标题： 秒杀所有的 SSDT HOOK 和 ShadowSSDT hook
作者： 兵糕
时间： 2011.12.9 1:40
地点： 某大学的某宿舍

有点纠结，不知道应不应该发，本来想多攒点东西，留着以后找工作。毕竟说空话被 bs 过。其实技术含量也不高，耐心研究下都能实现。发出来了，权当促进游戏保护和反保护事业的发展吧。大牛飘过吧，那些还没成为大牛就开始倚老卖老喜欢对刚入门的小菜指手画脚的，时不时的来一句“别瞎鼓捣了，你应该从这学起，你应该从那学起的吧“请嘴上饶人吧。别人实现过，我纯粹研究，不做破坏的事情。我的学习是单纯的兴趣使然。

首先是我研究的内容，ring0 下 过掉所有的 SSDT hook 和 Shadow SSDT hook（思路是从 360 的 hook 框架来的）。当然，对于过掉所有的 inline hook 也是我接下来要实现的 。具体实现也就是 pe 加载和重定位的问题，我有把握能实现。研究这个的目的是通杀所有的 SSDT hook 和 inline hook，很多游戏保护，很多木马都是通过给系统打补丁或者替换地址栏来执行恶意代码的目的。假如重新载入内核，替换各种内核中的表，那么可以就爽歪歪了。

接下来入正题，通杀所有的 SSDT hook 和 Shadow SSDT hook。
1，原理: hook 系统函数,Kifastcallentry，然后给这个函数打补丁，让它执行我们自己的代码。然后对代码中用到的内核表（也就是 KeServiceDescriptorTable 和 KeServiceDescriptorTableShadow 表）进行替换。从而使得木马，病毒，游戏保护们去处理原来的表去吧，内核已经不使用它们了。
2，实现：

A, 首先是找到两张系统表 KeServiceDescriptorTable 和 KeServiceDescriptorTableShadow 表。KeServiceDescriptorTable 是内核导出的，我们定义一下就 ok 了。

```
extern "C" ULONG KeServiceDescriptorTable;
```

对于后边一张表，由于内核并没有导出，那么我们可以有好几种方法得到，比如搜索 KeServiceDescriptorTable 周围的内存，比如 KeAddSystemServiceTable 函数中特征码搜索，比如线程对象 kthread 的 ServiceTable 域得到（这种方法最稳定）。在这里，我选择的是第二种方法 特征码搜索。

```
//获取keservicedescriptorTableShadow的地址 特征码搜索
UNICODE_STRING strFuncName;
ULONG  ulScan = 0;
RtlInitUnicodeString(&strFuncName,L"KeAddSystemServiceTable");
ulScan = (ULONG)MmGetSystemRoutineAddress(&strFuncName);

for(ULONG i=0;i<200;i++)
{
   if(*(PUSHORT)ulScan== 0x888d && *(PUCHAR)(ulScan+6)== 0x83)
      {
            pKeServiceDescriptorTableShadow = *(PULONG)(ulScan+2);
            KdPrint(("找到KeServiceDescriptorTableShadow:0x%x\n",pKeServiceDescriptorTableShadow));
            break;
      }
   ulScan++;
}
```

获取表的地址后，解析来我们来把这张表保存到我们申请的内存吧。

```
pSSDT = (PULONG)ExAllocatePool(NonPagedPool,0x500);

RtlZeroMemory(pSSDT,0x500);
KIRQL kIrql;
kIrql = KeRaiseIrqlToDpcLevel();
for(ULONG i=0;i<300;i++)
{
   pSSDT[i] = ulTable1[i];
}
KeLowerIrql(kIrql);
KdPrint(("pSSDT:0x%x\n",pSSDT));
//表拷贝复制完了
```

这是复制 SSDT 表，这里提高了系统的 irql 到 Dpclevel，防止线程切换。假如切换的话，我们复制的数据就会有问题了。同样的保存一下 ShadowSSDT 表

```
PEPROCESS pEprocess = NULL;
PKAPC_STATE pApcState = (PKAPC_STATE)ExAllocatePool(NonPagedPool,sizeof(KAPC_STATE));
PsLookupProcessByProcessId((HANDLE)604,&pEprocess);
if(MmIsAddressValid(pEprocess) != TRUE)
{
  KdPrint(("获取指定的进程失败了\n"));
      return;
}
KeStackAttachProcess((PKPROCESS)pEprocess,pApcState);

PULONG ulTable = (PULONG)*(PULONG)(pKeServiceDescriptorTableShadow+0x10);
ulSHADOWSSDTCount = (ULONG)*(PULONG)(pKeServiceDescriptorTableShadow+0x18);
for(ULONG i=0;i<700;i++)
{
  pSHADOWSSDT[i] = ulTable[i];
}

KeUnstackDetachProcess(pApcState);
ExFreePool(pApcState);
KdPrint(("pSHADOWSSDT:0x%x\n",pSHADOWSSDT));
```

这里要注意的就是 ShadowSSDT 表并不存在于系统进程空间的物理内存中，必须用 gui 线程来访问这张表。所以，我挂在到了 csrss.exe 进程。这里偷懒了，604 是测试环境下 csrss.exe 的进程 id。

B，接下来是 hook kifastcallentry(所有系统调用都必须经过这个内核函数)函数了。这个函数也不是系统内核导出的函数，所以，我们必须用别的方法获取到。方法有堆栈回溯法（很牛，很稳定的大法），rdmsr 的方法。360 采取的是堆栈回溯法，hook 了 ZwSetEvent，这里不展开了。因为程序仅仅是测试，所以，从简处理。运用 rdmsr 的方法获取 kifastcallentry 的地址。

```
ULONG ulKiFastCallEntry = 0;
__asm
{
   pushad
              pushfd
              mov ecx,0x176
              rdmsr
              mov ulKiFastCallEntry,eax
              popfd
              popad
}
```

接下里就是重点了。我没有用平常的 wtmsr 指令来替换 kifastcallentry 的地址，而是在这个函数内部实现 inline hook 跳转。所以，我要设法获取这个 hook 地址，我要 hook 的地址是

```
//804df7d0 8b3f            mov     edi,dword ptr [edi]
//804df7d2 8b1c87          mov     ebx,dword ptr [edi+eax*4]
```

为什么选这里呢？原因是 edi 指向的是表的地址。并且 eax 存放的是系统调用功能号。研究这个应该熟悉从 ring3 到 ring0 系统调用全过程的，这个我就不多说了。

```
ulScan = ulKiFastCallEntry;
for(ULONG i=0;i<300;i++)
{
   if(*(PULONG)ulScan == 0x1c8b3f8b && *(PUCHAR)(ulScan+4)== 0x87)
       {
               KdPrint(("找到了,ulHookAddr:0x%x\n",ulScan));
               break;
       }
   ulScan++;
}
```

特征码就是"0x83,0x3f,0x8b,0x1c,0x87"。这段特征码将被我替换为跳转的代码。

```
BYTE  bJmpCode[5] = {0xe9,0,0,0,0};
BYTE  bSource[5] = {0x8b,0x3f,0x8b,0x1c,0x87};
```

跳转我要多说几句。我们可以搭建跳板。怎么搭建呢，我们可以 ExallocatePool 分配跳转的中继地址，从 hook 出跳到中继地址，然后跳到最终地址 Filter 函数。跳转的偏移计算方法

```
ULONG ulOffset = (ULONG)Filter - ulHookAddr - 5;
*(PULONG)&bJmpCode[1] = ulOffset;
```

跳转偏移 = 目的地址 - 源地址 -5;

```
WPOFF();

RtlCopyMemory((PUCHAR)ulHookAddr,(PUCHAR)bJmpCode,5);

WPON();
```

然后，去除内核的代码段写保护，patch 地址，复位写保护。
这样之后，经过 Kifastcallentry 函数的线程都要到我们的跳转目的地址去执行了。

```
#pragma code_seg("PAGE")
__declspec(naked) VOID Filter()
{
       //这个函数比较重要  是替换的关键
       __asm
       {
          pushad
          pushfd

          mov edi,[edi]
          mov eax,KeServiceDescriptorTable
          mov eax,[eax]
          cmp edi,eax

          je  SSDTProxy    //先仅仅对ssdt hook 全部屏蔽
          jmp Pass

SSDTProxy:
          popfd
          popad
          mov edi,pSSDT
          mov ebx,[edi + eax*4]
          jmp [ulRet]
Pass:
          popfd
          popad
          mov edi,[edi]
          mov ebx,[edi + eax*4]
          jmp [ulRet]
       }
}
```

C,hook 跳转的目的地址是我自己定义的 filter 裸函数

现在代码仅仅替换 KeServiceDescriptorTable 这张表毕竟测试吗。对了，落了对这两张表的介绍。让我们来认识一下这两张表吧。

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

ShadowSSDT 表。可以看出，它包含 SSDT 表。接下里，SSDT 表

```
typedef struct _ServiceDescriptorEntry {
  unsigned int *ServiceTableBase;        //SSDT基址
  unsigned int *ServiceCounterTableBase; //SSDT中服务被调用次数的计数器
  unsigned int NumberOfServices;         //SSDT服务个数
  unsigned char *ParamTableBase;         //SSPT基址
}SSDT, *PSSDT;
```

需要讲一下的时 edi 指向的是 PSSDT->ServiceTableBase，或者是 shadow 表的 ServiceTableBase。

原理说完了。接下来让我们看看实验效果吧。

```
kd> dd KeServiceDescriptortable
8055b220  804e36b8 00000000 0000011c 805110c8
8055b230  00000000 00000000 00000000 00000000
8055b240  00000000 00000000 00000000 00000000
8055b250  00000000 00000000 00000000 00000000
8055b260  00002710 bf80c31c 00000000 00000000
8055b270  f8a45a80 f82bdb60 81c2e930 80700f40
8055b280  00000000 00000000 61b30fba 000004f4
8055b290  3c9b677a 01ccb5ae 00000000 00000000
kd> dd 804e36b8+0x7a*4
804e38a0  805729ac 8056f0cd 8056f2c6 805721b4
804e38b0  805a0042 8058acfe 8058f5c4 8056eb6a
804e38c0  8056eadb 8064a3f1 805dc394 8059da1e
804e38d0  805dea4e 805de2e8 805ab8cc 80572e96
804e38e0  805dc12c 80575692 8064a3d5 8064a3d5
804e38f0  804f8e5d 80567b82 8057fc87 805732f6
804e3900  8058558d 80617d6c 8058aeaf 8057d9fa
804e3910  805d8798 80573e4f 80581a8d 80624583
kd> ln 805729ac
(805729ac)   nt!NtOpenProcess   |  (80572b33)   nt!PsLookupProcessByPro
Exact matches:
    nt!NtOpenProcess = <no type information>
kd> ed 804e38a0 0
kd> g
Single step exception - code 80000004 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
PsSuspendThread!Filter+0xd:
f87fc01d 7402            je      PsSuspendThread!Filter+0x11 (f87fc021)
kd> g
```

如上图，我把系统本身的 SSDT 表中的 NtOpenProcess 修改为 0 了，然后运行系统。正常。我们来到 ark 工具中看看吧。

In...	Service Name	Current Address	Original Address	State	Module
162	NtQueryMutant	0x80651666	0x80651666	-	C:\WINDOWS\system32\ntoskrnl.exe
163	NtQueryObject	0x8058B270	0x8058B270	-	C:\WINDOWS\system32\ntoskrnl.exe

117	NtOpenIoCompletion	0x80617B1F	0x80617B1F	-	C:\WINDOWS\system32\ntoskrnl.exe
118	NtOpenJobObject	0x8063143B	0x8063143B	-	C:\WINDOWS\system32\ntoskrnl.exe
119	NtOpenKey	0x80569D48	0x80569D48	-	C:\WINDOWS\system32\ntoskrnl.exe
120	NtOpenMutant	0x805792C5	0x805792C5	-	C:\WINDOWS\system32\ntoskrnl.exe
121	NtOpenObjectAuditAlarm	0x80596401	0x80596401	-	C:\WINDOWS\system32\ntoskrnl.exe
122	NtOpenProcess	0x00000000	0x805729AC	Modified	-
123	NtOpenProcessToken	0x8056F0CD	0x8056F0CD	-	C:\WINDOWS\system32\ntoskrnl.exe
124	NtOpenProcessTokenEx	0x8056F2C6	0x8056F2C6	-	C:\WINDOWS\system32\ntoskrnl.exe
125	NtOpenSection	0x805721B4	0x805721B4	-	C:\WINDOWS\system32\ntoskrnl.exe
126	NtOpenSemaphore	0x805A0042	0x805A0042	-	C:\WINDOWS\system32\ntoskrnl.exe
127	NtOpenSymbolicLinkObject	0x8058ACFE	0x8058ACFE	-	C:\WINDOWS\system32\ntoskrnl.exe
128	NtOpenThread	0x8058F5C4	0x8058F5C4	-	C:\WINDOWS\system32\ntoskrnl.exe
129	NtOpenThreadToken	0x8056EB6A	0x8056EB6A	-	C:\WINDOWS\system32\ntoskrnl.exe

很闪亮的 NtOpenProcess 的当前地址变为零了。可是系统跑的很正常。　　哈哈。