

SSDT 中文名称为系统服务描述符表，该表的作用是将Ring3应用层与Ring0内核层，两者的API函数连接起来，起到承上启下的作用，SSDT并不仅仅只包含一个庞大的地址索引表，它还包含着一些其它有用的信息，诸如地址索引的基址、服务函数个数等，SSDT 通过修改此表的函数地址可以对常用 Windows 函数进行内核级的Hook，从而实现对一些核心的系统动作进行过滤、监控的目的，接下来将演示如何通过编写简单的驱动程序，来实现搜索 SSDT 函数的地址，并能够实现简单的内核 Hook 挂钩。

在开始编写驱动之前，我们先来分析一下Ring3到Ring0是如何协作的，这里通过C语言调用 `OpenProcess` 函数，并分析它的执行过程，先来创建一个C程序。

```
#include <windows.h>

int main(int argc, char* argv[])
{
    HANDLE handle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, 2548);
    return 0;
}
```

通过VC6编译器编译，并使用OD载入程序，找到程序的OEP，分析第一次调用，可以看到CALL的地址是 `<&KERNEL32.OpenProcess>` 此处我们F7直接跟进这个CALL。

```
00401028 |. 8BF4          mov     esi, esp
0040102A |. 68 F4090000   push    0x9F4
/ProcessId = 0x9F4
0040102F |. 6A 00         push    0x0
|Inheritable = FALSE
00401031 |. 68 FF0F1F00   push    0x1F0FFF
|Access = PROCESS_ALL_ACCESS
00401036 |. FF15 4CA14200 call     dword ptr [<&KERNEL32.OpenProcess>]
\OpenProcess
```

此时我们已经进入到了 `00401036` 这个地址中，观察下方的代码，发现其调用了 `&ntdll.NtOpenProcess` 这个函数，我们继续F7跟进。

```
75BC83C3 50          push    eax
75BC83C4 8975 FC     mov     dword ptr [ebp-0x4], esi
75BC83C7 C745 E0 18000000>mov     dword ptr [ebp-0x20], 0x18
75BC83CE 8975 E4     mov     dword ptr [ebp-0x1C], esi
75BC83D1 8975 E8     mov     dword ptr [ebp-0x18], esi
75BC83D4 8975 F0     mov     dword ptr [ebp-0x10], esi
75BC83D7 8975 F4     mov     dword ptr [ebp-0xC], esi
75BC83DA FF15 D411BC75 call     dword ptr [<&ntdll.NtOpenProcess>]
ntdll.ZwOpenProcess
```

当我们进入到 `NtOpenProcess` 这个函数时，会看到以下代码，其中 `0xBE` 将其转换成十进制是 `190`

```
77A05D88 > B8 BE000000   mov     eax, 0xBE
77A05D8D BA 0003FE7F   mov     edx, 0x7FFE0300
77A05D92 FF12         call    dword ptr [edx]
77A05D94 C2 1000      retn     0x10
```

dsqjgqqb

进程 | 驱动模块 | 内核 | 内核钩子 | 应用层钩子 | 网络 | 注册表 | 文件 | 启动信息 | 系统杂项 | 电脑体检 | 配置 | 关于

SSDT | ShadowSSDT | FSD | 键盘 | I8042prt | 鼠标 | Partmgr | Disk | Atapi | Acpi | Scsi | 内核钩子 | Object钩子 | 系统中断表

序号	函数名称	当前函数地址	Hook	原始函数地址	当前函数地址所在模块
189	NtOpenPrivateNamespace	0x83E15E07	-	0x83E15E07	C:\Windows\system32\ntkrnlp.exe
190	NtOpenProcess	0x83E4D9DC	-	0x83E4D9DC	C:\Windows\system32\ntkrnlp.exe
191	NtOpenProcessToken	0x83E9FFFF	-	0x83E9FFFF	C:\Windows\system32\ntkrnlp.exe
192	NtOpenProcessTokenEx	0x83E8DB37	-	0x83E8DB37	C:\Windows\system32\ntkrnlp.exe

通过使用Xuetz工具对比，可以发现这个 0xBE 正好就是 NtOpenProcess 函数在内核中的调用号，此时我们继续F7进入到 `call dword ptr [edx]` 地址中，可以看到以下代码片段。

```

77A070AC    8D6424 00    lea     esp, dword ptr [esp]
77A070B0 > 8BD4        mov     edx, esp
77A070B2    0F34        sysenter
77A070B4 > C3         retn

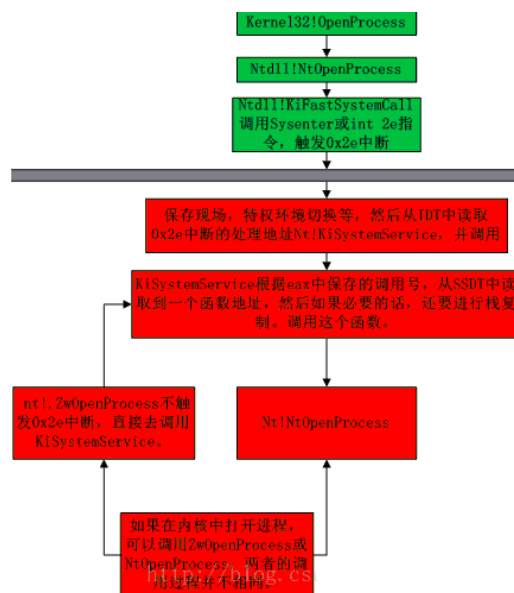
```

发现已经到达Ring3层的终点了，其中 `sysenter` 指令就是用来快速调用一个 Ring0 层的系统过程，简单来说就是将用户层代码向内核层发起的系统调用，由 `ntoskrnl.exe` 程序向内核发送IO请求，然后内核与驱动程序返回执行的结果。

网上找到一张图，可以很好的解释这个调用的顺序。

进入用户层： `kernel32(OpenProcess) -> ntdll(NTOpenProcess)->ntdll(SysEnter)`

进入内核层： `ntoskrnl.exe(nt!ZwOpenProcess) -> ntoskrnl.exe(nt!KiSystemService) -> ntoskrnl.exe(nt!NtOpenProcess)`



读取 SSDT 获得函数地址

上面的实验我们通过一个函数的调用流程了解到了用户层与内核层的通信过程，其中提到了SSDT索引号的相关概念，SSDT索引号在系统中是固定不变的，利用这个特性就可以定位到原始API函数地址。

Windows 系统提供的SSDT表其作用就是方便应用层之间API的调用，所有的API调用都会转到SSDT这张表中进行参考，这样就能够使不同的API调用全部都转到对应的SSDT表中，从而方便管理。

在SSDT表中有一个 `KeServiceDescriptorTable` 的结构，该结构是由内核导出的表，该表拥有一个指针，指向SSDT中包含由 `Ntoskrnl.exe` 实现的核心系统服务的相应部分，`ntoskrnl.exe` 中导出了 `PSERVICE_DESCRIPTOR_TABLE` 类型指针，变量为 `KeServiceDescriptorTable` 它是内核的主要组成部分，该表结构如下：

```
typedef struct _SERVICE_DESCRIPTOR_TABLE
{
    PULONG ServiceTableBase;           // SSDT 指针，服务表基址
    PULONG ServiceCounterTableBase;    // 包含 SSDT 中每个服务被调用次数的计数器
    ULONG  NumberOfService;           // SSDT 索引数目
    PCHAR  ParamTableBase;             // 包含每个系统服务参数字节数表的基地址-系统服务参数表
} SERVICE_DESCRIPTOR_TABLE, *PSERVICE_DESCRIPTOR_TABLE;
```

表结构中的 `SERVICE_DESCRIPTOR_TABLE` 包含了所有内核导出函数的地址，在32位系统中每个地址长度为4个字节，所以要获得某个函数在SSDT中的偏移量，可以使用 `KeServiceDescriptorTable->ServiceTableBase + 函数ID * 4` 的方式来得到。

上方都是一些理论部分，接着我们通过使用WinDBG来具体查看一些这个表的一些结构信息，此处测试系统是XP

打开WinDBG调试器，选择【File -> Kernel Debug -> Local -> OK】输入以下命令完成符号文件的加载。

```
1kd> .sympath srv*d:\symbols*http://msdl.microsoft.com/download/symbol
1kd> .reload

Connected to Windows XP 2600 x86 compatible target at (Sat Sep 21 07:23:56.796 2019
(UTC + 8:00)), ptr64 FALSE
Loading Kernel Symbols
```

当符号文件加载完成以后，在命令窗口输入 `dd KeServiceDescriptorTable` 命令。

```
1kd> dd KeServiceDescriptorTable

8055d700  80505570 00000000 0000011c 805059e4
8055d710  00000000 00000000 00000000 00000000
8055d720  00000000 00000000 00000000 00000000
8055d730  00000000 00000000 00000000 00000000
8055d740  00000002 00002710 bf80c401 00000000
8055d750  b69c4a80 b8e4ab60 8ad620f0 806f80c0
8055d760  00000000 00000000 fee134ac ffffffff
8055d770  5a5a626c 01d56f51 00000000 00000000
```

从以上结构定义可看出，SSDT的首地址为 `80505570` 该地址对应结构中的 `ServiceTableBase`，可索引的函数有 `11c` 对应结构中的 `NumberOfService`，由于SSDT是数组结构，所以里面存放了所有的 `nt!nt*` 函数的地址，使用 `dd kiservicetable` 查看 SSDT 下的所有数组成员信息。

```
1kd> dd kiservicetable
```

```
80505570 805a5664 805f23ea 805f5c20 805f241c
80505580 805f5c5a 805f2452 805f5c9e 805f5ce2
80505590 80616e80 806180e4 805ed7e8 805ed440
805055a0 805d5c0c 805d5bbc 806174a6 805b6fea
805055b0 80616ac2 805a9aee 805b15fe 805d76d0
805055c0 805028e8 805c96a4 80577b04 80539d88
805055d0 80610090 805bd564 805f615a 80624e3a
805055e0 805fa66e 805a5d52 8062508e 805a5604
```

为了能够定位到我们所需要的函数调用号，我们还需要手动查找一下 `ZwOpenProcess` 这个函数的ID号，可以使用WinDBG来获取，如下显示调用号为 `7A`

```
1kd> u ZwOpenProcess
```

```
ntdll!ZwOpenProcess:
7c92d5fe b87a000000      mov     eax,7Ah
7c92d603 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
7c92d608 ff12             call    dword ptr [edx]
7c92d60a c21000          ret     10h
7c92d60d 90              nop
```

上方代码可以得到 `nt!NtOpenProcess` 地址在SSDT表中的索引号。

```
1kd> dd kiservicetable +0x7A * 4 1 1
80502d74 805c2296
```

```
1kd> u 805c2296
```

```
nt!NtOpenProcess:
805c2296 68c4000000      push    0c4h
805c229b 68a8aa4d80      push    offset nt!ObwatchHandles+0x25c (804daaa8)
805c22a0 e86b6cf7ff      call    nt!_SEH_prolog (80538f10)
```

如果符号文件没有加载成功，可以使用下面的方式来查询，找到结构的首地址，然后与函数编号相加来获取。

```

1kd> dd KeServiceDescriptorTable
80553fa0 80502b8c 00000000 0000011c 80503000

1kd> dd 80502b8c+0x7A*4
80502d74 805c2296 805e49fc 805e4660 805a0722

1kd> u 805c2296
nt!NtOpenProcess:
805c2296 68c4000000      push    0C4h
805c229b 68a8aa4d80      push    offset nt!FsRtlLegalAnsiCharacterArray+0x2008
(804daaa8)
805c22a0 e86b6cf7ff      call    nt!wctomb+0x45 (80538f10)
805c22a5 33f6           xor     esi,esi

```

注意：在验证的时候需要请关闭杀毒软件，因为杀毒软件会Hook这些地址来达到防御的目的，Hook后这些地址会发生变化无法完成整个查询过程，另外 ZwOpenProcess 与 NtOpenProcess 其实是一回事。

编写驱动程序：接着我们分别使用C语言和汇编实现读取系统的SSDT表，此处使用的系统是Win7，由于Win7系统默认情况下本地内核调试功能被屏蔽了，所以必须在控制台下运行 `bcdedit -debug on` 命令并且重启来进入调试模式。

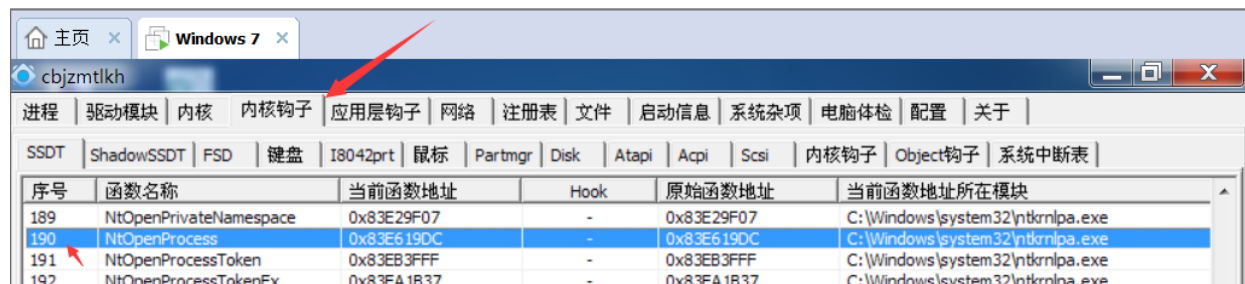
进入调试模式后，我们首先通过WinDBG调试器，来查询一下 ZwOpenProcess 函数的调用号，执行命令如下。

```

1kd> u ZwOpenProcess
nt!ZwOpenProcess:
83c8a62c b8be000000      mov     eax,0BEh
83c8a631 8d542404      lea     edx,[esp+4]
83c8a635 9c           pushfd
83c8a636 6a08         push    8
83c8a638 e8b1190000      call    nt!ZwYieldExecution+0x95a (83c8bfee)
83c8a63d c21000       ret     10h
nt!ZwOpenProcessToken:
83c8a640 b8bf000000      mov     eax,0BFh
83c8a645 8d542404      lea     edx,[esp+4]

```

上方代码中可以看到 `mov eax,0BEh` 其中的BE就是 ZwOpenProcess 函数在当前系统下的调用号，我们将其转换为十进制是 190 当然也可以使用Xuetr等工具来查询。



序号	函数名称	当前函数地址	Hook	原始函数地址	当前函数地址所在模块
189	NtOpenPrivateNamespace	0x83E29F07	-	0x83E29F07	C:\Windows\system32\ntkrnlpa.exe
190	NtOpenProcess	0x83E619DC	-	0x83E619DC	C:\Windows\system32\ntkrnlpa.exe
191	NtOpenProcessToken	0x83EB3FFF	-	0x83EB3FFF	C:\Windows\system32\ntkrnlpa.exe
192	NtOpenProcessTokenEx	0x83EA1B37	-	0x83EA1B37	C:\Windows\system32\ntkrnlpa.exe

接着我们来编译以下驱动代码，重要的内容已经备注好了，唯一需要更改的地方是 `SSDT_Adr = (PLONG)(STB_addr + 0x7A * 4)`；其中的0x7A需要改为0xBE

```

#include <ntddk.h>

//声明:服务描述表 结构 4个参数
typedef struct _ServiceDescriptorTable {
    PULONG ServiceTableBase;           // 服务表基址
    PULONG ServiceCounterTable;        // 服务计数器
    ULONG NumberOfServices;            // 服务的数目
    PCHAR ParamTableBase;              // 系统服务参数表
}*PServiceDescriptorTable;

// 用指针PServiceDescriptorTable指向: _ServiceDescriptorTable服务描述表结构
// 必须extern "C" , 因为文件为CPP
extern "C" PServiceDescriptorTable KeServiceDescriptorTable;

void UnloadDriver(PDRIVER_OBJECT pDriver)
{
    DbgPrint("驱动已卸载!\n");
}

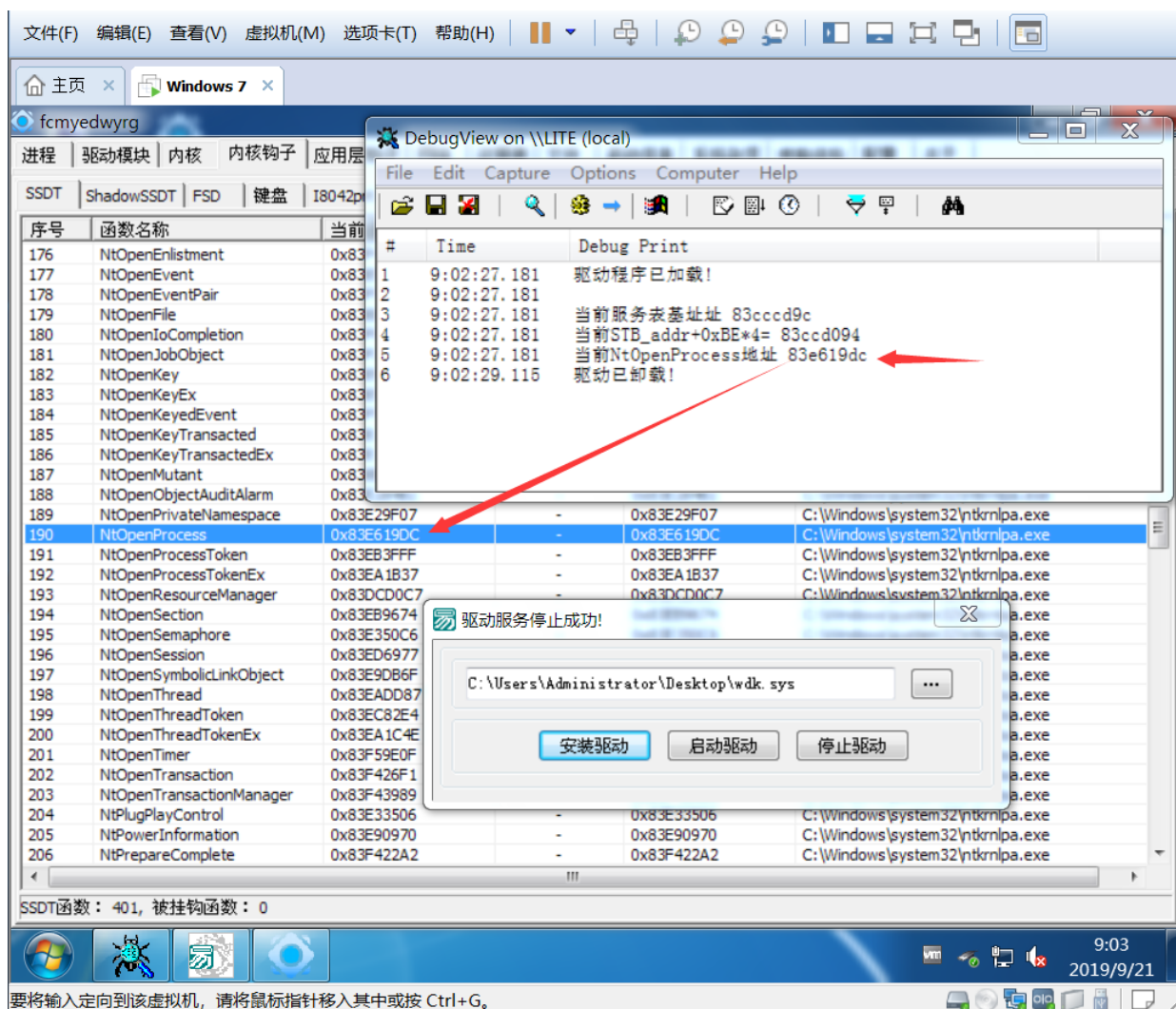
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING str)
{
    /*SSDT_Adr 存放系统描述符号表地址。
    //STB_addr 存放 ServiceTableBase 服务表基址。
    //SDT_Nt 函数 NtOpenProcess的当前地址。
    LONG *SSDT_Adr, STB_addr, SSDT_NtOpenProcess_Adr;

    DbgPrint("驱动程序已加载! \n\r");
    STB_addr = (LONG)KeServiceDescriptorTable->ServiceTableBase;
    DbgPrint("当前服务表基址址 %x \n", STB_addr);
    SSDT_Adr = (PLONG)(STB_addr + 0xBE * 4);           // 此处需要修改
    DbgPrint("当前STB_addr+0xBE*4= %x \n", SSDT_Adr);
    SSDT_NtOpenProcess_Adr = *SSDT_Adr;
    DbgPrint("当前NtOpenProcess地址 %x \n", SSDT_NtOpenProcess_Adr);

    pDriver->DriverUnload = UnloadDriver;
    return STATUS_SUCCESS;
}

```

编译程序以后，将其拖入虚拟机，打开DebugView工具，然后加载这个驱动程序，观察是否能够读取到我们想要的数据库。



再次编译下方的汇编版本，调试观察，结果与C版本相同。

```
#include <ntddk.h>

extern "C" LONG KeServiceDescriptorTable;

void UnloadDriver(PDRIVER_OBJECT pDriver)
{
    DbgPrint("驱动卸载成功\n\r");
}

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING str)
{
    ULONG SSDT_Addr;
    DbgPrint("驱动加载成功! \n");
    __asm
    {
        push ebx
        push eax
        mov ebx, KeServiceDescriptorTable // 系统描述符号表的地址
        mov ebx, [ebx] // 取服务表基址给EBX
        mov eax, 0xBE // NtOpenProcess=转成十六进制等于BE
        imul eax, eax, 4 // eax=eax*4 -> 7a*4=1e8
        add ebx, eax // eax=1e8与服务表基址EBX相加
    }
}
```

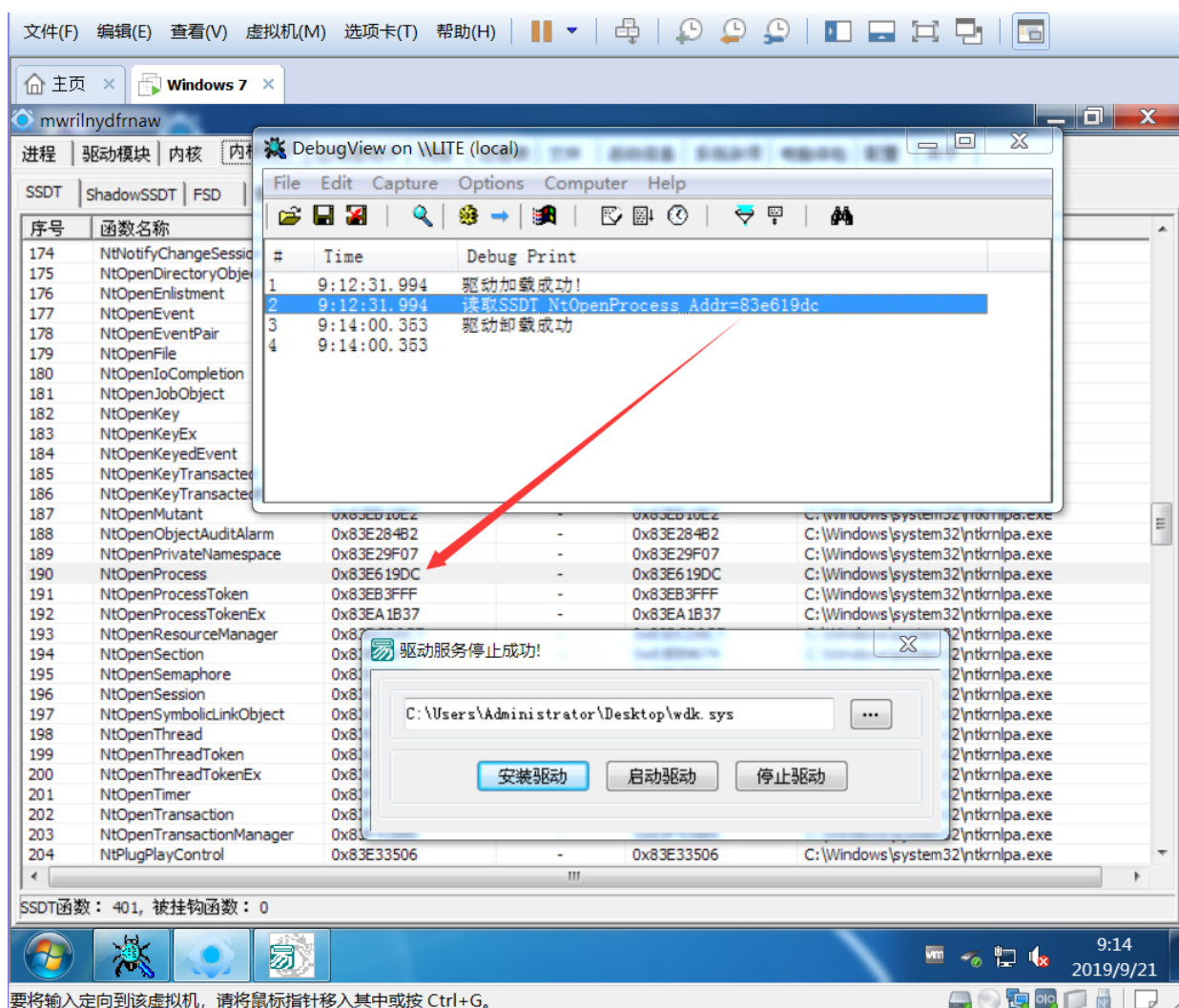


```

mov ebx, [ebx]                // 取ebx里面的内容给EBX
mov SSDT_Addr, ebx            // 将得到的基址给变量
pop eax
pop ebx
}
DbgPrint("读取SSDT_NtOpenProcess_Addr=%0x \n", SSDT_Addr);

pDriver->DriverUnload = UnloadDriver;
return STATUS_SUCCESS;
}

```



Hook 挂钩内核函数

挂钩函数有多种用途, 下面的第一种挂钩方式可以实现对特定内核函数的重写, 而第二种挂钩方式则可以用于驱动保护。

Hook挂钩重写函数: 挂钩代码如下, 关键点已经备注清楚了, 编译这段代码, 并放入虚拟机执行。

```

#include <ntddk.h>

#ifdef __cplusplus

```



```

extern "C" NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING
    RegistryPath);
#endif

void SSDTHookUnload(PDRIVER_OBJECT);
// *****
//这个结构是服务描述表
typedef struct _SERVICE_DESCRIPTOR_TABLE
{
    PULONG ServiceTableBase;
    PULONG ServiceCounterTableBase;
    ULONG NumberOfService;
    PCHAR ParamTableBase;
}SERVICE_DESCRIPTOR_TABLE, *PSERVICE_DESCRIPTOR_TALBE;
extern "C" PSERVICE_DESCRIPTOR_TALBE KeServiceDescriptorTable;

// *****
// 此处为函数的原型声明部分,可通过百度查询到
typedef NTSTATUS(*NtOpenProcessEx)(
    OUT PHANDLE ProcessHandle,
    IN ACCESS_MASK AccessMask,
    IN PVOID ObjectAttributes,
    IN PCLIENT_ID ClientId);

NtOpenProcessEx uNtOpenProcessEx = NULL;          // 函数指针,存放原始函数地址
ULONG uNtOpenProcessExAddr = 0;                   // 在SSDT中的函数地址的指针

// *****
// SSDT 内核内存页默认只读,需要修改CR0 WP位实现读写
// CR0 的第16位是WP位,为0可读写,为1则只读
void REMOVE_ONLY_READ()
{
    __asm
    {
        push eax
        mov eax, CR0
        and eax, ~10000h //16th bit is 0
        mov CR0, eax
        pop eax
    }
}

void RESET_ONLY_READ()
{
    __asm
    {
        push eax
        mov eax, CR0
        or eax, 10000h //16th bit is 1
        mov CR0, eax
        pop eax
    }
}

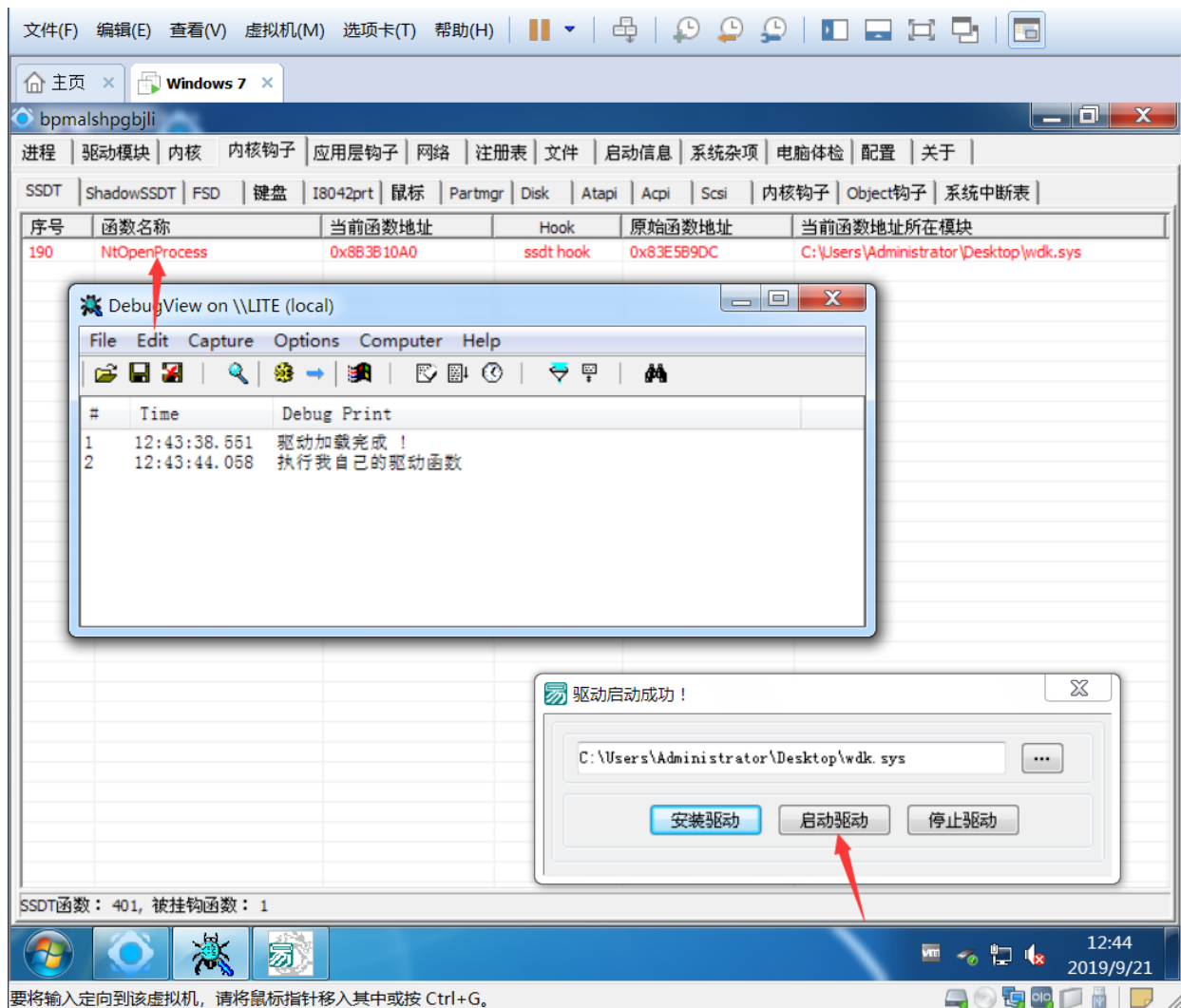
```

```

}
// *****
// 此处就是我们DIY的函数,声明要和NtOpenProcessEx保持一致.
NTSTATUS MyNtOpenProcessEx(
    OUT PHANDLE ProcessHandle,
    IN ACCESS_MASK AccessMask,
    IN PVOID ObjectAttributes,
    IN PCLIENT_ID ClientId)
{
    //DbgPrint("执行我自己的驱动函数\r\n");
    NTSTATUS Status = STATUS_SUCCESS;
    Status = u!NtOpenProcessEx(
        ProcessHandle,
        AccessMask,
        ObjectAttributes,
        ClientId
    );
    return Status;
}
// *****
VOID HookOpenProcess()
{
    ULONG u!Ssdt = 0;
    u!Ssdt = (ULONG)KeServiceDescriptorTable->ServiceTableBase;           // 读取
到SSDT表的基地址
    u!NtOpenProcessExAddr = u!Ssdt + 0xBE * 4;                             // 索引
到指定的函数
    u!NtOpenProcessEx = (NtOpenProcessEx)*(PULONG)u!NtOpenProcessExAddr;   // 存储
原始函数地址
    REMOVE_ONLY_READ();                                                     // 关闭
只读保护
    *(PULONG)u!NtOpenProcessExAddr = (ULONG)MyNtOpenProcessEx;           // 将新
函数地址赋值
    RESET_ONLY_READ();                                                      // 开启
只读保护
}
// *****
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING
    RegistryPath)
{
    DbgPrint("驱动加载完成 !\n");
    DriverObject->DriverUnload = SSDTHookUnload;
    HookOpenProcess();
    return STATUS_SUCCESS;
}
void SSDTHookUnload(IN PDRIVER_OBJECT DriverObject)
{
    REMOVE_ONLY_READ();
    *(PULONG)u!NtOpenProcessExAddr = (ULONG)u!NtOpenProcessEx;
    RESET_ONLY_READ();
    DbgPrint("驱动卸载完成 !\n");
}

```

当驱动被加载时，可以通过Xuetr查看到内核SSDT层出现了红色的钩子。



驱动进程保护：进程的创建离不开 `zwTerminateProcess` 这个函数的支持，所以我们只需要Hook这个函数并在其内部判断是否是计算器进程，如果是则返回错误，否则返回原始调用，即可完成进程保护，这里就不演示了，核心伪代码如下。

```
#define EXE_Name "calc.exe" // 要检索的进程名

PEPROCESS processePROCESS = NULL; // 保存访问者的EPROCESS
ANSI_STRING p_StrName1, p_StrName2; // 保存进程名称

__declspec(naked) VOID inline_Hook()
{
    processePROCESS = IoGetCurrentProcess(); // 获取调用者的EPROCESS保存
    DbgPrint("EPROCESS=%x", processePROCESS); // 打印出来

    // 通过遍历将调用者名字保存到p_StrName1中,下方+0x174是表结构 ImageFileName 的偏移地址
    RtlInitAnsiString(&p_StrName1, (PUCHAR)processePROCESS + 0x174);

    // 将欲对比的字符串保存到p_StrName2中,初始化ANSI字符串
    RtlInitAnsiString(&p_StrName2, EXE_Name);
    // 判断是否相等,相等则说明是calc.exe进程,我们直接返回假
    if (RtlCompareString(&p_StrName1, &p_StrName2, TRUE) == 0)
```

```
{  
    // 相等说明是calc进程调用了该函数  
    // 直接返回假,否则执行原函数  
}  
}
```