这节课我们来讲 SSDT。我估计 SSDT 这个词对很多底层爱好者都有特殊的含义，绝对不仅仅是“系统服务描述表”这么简单，相信不少人都是从玩 SSDT HOOK 开始玩 WINDOWS 内核的，至少我就是如此。好了，扯淡的话就不说了，说多了估计有读者会拿砖头拍我。言归正传，本文只解决两个问题。第一，如何在内核里动态获得 SSDT 的基址；第二，如何在内核里动态获得 SSDT 函数的地址。

在 WIN32 下，第一个问题就根本不是问题，因为 KeServiceDescriptorTable 直接被导出了。但是 WIN64 下 KeServiceDescriptorTable 没有被导出。所以我们必须搜索得到它的地址。首先反汇编一下 KiSystemCall64：

```
lkd> uf KiSystemCall64
Flow analysis was incomplete, some code may be missing
nt!KiSystemCall64:
fffff800`03cc7ec0 0f01f8          swapgs
fffff800`03cc7ec3 654889242510000000 mov   qword ptr gs:[10h],rsp
fffff800`03cc7ecc 65488b2425a8010000 mov   rsp,qword ptr gs:[1A8h]
fffff800`03cc7ed5 6a2b            push    2Bh
fffff800`03cc7ed7 65ff342510000000 push    qword ptr gs:[10h]
fffff800`03cc7edf 4153            push    r11
fffff800`03cc7ee1 6a33            push    33h
fffff800`03cc7ee3 51              push    rcx
fffff800`03cc7ee4 498bca          mov     rcx,r10
fffff800`03cc7ee7 4883ec08        sub     rsp,8
fffff800`03cc7eeb 55              push    rbp
fffff800`03cc7eec 4881ec58010000  sub     rsp,158h
fffff800`03cc7ef3 488dac2480000000 lea     rbp,[rsp+80h]
fffff800`03cc7efb 48899dc0000000  mov     qword ptr [rbp+0C0h],rbx
fffff800`03cc7f02 4889bdc8000000  mov     qword ptr [rbp+0C8h],rdi
fffff800`03cc7f09 4889b5d0000000  mov     qword ptr [rbp+0D0h],rsi
fffff800`03cc7f10 c645ab02        mov     byte ptr [rbp-55h],2
fffff800`03cc7f14 65488b1c2588010000 mov   rbx,qword ptr gs:[188h]
fffff800`03cc7f1d 0f0d8bd8010000  prefetchw [rbx+1D8h]
fffff800`03cc7f24 0fae5dac        stmxcsr dword ptr [rbp-54h]
fffff800`03cc7f28 650fae142580010000 ldmxcsr dword ptr gs:[180h]
fffff800`03cc7f31 807b0300        cmp     byte ptr [rbx+3],0
fffff800`03cc7f35 66c785800000000000 mov   word ptr [rbp+80h],0
fffff800`03cc7f3e 0f848c000000    je      nt!KiSystemCall64+0x110 (fffff800`03cc7fd0)
【省略大量无关代码】
nt!KiSystemCall64+0x110:
fffff800`03cc7fd0 fb              sti
fffff800`03cc7fd1 48898be0010000  mov     qword ptr [rbx+1E0h],rcx
fffff800`03cc7fd8 8983f8010000    mov     dword ptr [rbx+1F8h],eax
fffff800`03cc7fde 4889a3d8010000  mov     qword ptr [rbx+1D8h],rsp
fffff800`03cc7fe5 8bf8            mov     edi,eax
fffff800`03cc7fe7 c1ef07          shr     edi,7
```

```
fffff800`03cc7fea 83e720          and     edi,20h
fffff800`03cc7fed 25ff0f0000      and     eax,0FFFh

nt!KiSystemServiceRepeat:
fffff800`03cc7ff2 4c8d1547782300  lea     r10,[nt!KeServiceDescriptorTable
(fffff800`03eff840)]
fffff800`03cc7ff9 4c8d1d80782300  lea     r11,[nt!KeServiceDescriptorTableShadow
(fffff800`03eff880)]
fffff800`03cc8000 f7830001000080000000 test dword ptr [rbx+100h],80h
fffff800`03cc800a 4d0f45d3        cmovne  r10,r11
fffff800`03cc800e 423b441710      cmp     eax,dword ptr [rdi+r10+10h]
fffff800`03cc8013 0f83e9020000    jae     nt!KiSystemServiceExit+0x1a7 (fffff800`03cc8302)

nt!KiSystemServiceRepeat+0x27:
fffff800`03cc8019 4e8b1417        mov     r10,qword ptr [rdi+r10]
fffff800`03cc801d 4d631c82        movsxd  r11,dword ptr [r10+rax*4]
fffff800`03cc8021 498bc3          mov     rax,r11
fffff800`03cc8024 49c1fb04        sar     r11,4
fffff800`03cc8028 4d03d3          add     r10,r11
fffff800`03cc802b 83ff20          cmp     edi,20h
fffff800`03cc802e 7550            jne     nt!KiSystemServiceGdiTebAccess+0x49
(fffff800`03cc8080)

【省略大量无关代码】
```

最终，我们在 KiSystemServiceRepeat 里找到了 KeServiceDescriptorTable 的踪影。可能会有人问，为什么不直接反汇编 KiSystemServiceRepeat 呢？原因很简单，因为你找不到 KiSystemServiceRepeat 的地址。虽然 KiSystemCall64 和 KiSystemServiceRepeat 都没有由 ntoskrnl.exe 导出，但是我们能直接找到 KiSystemCall64 的地址。怎么找？直接读取指定的 msr 得出。很多人只听过通用寄存器和调试寄存器，其实还有很多其它的寄存器（你想想再古老的 586 CPU 的一级缓存都有 32KB 呢，而现在的 AMD64 CPU 的每个核心的一级缓存正好有 64KB）。Msr 的中文全称是就是“特别模块寄存器”（model specific register），它控制 CPU 的工作环境和标示 CPU 的工作状态等信息（例如倍频、最大 TDP、危险警报温度），它能够读取，也能够写入，但是无论读取还是写入，都只能在 Ring 0 下进行。我们通过读取 C0000082 寄存器，能够得到 KiSystemCall64 的地址，然后从 KiSystemCall64 的地址开始，往下搜索 0x500 字节左右（特征码是 4c8d15），就能得到 KeServiceDescriptorTable 的地址了。同理，我们换一下特征码（4c8d1d），就能获得 KeServiceDescriptorTableShadow 的地址了。

先用 WINDBG 证明一下（输入 rdmsr c0000082）：

```
Command - Local kernel - WinDbg:6.11.0001.404 AMD64
lkd> rdmsr C0000082
msr[c0000082] = fffff800`03cc7ec0
lkd> u fffff800`03cc7ec0
nt!KiSystemCall64:
fffff800`03cc7ec0 0f01f8          swapgs
fffff800`03cc7ec3 6548892425100000000 mov   qword ptr gs:[10h],rsp
fffff800`03cc7ecc 65488b2425a8010000 mov   rsp,qword ptr gs:[1A8h]
fffff800`03cc7ed5 6a2b            push    2Bh
fffff800`03cc7ed7 65ff342510000000 push    qword ptr gs:[10h]
fffff800`03cc7edf 4153            push    r11
fffff800`03cc7ee1 6a33            push    33h
fffff800`03cc7ee3 51              push    rcx
```

代码实现如下：

```
ULONGLONG MyGetKeServiceDescriptorTable64()
{
    PUCHAR StartSearchAddress = (PUCHAR)__readmsr(0xC0000082);
    PUCHAR EndSearchAddress = StartSearchAddress + 0x500;
    PUCHAR i = NULL;
    UCHAR b1=0,b2=0,b3=0;
    ULONG templong=0;
    ULONGLONG addr=0;
    for(i=StartSearchAddress;i<EndSearchAddress;i++)
    {
        if( MmIsAddressValid(i) && MmIsAddressValid(i+1) && MmIsAddressValid(i+2) )
        {
            b1=*i;
            b2=*(i+1);
            b3=*(i+2);
            if( b1==0x4c && b2==0x8d && b3==0x15 ) //4c8d15
            {
                memcpy(&templong,i+3,4);
                addr = (ULONGLONG)templong + (ULONGLONG)i + 7;
                return addr;
            }
        }
    }
    return 0;
}
```

计算地址的核心代码是 4c8d15 后面的那 4 个字节（正好算是一个 long）加上当前指令的起始地址再加上 7。为什么要加上 7 呢？因为[lea r10,XXXXXXXX]指令的长度是 7 个字节。另外我在外国的网站上看到了同样功能的另外一段代码，也贴出来给大家围观下：

```
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;
    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 ((LONGLONG)tbl_address +7) + *(LONG*)(tbl_address +3);
            }
        }
    }
    return 0;
}
```

接下来就是讲述 SSDT 函数地址了。在获得地址之前，需要知道 SSDT 函数的 INDEX。获得这个 INDEX 的方法很简单，直接在 RING3 读取 NTDLL 的内容即可。使用 WINDBG 的方法如下：随便创建一个进程，然后使用 WINDBG 附加，然后在命令栏里输入：

```
u ntdll!函数名
```

比如输入 u ntdll!NtOpenProcess，出现以下结果：

```
0:004> u ntdll!ntopenprocess
ntdll!ZwOpenProcess:
00000000`772b0110 4c8bd1          mov     r10,rcx
00000000`772b0113 b823000000      mov     eax,23h
00000000`772b0118 0f05            syscall
00000000`772b011a c3              ret
```

再输入 u ntdll!NtTerminateProcess，出现以下结果：

```
0:004> u ntdll!NtTerminateProcess
ntdll!ZwTerminateProcess:
00000000`772b0170 4c8bd1          mov     r10,rcx
00000000`772b0173 b829000000      mov     eax,29h
00000000`772b0178 0f05            syscall
00000000`772b017a c3              ret
```

可以看到两次反汇编的结果几乎完全相同，唯一不同的地方是第二句。XXh 就是此函数的 index。知道 INDEX 之后，就可以计算地址了。首先看看原版的反汇编代码是怎么实现的（计算方法就藏在 KiSystemServiceStart 里）。

```
lkd> uf KiSystemServiceStart
```

```
nt!KiSystemServiceStart:
fffff800`03cc7fde 4889a3d8010000  mov     qword ptr [rbx+1D8h],rsp
;Native API Index
fffff800`03cc7fe5 8bf8            mov     edi,eax
;操作 1
fffff800`03cc7fe7 c1ef07          shr     edi,7
;操作 2
fffff800`03cc7fea 83e720          and     edi,20h
;操作 3（和获得地址无关，和对比函数有效性有关）
fffff800`03cc7fed 25ff0f0000      and     eax,0FFFh
nt!KiSystemServiceRepeat:
;取得 SSDT 地址
fffff800`03cc3ff2 4c8d1547782300  lea     r10,[nt!KeServiceDescriptorTable
(fffff800`03efb840)]
;取得 SSSDT 地址
fffff800`03cc3ff9 4c8d1d80782300  lea     r11,[nt!KeServiceDescriptorTableShadow
(fffff800`03efb880)]
;判断调用的是 ssdt 函数还是 sssdt 函数
fffff800`03cc4000 f7830001000080000000 test dword ptr [rbx+100h],80h
;根据上面的判断把 ssdt 或 sssdt 的基址放入 r10
fffff800`03cc400a 4d0f45d3        cmovne  r10,r11
;判断函数是否有效
fffff800`03cc400e 423b441710      cmp     eax,dword ptr [rdi+r10+10h]
;条件跳转
fffff800`03cc4013 0f83e9020000    jae     nt!KiSystemServiceExit+0x1a7
(fffff800`03cc4302)
;计算步骤 1
fffff800`03cc4019 4e8b1417        mov     r10,qword ptr [rdi+r10]
;计算步骤 2
fffff800`03cc401d 4d631c82        movsxd  r11,dword ptr [r10+rax*4]
;计算步骤 3
fffff800`03cc4021 498bc3          mov     rax,r11
;计算步骤 4
fffff800`03cc4024 49c1fb04        sar     r11,4
;计算步骤 5
fffff800`03cc4028 4d03d3          add     r10,r11
;edi 和 0x20 对比（和计算函数地址无关）
fffff800`03cc402b 83ff20          cmp     edi,20h
;条件跳转
fffff800`03cc402e 7550            jne     nt!KiSystemServiceGdiTebAccess+0x49
(fffff800`03cc4080)
【省略大量无关代码】
;调用 Native API
fffff800`03cc4150 41ffd2          call    r10
```

提取出来的代码就是：

```
mov rax, rcx ;rcx=Native API 的 index
lea r10,[rdx] ;rdx=ssdt 基址
mov edi,eax
shr edi,7
and edi,20h
mov r10, qword ptr [r10+rdi]
movsxd r11,dword ptr [r10+rax]
mov rax,r11
sar r11,4
add r10,r11
mov rax,r10
ret
```

由于微软的 x64 编译器不能内联汇编，所以使用我只能使用 Shellcode 了：

```
typedef UINT64 (__fastcall *SCFN)(UINT64,UINT64);
SCFN scfn;
```

```
VOID Initxxxx()
{
    UCHAR strShellCode[36]="\x48\x8B\xC1\x4C\x8D\x12\x8B\xF8\xC1\xEF\x07\x83\xE7\x20\x4E\x8B\x14\x17\x4D\x63\x1C\x82\x49\x8B\xC3\x49\xC1\xFB\x04\x4D\x03\xD3\x49\x8B\xC2\xC3";
    scfn=ExAllocatePool(NonPagedPool,36);
    memcpy(scfn,strShellCode,36);
}
```

```
ULONGLONG GetSSDTFunctionAddress64(ULONGLONG NtApiIndex)
{
    ULONGLONG ret=0;
    ULONGLONG ssdt=GetKeServiceDescriptorTable64();
    if(scfn==NULL)
        Initxxxx();
    ret=scfn(NtApiIndex, ssdt);
    return ret;
}
```

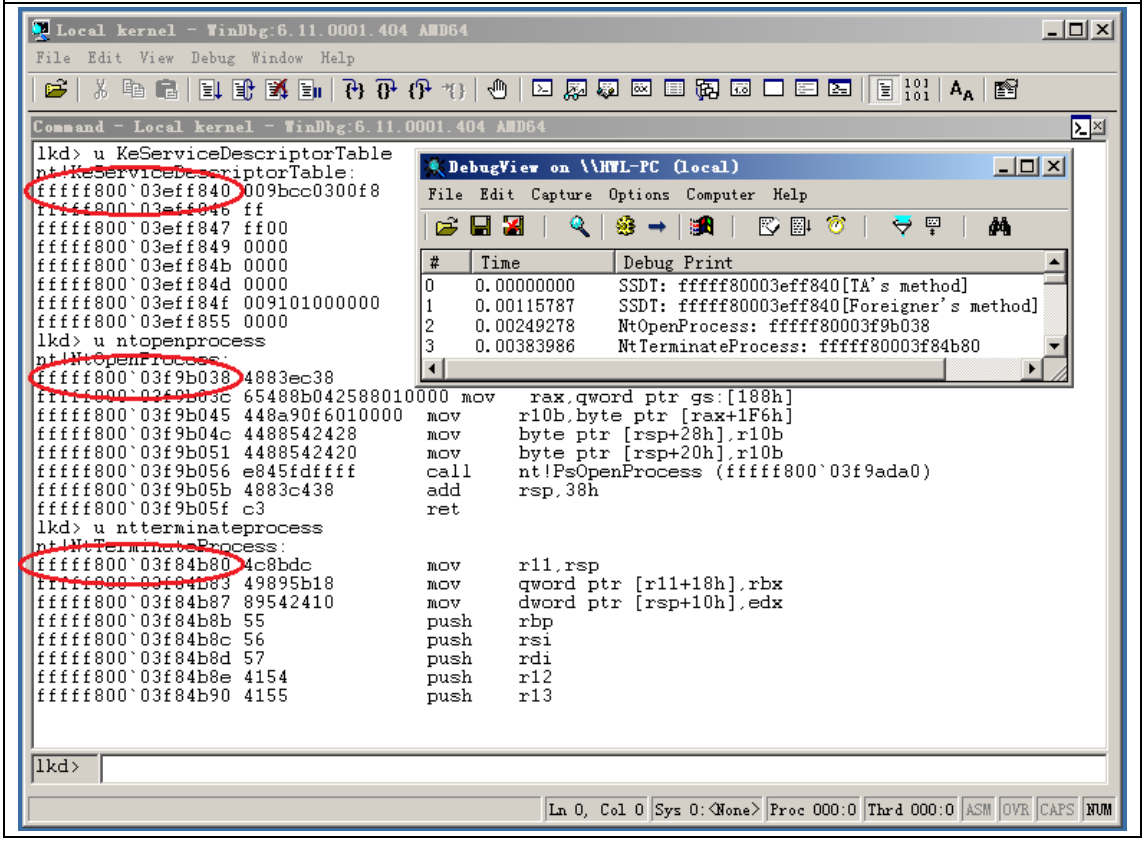
把汇编代码转换为 C 语言的代码如下（**注意以下结构体的第三个成员，通过它可以获得 SSDT 函数的数量**）：

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

```
ULONGLONG GetSSDTFunctionAddress64_2(ULONGLONG Index)
{
	LONG dwTemp=0;
	ULONGLONG qwTemp=0,stb=0,ret=0;
	PSYSTEM_SERVICE_TABLE ssdt=(PSYSTEM_SERVICE_TABLE)GetKeServiceDescriptorTable64();
	stb=(ULONGLONG)(ssdt->ServiceTableBase);
	qwTemp = stb + 4 * Index;
	dwTemp = *(PLONG)qwTemp;
	dwTemp = dwTemp >> 4;
	ret = stb + (LONG64)dwTemp;
	return ret;
}
```

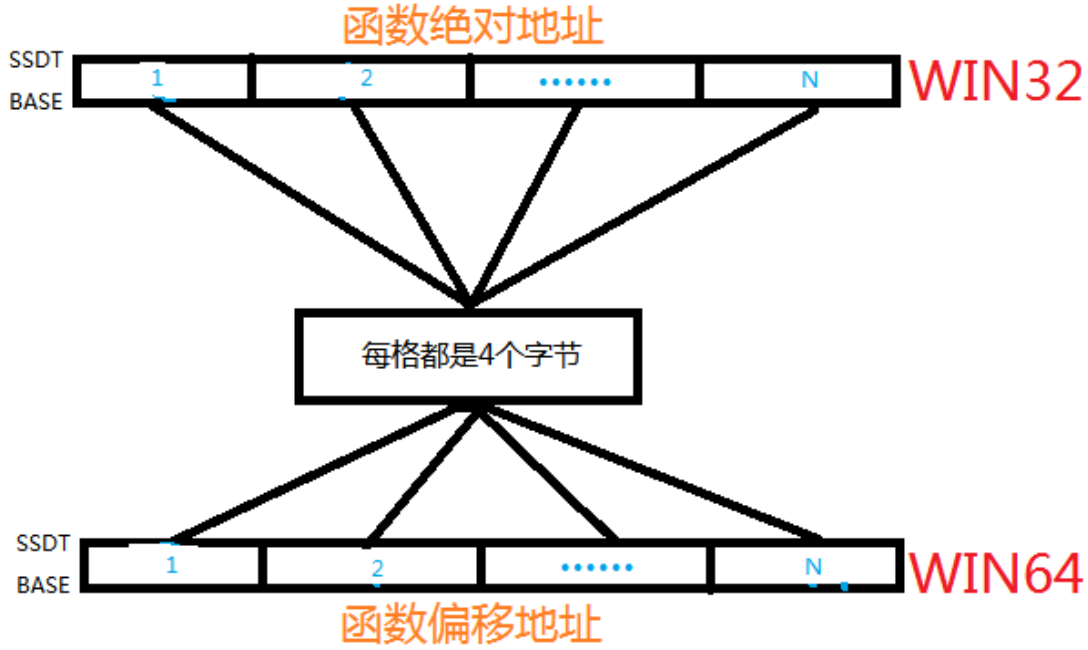
测试代码和运行结果：

```
	DbgPrint("[method 1]SSDT: %llx",MyGetKeServiceDescriptorTable64());
	DbgPrint("[method 2]SSDT: %llx",GetKeServiceDescriptorTable64());
	DbgPrint("[method 1]NtOpenProcess: %llx",GetSSDTFunctionAddress64(0x23));
	DbgPrint("[method 1]NtTerminateProcess: %llx",GetSSDTFunctionAddress64(0x29));
	DbgPrint("[method 2]NtOpenProcess: %llx",GetSSDTFunctionAddress64_2(0x23));
	DbgPrint("[method 2]NtTerminateProcess: %llx",GetSSDTFunctionAddress64_2(0x29));
```



最后，总结一下 WIN32 和 WIN64 在 SSDT 方面的不同。大家可以把 SSDT（其实 SHADOW SSDT 同理）想像成一排保险柜，每个柜子都有编号（从 0 开始），柜子的长度为四字节，每个柜子里都放了一个 LONG 数据。但不同的是，WIN32 的“柜子”里放的数据是某个函数

的绝对地址，而 WIN64 的“柜子”里放的数据是某个函数的偏移地址。这个偏移地址要经过一定的计算才能变成绝对地址。



课后作业：写一个枚举 SSDT 函数地址的程序，不需要函数名，把所有 SSDT 函数的地址打印出来即可。