直是在吓唬人。现在，我用一行代码凸显出 SSDT HOOK 的本质：

WIN32 内核：
KeServiceDescriptorTable->ServiceTableBase[Index] = 代理函数绝对地址
WIN64 内核：
KeServiceDescriptorTable->ServiceTableBase[Index] = 代理函数偏移地址

也就是说，在 WIN32 下只需要四行代码即可实现 SSDT HOOK，分别是：关闭内存写保护、保存旧地址、设置新地址、打卡内存写保护。而 WIN64 系统显得复杂些，还需要计算偏移地址、找出一块在位于 NTOSKRNL 空间里的废弃内存，并在这块废弃内存里进行二次跳转才能转到自己的处理函数。

二、UNHOOK SSDT

要恢复 SSDT，首先要获得 SSDT 各个函数的原始地址，而 SSDT 各个函数的原始地址，自然是存储在内核文件里的。于是，有了以下思路：

1. 获得内核里 KiServiceTable 的地址（变量名称：KiServiceTable）
2. 获得内核文件在内核里的加载地址（变量名称：NtosBase）
3. 获得内核文件在 PE32+结构体里的映像基址（变量名称：NtosImageBase）
4. 在自身进程里加载内核文件并取得映射地址（变量名称：NtosInProcess）
5. 计算出 KiServiceTable 和 NtosBase 之间的“距离”（变量名称：RVA）
6. 获得指定 INDEX 函数的地址（计算公式： $*(PULONGLONG)(NtosInProcess + RVA + 8 * index) - NtosImageBase + NtosBase$ ）

思路和 WIN32 下获得 SSDT 函数原始地址差异不大，接下来解释一下第六步的计算公式是怎么得来的。首先看一张 IDA 的截图：

```

.text:000000014007D818
.text:000000014007D900 KiServiceTable
.text:000000014007D900
.text:000000014007D908
.text:000000014007D910
.text:000000014007D918
.text:000000014007D920
.text:000000014007D928
.text:000000014007D930
.text:000000014007D938
.text:000000014007D940
.text:000000014007D948
.text:000000014007D950
.text:000000014007D958
.text:000000014007D960
.text:000000014007D968
.text:000000014007D970
.text:000000014007D978

```

```

db 0E8h dup(90h)
dq offset NtMapUserPhysicalPagesScatter ; DATA XREF: KiInitializeKernel+344j0
dq offset NtWaitForSingleObject
dq offset NtCallbackReturn
dq offset NtReadFile
dq offset NtDeviceIoControlFile
dq offset NtWriteFile
dq offset NtRemoveIoCompletion
dq offset NtReleaseSemaphore
dq offset NtReplyWaitReceivePort
dq offset NtReplyPort
dq offset NtSetInformationThread
dq offset NtSetEvent
dq offset NtClose
dq offset NtQueryObject
dq offset NtQueryInformationFile
dq offset NtOpenKey

```

可见，从文件中的 **KiServiceTable** 地址开始，每 8 个字节，存储一个函数的“理想地址”（之所以说是理想地址，是因为这个地址是基于『内核文件的映像基址 **NtosImageBase**』的，而不是基于『内核文件的加载基址 **NtosBase**』的）。因此，得到 $8 * \text{index}$ 。由于已经获得了 **KiServiceTable** 和 **NtosBase** 之间的“距离”（ $\text{RVA} = \text{KiServiceTable} - \text{NtosBase}$ ），也已知内核文件在自身进程里的映射地址（**NtosInProcess**），所以就能算出文件中的 **KiServiceTable** 的地址（ $\text{NtosInProcess} + \text{RVA}$ ）。所以，存储各个函数原始地址的文件地址就是： $\text{NtosInProcess} + \text{RVA} + 8 * \text{index}$ 。把这个地址的值取出来（长度为 8），就是： $*(\text{PULONGLONG})(\text{NtosInProcess} + \text{RVA} + 8 * \text{index})$ 。前面说了，由于得到的这个函数地址是理想地址，因为它假设的加载基址是 PE32+ 结构体里的成员 **ImageBase**（映像基址）的值。而实际上，内核文件的加载基址肯定不可能是这个值，所以还要减去内核文件的映像基址（**NtosImageBase**）再加上内核文件的实际加载基址（**NtosBase**）。接下来，给出每一步的具体实现过程的代码。

1. 获得 KiServiceTable 的地址

其实就是获得 **KeServiceDescriptorTable**→**ServiceTableBase** 的地址而已，具体知识之前已经讲过，这里就不赘述了，直接给出代码：

```
ULONGLONG GetKeServiceDescriptorTable64()
{
    char KiSystemServiceStart_pattern[13] = "\x8B\xF8\xC1\xEF\x07\x83\xE7\x20\x25\xFF\x0F\x00\x00";
    ULONGLONG CodeScanStart = (ULONGLONG)&_strnicmp;
    ULONGLONG CodeScanEnd = (ULONGLONG)&KdDebuggerNotPresent;
    UNICODE_STRING Symbol;
    ULONGLONG i, tbl_address, b;
    for (i = 0; i < CodeScanEnd - CodeScanStart; i++)
    {
        if (!memcmp((char*)(ULONGLONG)CodeScanStart + i, (char*)KiSystemServiceStart_pattern, 13))
        {
            for (b = 0; b < 50; b++)
            {
                tbl_address = (ULONGLONG)CodeScanStart + i + b;
                if (*(USHORT*)(ULONGLONG)tbl_address) == (USHORT)0x8d4c)
                    return ((ULONGLONG)tbl_address + 7) + *(LONG*)(tbl_address + 3);
            }
        }
    }
    return 0;
}

ULONG64 ssdt_base_address = GetKeServiceDescriptorTable64();
KiServiceTable = *(PULONGLONG)ssdt_base_address;
```

2. 获得内核文件在内核里的加载地址

这个本质上属于枚举内核模块，使用 **ZwQuerySystemInformation** 的

SystemModuleInformation 功能号实现。由于第一个加载的总是内核文件，所以直接获得 0 号模块的基址即可。另外，还要获得内核文件的名称，因为根据 CPU 核心数目等硬件条件的不同，内核文件的名称也是不尽相同的。

```

ULONGLONG GetNtosBaseAndPath(char *ModuleName)
{
    ULONG NeedSize, i, ModuleCount, BufferSize = 0x5000;
    PVOID pBuffer = NULL;
    ULONGLONG qwBase = 0;
    NTSTATUS Result;
    PSYSTEM_MODULE_INFORMATION pSystemModuleInformation;
    do
    {
        pBuffer = malloc( BufferSize );
        if( pBuffer == NULL )
        {
            return FALSE;
        }
        Result = ZwQuerySystemInformation( SystemModuleInformation, pBuffer, BufferSize,
        &NeedSize );
        if( Result == STATUS_INFO_LENGTH_MISMATCH )
        {
            free( pBuffer );
            BufferSize *= 2;
        }
        else if( !NT_SUCCESS(Result) )
        {
            free( pBuffer );
            return FALSE;
        }
    }
    while( Result == STATUS_INFO_LENGTH_MISMATCH );
    pSystemModuleInformation = (PSYSTEM_MODULE_INFORMATION)pBuffer;
    if( ModuleName!=NULL)
        strcpy( ModuleName, pSystemModuleInformation->Module[0]. ImageName+pSystemModuleInformat
        ion->Module[0]. ModuleNameOffset);
    qwBase=(ULONGLONG)pSystemModuleInformation->Module[0]. Base;
    free( pBuffer );
    return qwBase;
}

```

3. 获得内核文件的映像基址

这个直接解析 PE32+ 文件的结构即可。

```

DWORD FileLen(char *filename)
{

```

```

WIN32_FIND_DATA fileInfo={0};
DWORD fileSize=0;
HANDLE hFind;
hFind = FindFirstFileA(filename,&fileInfo);
if(hFind != INVALID_HANDLE_VALUE)
{
    fileSize = fileInfo.nFileSizeLow;
    FindClose(hFind);
}
return fileSize;
}

CHAR *LoadDllContext(char *filename)
{
    DWORD dwReadWrite, LenOfFile=FileLen(filename);
    HANDLE hFile = CreateFileA(filename, GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ |
FILE_SHARE_WRITE, 0, OPEN_EXISTING, 0, 0);
    if (hFile != INVALID_HANDLE_VALUE)
    {
        PCHAR buffer=(PCHAR)malloc(LenOfFile);
        SetFilePointer(hFile, 0, 0, FILE_BEGIN);
        ReadFile(hFile, buffer, LenOfFile, &dwReadWrite, 0);
        CloseHandle(hFile);
        return buffer;
    }
    return NULL;
}

VOID GetNtosImageBase()
{
    PIMAGE_NT_HEADERS64 pinths64;
    PIMAGE_DOS_HEADER pdih;
    char *NtosFileData=NULL;
    NtosFileData=LoadDllContext(NtosName);
    pdih=(PIMAGE_DOS_HEADER)NtosFileData;
    pinths64=(PIMAGE_NT_HEADERS64) (NtosFileData+pdih->e_lfanew);
    NtosImageBase=pinths64->OptionalHeader.ImageBase;
    printf("ImageBase: %llx\n",NtosImageBase);
}

```

4/5/6. 获得 SSDT 函数的原始地址

原理已经在前面解释过，这里直接给出代码。

```

ULONGLONG GetFunctionOriginalAddress(DWORD index)
{
    if ( NtosInProcess==0 )
        NtosInProcess = (ULONGLONG)LoadLibraryExA(NtosName, 0,

```

```

DONT_RESOLVE_DLL_REFERENCES);

ULONGLONG RVA=KiServiceTable-NtosBase;

ULONGLONG temp=(PULONGLONG) (NtosInProcess+RVA+8*(ULONGLONG) index);

ULONGLONG RVA_index=temp-NtosImageBase;

return RVA_index+NtosBase;

}

```

接下来测试一下效果，在测试前，运行 SSDT HOOK NtTerminateProcess 的 DEMO（检测出了 SSDT 的异常项）。

```

C:\temp\EnumSSDT\EnumSSDT_x64_exe.exe
0x174 fffff80001b3e430 fffff80001b3e430 NtShutdownSystem
0x175 fffff80001976398 fffff80001976398 NtShutdownWorkerFactory
0x176 fffff800017d2c20 fffff800017d2c20 NtSignalAndWaitForSingleObject
0x177 fffff80001b0e7a0 fffff80001b0e7a0 NtSinglePhaseReject
0x178 fffff80001b1d240 fffff80001b1d240 NtStartProfile
0x179 fffff80001ae9450 fffff80001ae9450 NtStopProfile
0x17A fffff80001b0c480 fffff80001b0c480 NtSuspendProcess
0x17B fffff8000190f328 fffff8000190f328 NtSuspendThread
0x17C fffff8000194d33c fffff8000194d33c NtSystemDebugControl
0x17D fffff8000191fa90 fffff8000191fa90 NtTerminateJobObject
0x029! fffff800016bcf00 fffff80001978b80 NtTerminateProcess
0x050 fffff800019a3498 fffff800019a3498 NtTerminateThread
0x17E fffff800019881b4 fffff800019881b4 NtTestAlert
0x17F fffff800017bb770 fffff800017bb770 NtThawRegistry
0x180 fffff80001a93b70 fffff80001a93b70 NtThawTransactions
0x181 fffff80001965640 fffff80001965640 NtTraceControl
0x05B fffff800016a2ec8 fffff800016a2ec8 NtTraceEvent
0x182 fffff80001ae34c0 fffff80001ae34c0 NtTranslateFilePath
0x183 fffff80001a448d0 fffff80001a448d0 NtUmsThreadYield
0x184 fffff80001aa5dd0 fffff80001aa5dd0 NtUnloadDriver
0x185 fffff80001948a44 fffff80001948a44 NtUnloadKey
0x186 fffff80001947bc0 fffff80001947bc0 NtUnloadKey2
0x187 fffff80001b0df80 fffff80001b0df80 NtUnloadKeyEx
0x188 fffff80001930180 fffff80001930180 NtUnlockFile
0x189 fffff800017d2150 fffff800017d2150 NtUnlockVirtualMemory

```

检测出了异常的项目就需要恢复。其实恢复 SSDT 本质上和挂钩 SSDT 本质上没有不同，都是在 KiServiceTable 的指定偏移处写入一个 INT32 值。代码如下：

```

LONG GetOffsetAddress(ULONGLONG FuncAddr)
{
    LONG dwtmp=0;
    PULONG ServiceTableBase=NULL;
    if(KeServiceDescriptorTable==NULL)
        KeServiceDescriptorTable=(PSYSTEM_SERVICE_TABLE)GetKeServiceDescriptorTable64();
    ServiceTableBase=(PULONG) KeServiceDescriptorTable->ServiceTableBase;
    dwtmp=(LONG) (FuncAddr-(ULONGLONG) ServiceTableBase);
    return dwtmp<<4;
}

VOID UnHookSSDT(ULONG id, ULONGLONG FuncAddr)    //传入正确的地址
{
    KIRQL irql;
    LONG dwtmp;

```

```

PULONG ServiceTableBase=NULL;

dwtmp=GetOffsetAddress(FuncAddr);

ServiceTableBase=(PULONG)KeServiceDescriptorTable->ServiceTableBase;

irq1=WPOFFx64();

ServiceTableBase[id]=dwtmp;    //核心就这一句

WPONx64(irq1);

}

```

接下来测试效果（输入要恢复的函数的 Index）：

```

C:\temp\EnumSSDT\EnumSSDT_x64_exe.exe
0x185 fffff80001948a44 fffff80001948a44 NtUnloadKey
0x186 fffff80001947bc0 fffff80001947bc0 NtUnloadKey2
0x187 fffff80001b0df80 fffff80001b0df80 NtUnloadKeyEx
0x188 fffff80001930180 fffff80001930180 NtUnlockFile
0x189 fffff800017d2150 fffff800017d2150 NtUnlockVirtualMemory
0x027 fffff800019d31fc fffff800019d31fc NtUnmapViewOfSection
0x18a fffff80001b22460 fffff80001b22460 NtUdmControl
0x18b fffff80001af5cb0 fffff80001af5cb0 NtWaitForDebugEvent
0x18c fffff800019a692c fffff800019a692c NtWaitForKeyedEvent
0x058 fffff800019b63f0 fffff800019b63f0 NtWaitForMultipleObjects
0x017 fffff800019e0f80 fffff800019e0f80 NtWaitForMultipleObjects32
0x001 fffff800019b4a00 fffff800019b4a00 NtWaitForSingleObject
0x18d fffff800016cd010 fffff800016cd010 NtWaitForWorkViaWorkerFactory
0x18e fffff80001adcb00 fffff80001adcb00 NtWaitHighEventPair
0x18f fffff80001adcb90 fffff80001adcb90 NtWaitLowEventPair
0x190 fffff800016a4fc4 fffff800016a4fc4 NtWorkerFactoryWorkerReady
0x005 fffff800019d0ee0 fffff800019d0ee0 NtWriteFile
0x018 fffff80001b156c0 fffff80001b156c0 NtWriteFileGather
0x054 fffff80001b23120 fffff80001b23120 NtWriteRequestData
0x037 fffff80001964668 fffff80001964668 NtWriteVirtualMemory
0x043 fffff80001686c20 fffff80001686c20 NtYieldExecution

Total of SSDT function: 401

Input SSDT function index which you want to unhook (like 0x29): 0x29

```

再次运行这个枚举 SSDT 的程序，发现 NtTerminateProcess 项目已经没异常了：

```

C:\temp\EnumSSDT\EnumSSDT_x64_exe.exe
0x174 fffff80001b3e430 fffff80001b3e430 NtShutdownSystem
0x175 fffff80001976398 fffff80001976398 NtShutdownWorkerFactory
0x176 fffff800017d2c20 fffff800017d2c20 NtSignalAndWaitForSingleObject
0x177 fffff80001b0e7a0 fffff80001b0e7a0 NtSinglePhaseReject
0x178 fffff80001b1d240 fffff80001b1d240 NtStartProfile
0x179 fffff80001ae9450 fffff80001ae9450 NtStopProfile
0x17a fffff80001b0c480 fffff80001b0c480 NtSuspendProcess
0x17b fffff8000190f328 fffff8000190f328 NtSuspendThread
0x17c fffff8000194d33c fffff8000194d33c NtSystemDebugControl
0x17d fffff8000191fa90 fffff8000191fa90 NtTerminateJobObject
0x029 fffff80001978b80 fffff80001978b80 NtTerminateProcess
0x050 fffff800019a3498 fffff800019a3498 NtTerminateThread
0x17e fffff800019881b4 fffff800019881b4 NtTestAlert
0x17f fffff800017bb770 fffff800017bb770 NtThawRegistry
0x180 fffff80001a93b70 fffff80001a93b70 NtThawTransactions
0x181 fffff80001965640 fffff80001965640 NtTraceControl
0x05b fffff800016a2ec8 fffff800016a2ec8 NtTraceEvent
0x182 fffff80001ae34c0 fffff80001ae34c0 NtTranslateFilePath
0x183 fffff80001a448d0 fffff80001a448d0 NtUmsThreadYield
0x184 fffff80001aa5dd0 fffff80001aa5dd0 NtUnloadDriver
0x185 fffff80001948a44 fffff80001948a44 NtUnloadKey
0x186 fffff80001947bc0 fffff80001947bc0 NtUnloadKey2
0x187 fffff80001b0df80 fffff80001b0df80 NtUnloadKeyEx
0x188 fffff80001930180 fffff80001930180 NtUnlockFile
0x189 fffff800017d2150 fffff800017d2150 NtUnlockVirtualMemory

```

课后作业：写一个 HOOK NtCreateFile 的代码（在代理函数里打印一句

HELLOWORLD 就返回原函数)，发现问题并在论坛上提出。附件中有一个带 UNHOOK 功能的 SSDT 管理器，可以动态获得每个 SSDT 函数的原始地址和当前地址。