Windows驱动编程之键盘过滤杂谈

一半人生 / 2019-10-23 09:20:39 / 浏览数 6554 安全技术 二进制安全 顶(0) 踩(0)

键盘接口

还记着以前的老式电脑，键盘鼠标音响全是拆卸，主机后面全是各种拔插的设备孔，当时的键盘鼠标通过PS/2接口进行设备连接，就是圆头插孔，绿色是鼠标紫色是键盘。

Personal
2系列是IBM在80年代推出的，而且兼容性非常好，是可以做到无冲突，意思就是说同时按下两个键，会被精准识别。而USB来说只能说是逻辑无冲突，最多6个键同时按下无
HUB，PS/2算是完败。
而现在随着发展无线键盘鼠标更是非常普及，利用蓝牙连接，有些特殊的还用P2P来做为连接（长线连接）。

系统处理键盘过程：

下述是一段汇编代码，因为涉及两次硬中断与轮询，下述只是个伪汇编，为了介绍一些内容而已，内联汇编如下所示：

```
static byte scandata;

// ■■■
__asm
{
   push eax

   // ■■■■■
   IN al, 0x64h
   and al, 00000010b // 0x2
   cmp al, 0   // ■■■■■■■■
   // ■■■■■■■■■jne or jnz■■■■■
   mov scandata, al

   pop eax
}
if(!(scandata & 2))
   printf("%x", scandata);

// ■■■■■■■64h■■■■■■■■,DOS■■■■JJ
__asm
{
   push eax

   mov al, scandata
   OUT 0x64, al

   pop eax
}
```

键盘控制器KBC，Intel
8042这个东西负责读取键盘扫描缓冲区数据，ECE1007负责连接键盘和EC，将键盘动作转换成扫描码。所以说两个IO端口进行通信的，分别是0x60与0x64，引用一段上古

```
#define I8042_COMMAND_REG       0x64
#define I8042_STATUS_REG        0x64
#define I8042_DATA_REG          0x60
```

```
■■8042■■■■■■■■■■64h,■■■■60h■■■■■■■■■■■■■■60h■■■■■■■■■■■■■■■■■■■2■■■■■■■■■■
■■■64h■■■■■■■■■■■■■■■■■■■■■■■■status■■■■■■■■■■0x64■■■■■■I8042_STATUS_REG.■■command■■■■■■■■■■I8042_COMMAND_REG■■■■■■
■■8048■■■■■■■■■■60h.■■■■■■Keyboard■■■■■■■60h■■■■■■■■■Scan Code■■■■■■■■■■■■■■■8048■■■■■■■■■■■■ACK■■■■■■■■Command■■■■
```

有兴趣的可以写一个真正的键盘端口读写过滤，我记着王爽老师汇编在最后几章代码描述键盘DOS下描述全面，同样寒江独了一书中也进行了直接端口读写章节介绍，都有

当按下键盘是会发送一个硬件外部中断，比如键盘中断、打印机中断、定时器中断等，然后内部会通过中断码去找对应的中断处理服务，如键盘管理中断服务等，如触发0x9

PS/2键盘端口是60h，IN AL,
60h从端口输入，端口获取的数据最高位进行逻辑与比较，当我们按下键盘触发中断，CPU会读取0x60的扫描码，0x60有一个字节，扫描码保存可以是两个字节，键盘弹起
= 通码 + 0x80，这里深层原理不在深究。

ps/2键盘扫描码表:

键	按下码	释放码	键	按下码	释放码	键	按下码	释放码
A	1E	9E	9	0A	8A	[	1A	9A
B	30	B0	`	29	89	INSERT	E0,52	E0,D2
C	2E	AE	-	0C	8C	HOME	E0,47	E0,97
D	20	A0	=	0D	8D	PG UP	E0,49	E0,C9
E	12	92	\	2B	AB	DELETE	E0,53	E0,D3
F	21	A1	BKSP	0E	8E	END	E0,4F	E0,CF
G	22	A2	SPACE	39	B9	PG DN	E0,51	E0,D1
H	23	A3	TAB	0F	8F	U ARROW	E0,48	E0,C8
I	17	97	CAPS	3A	BA	L ARROW	E0,4B	E0,CB
J	24	A4	L SHFT	2A	AA	D ARROW	E0,50	E0,D0

寒江独钓书中是这样表述的：PDO字面意思就是说物理设备，然后是设备栈最下面的设备对象，csrss.exe进行中RawInputThread线程通过GUIDClass来获取键盘设备栈中的
RawInputThread执行函数OpenDevice，通过结构体OBJECT_ATTRIBUTES找到设备栈的PDO符号链接，这个对象我们在windbg看一下，写过ObjectHOOK的对这些理解

```
kd> dt _OBJECT_ATTRIBUTES
nt!_OBJECT_ATTRIBUTES
  +0x000 Length           : Uint4B
  +0x004 RootDirectory    : Ptr32 Void
  +0x008 ObjectName       : Ptr32 _UNICODE_STRING   ■■■■
  +0x00c Attributes       : Uint4B
  +0x010 SecurityDescriptor : Ptr32 Void
  +0x014 SecurityQualityOfService : Ptr32 Void
```

然后调用ZwCreateFile打开设备，返回句柄操作。ZwCreateFile调用NtCreateFile --> IoParseDevice -->
IoGetAttachedDevice，然后就是得到了最顶端的设备对象，继续通过对象结构30 offset StackSize初始化irp。

ObCreateObject创建文件对象，offset 4
有一个DEVICE_OBJECT对象，这是一个比较有意思数据结构，可以通过_DRIVER_OBJECT对象找到一个驱动所全部的DEVICE_OBJECT，通过这个数据结构可以遍属于该驱

上述讲述的就是API层面或说windows如何通过进程来处理键盘响应的，其实你要做的与上述系统的处理试大差不差，也需要调用这些API来做。

```
kd> dt _DEVICE_OBJECT
nt!_DEVICE_OBJECT
  +0x000 Type             : Int2B
  +0x002 Size             : Uint2B
  +0x004 ReferenceCount   : Int4B
  +0x008 DriverObject     : Ptr32 _DRIVER_OBJECT ■■■■
  +0x00c NextDevice       : Ptr32 _DEVICE_OBJECT ■■■■■■■■■■
  +0x010 AttachedDevice   : Ptr32 _DEVICE_OBJECT
  +0x014 CurrentIrp       : Ptr32 _IRP
  +0x018 Timer            : Ptr32 _IO_TIMER
  +0x01c Flags            : Uint4B
  +0x020 Characteristics  : Uint4B
```

```
+0x024 Vpb              : Ptr32 _VPB
+0x028 DeviceExtension  : Ptr32 Void
+0x02c DeviceType       : Uint4B
+0x030 StackSize        : Char
+0x034 Queue            : <unnamed-tag>
+0x05c AlignmentRequirement : Uint4B
+0x060 DeviceQueue      : _KDEVICE_QUEUE
+0x074 Dpc              : _KDPC
+0x094 ActiveThreadCount : Uint4B
+0x098 SecurityDescriptor : Ptr32 Void
+0x09c DeviceLock       : _KEVENT
+0x0ac SectorSize       : Uint2B
+0x0ae Spare1           : Uint2B
+0x0b0 DeviceObjectExtension : Ptr32 _DEVOBJ_EXTENSION
+0x0b4 Reserved         : Ptr32 Void
```

然后就是按下键盘，通过一系列的中断就是我们上述说的那个，最后从端口读取扫描码在经过一些列处理数据给IRP，结束IRP。RawInputThread线程读操作后，会得到数据
设备栈情况：

最顶层：Kbdclass

中间层：i8042ptr

最底层：ACPI

在双机调试关机时候调试信息输出: Wait PDO address = xxxxx...数据，一直卡死等待，这时候你就要考虑是不是驱动绑定及解除出现了一些问题。

键盘数据过滤：

过滤串口时候，我们只用的设备名来作为绑定，返回的设备栈的顶层指针，那么如何找到所有的键盘设备呢？

1. 绑定最顶层的设备栈Kbdclass ，先获取Object:

```
// KBD_DRIVER_NAME = L"\\Driver\\Kbdclass"
RtlInitUnicodeString(&uniNtNameString, KBD_DRIVER_NAME);

// 通过设备对象获取底层对象指针
status = ObReferenceObjectByName(
    &uniNtNameString,
    OBJ_CASE_INSENSITIVE,
    NULL,
    0,
    IoDriverObjectType,
    KernelMode,
    NULL,
    &kbdDriverObject
);
```

然后进行遍历打开、绑定保存:

```
// 循环遍历所有的设备对象
while (pTargetDeviceObject)
{
    // 打开设备对象
    status = IoCreateDevice(
        IN DriverObject,
        IN sizeof(C2P_DEV_EXT),
        IN NULL,
        IN pTargetDeviceObject->DeviceType,
        IN pTargetDeviceObject->Characteristics,
        IN FALSE,
        OUT &pFilterDeviceObject);

    if (!NT_SUCCESS(status))
    {
        IoDeleteDevice(pFilterDeviceObject);
        pFilterDeviceObject = NULL;
        return status;
    }

    // 绑定设备对象
    status = IoAttachDeviceToDeviceStackSafe(pFilterDeviceObject, pTargetDeviceObject, &pLowerDeviceObject);
    if (!NT_SUCCESS(status))
    {
        IoDeleteDevice(pFilterDeviceObject);
        pFilterDeviceObject = NULL;
        return status;
    }
```


3. 这个函数功能仅仅是绑定，而并非通过绑定函数触发过滤机制，通过READ去读的，触发的是派遣函数IRP_MJ_READ。

```
// Read很重要
DriverObject->MajorFunction[IRP_MJ_READ] = c2pDispatchRead;

// IRP_MJ_POWER函数电源处理
DriverObject->MajorFunction[IRP_MJ_POWER] = c2pPower;

// 即插即用拔插函数
DriverObject->MajorFunction[IRP_MJ_PNP] = c2pPnP;
```


4. 调用IoSetCompletionRoutine函数，其实就是注册了IoCompletion例程，第二个参数就是我们处理Irp的函数：

```
void IoSetCompletionRoutine(
PIRP                    Irp,
PIO_COMPLETION_ROUTINE CompletionRoutine,
__drv_aliasesMem PVOID Context,
BOOLEAN                 InvokeOnSuccess,
BOOLEAN                 InvokeOnError,
BOOLEAN                 InvokeOnCancel
);
```

2.

```
// 设置IRP完成回调函数
DbgBreakPoint();
IoSetCompletionRoutine(Irp, c2pReadComplete, DeviceObjet, TRUE, TRUE, TRUE);
status = IoCallDriver(pLowDevice, Irp);
```


而c2pReadComplete函数主要截获了Irp保存在IRP栈中的扫描码，进行了替换（过滤），从而让通码成为我们指定的数据，达到效果:

```
IrpSp = IoGetCurrentIrpStackLocation(Irp);

DbgBreakPoint();

if (NT_SUCCESS(Irp->IoStatus.Status))
{
    gC2pKeyCount++;
    PKEYBOARD_INPUT_DATA KeyData;
    KeyData = Irp->AssociatedIrp.SystemBuffer;
    buf_len = Irp->IoStatus.Information / sizeof(KEYBOARD_INPUT_DATA);
    // 从KEYBOARD_INPUT_DATA中得到键盘

    for (i = 0; i < buf_len; ++i)
    {
        KdPrint(("扫描码:%x\n", KeyData->MakeCode;))
        KdPrint(("%s\n", KeyData->Flags ? "up" : "Down";))

        // 通码替换
        if (KeyData->MakeCode == 0x1f)
        {
            KeyData->MakeCode = 0x20;
        }
    }
}
```


动态卸载函数也很有意思，书中做稳妥的处理方式，如下所示：

```
// IRP请求设置1后秒转换成64位整数
lDelay = RtlConvertLongToLargeInteger(-10 * ulMicroSecond);
CurrentThread = KeGetCurrentThread();

// 低实模式
KeSetPriorityThread(CurrentThread, LOW_REALTIME_PRIORITY);

DeviceObjet = DriverObject->DeviceObject;
while (DeviceObjet)
{
    c2pDetach(DeviceObjet);
    DeviceObjet = DeviceObjet->NextDevice;
}
ASSERT(NULL == DriverObject->DeviceObject);

// 为了防止请求没有完成
while (gC2pKeyCount)
{
    // 内核睡眠，也就是延迟完成IRP请求1秒
    KeDelayExecutionThread(KernelMode, FALSE, &lDelay);
}
return;
```


设置全局标识，标识是否有请求处理为完成，如果有请求为处理完成，一直循环处理，这个很重要。如果你卸载了过滤设备，IRP请求还在处理状状态，ZwCreate仍即读

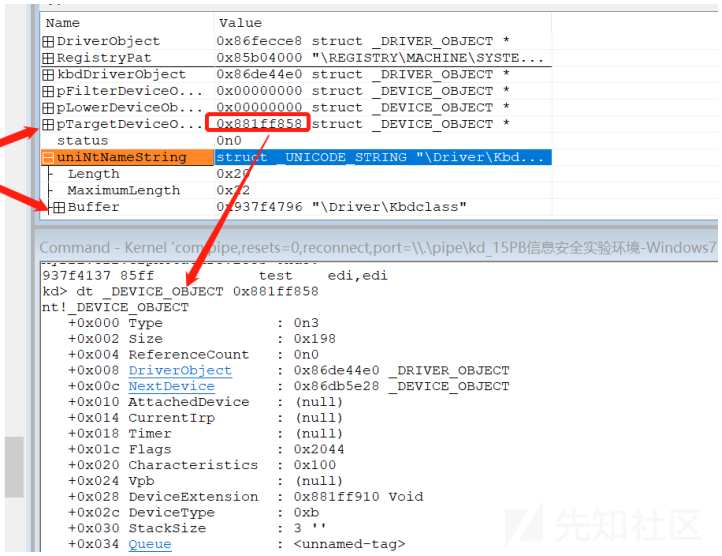
上述代码风格与书保持一致，因为去年写键盘驱动过滤发笔记，因为代码风格不同，很多人阅读代码去参考书籍理解时候带来了许多困难。

Windbg动态调试:

为了更清楚了解释上述原理与代码，动态调试看代码运行流程：

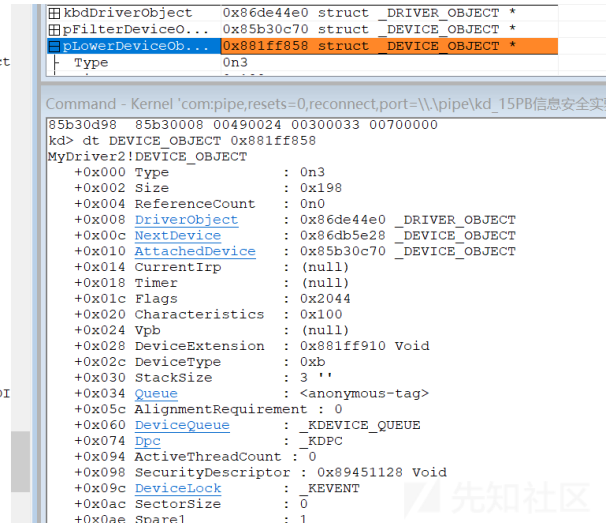
1. 打开、绑定PDO：

```
        // 通过设备对象获取底层对象指针
        status = ObReferenceObjectByName(
                &uniNtNameString,
                OBJ_CASE_INSENSITIVE,
                NULL,
                0,
                *IoDriverObjectType,
                KernelMode,
                NULL,
                &kbdDriverObject
        );

        if (!NT_SUCCESS(status))
        {
                return status;
        }

        // 引用计数减一
        ObDereferenceObject(DriverObject);

        // 将PDO设备对象赋值给pTar
        pTargetDeviceObject = kbdDriverObject->DeviceObject;

        // 循环遍历所有的设备对象
        while (pTargetDeviceObject)
        {
                // 打开设备对象
                status = IoCreateDevice(
                        IN DriverObject,
                        IN sizeof(C2P_DEV_EXT),
                        IN NULL,
                        IN pTargetDeviceObject->DeviceType,
                        IN pTargetDeviceObject->Characteristics,
                        IN FALSE,
                        OUT &pFilterDeviceObject);
```



我们先打开了顶层设备栈对象Kbdclass，然后DEVICE_OBJECT中获取对象，然后打开设备对象，上述DEVICE_OBJECT则是Kbdclass的设备对象，Type是3代表这是设备对象，下面就是绑定及生成过滤设备，如下：

```
                }

                // 绑定设备对象
                status = IoAttachDeviceToDeviceStackSafe(pFilterDeviceObject, pTargetDeviceObject
                if (!NT_SUCCESS(status))
                {
                        IoDeleteDevice(pFilterDeviceObject);
                        pFilterDeviceObject = NULL;
                        return status;
                }

                devExt = (PC2P_DEV_EXT)(pFilterDeviceObject->DeviceExtension);

                // 填充数据接结构
                c2pDevExtInit(
                        devExt,
                        pFilterDeviceObject,
                        pTargetDeviceObject,
                        pLowerDeviceObject
                );

                pFilterDeviceObject->DeviceType = pLowerDeviceObject->DeviceType;
                pFilterDeviceObject->Characteristics = pLowerDeviceObject->Characteristics;
                pFilterDeviceObject->StackSize = pLowerDeviceObject->StackSize + 1;
                pFilterDeviceObject->Flags |= pLowerDeviceObject->Flags & (DO_BUFFERED_IO | DO_DI
                pTargetDeviceObject = pTargetDeviceObject->NextDevice;
        }
        return status;
}

NTSTATUS DriverEntry(
        IN PDRIVER_OBJECT DriverObject,
        IN PUNICODE_STRING RegistryPath)
```



2. 键盘响应：

运行驱动，敲下键盘，这时候会在派遣的回调函数READ下发函数中断：

```
NTSTATUS c2pDispatchRead(
        IN PDEVICE_OBJECT DeviceObjet,
        IN PIRP Irp
)
{
        NTSTATUS status = STATUS_SUCCESS;
        PC2P_DEV_EXT devExt;
        PIO_STACK_LOCATION currentIrpStack;
        PDEVICE_OBJECT pLowDevice;
        KEVENT waitEvent;
        KeInitializeEvent(&waitEvent, NotificationEvent, FALSE);

        if (Irp->CurrentLocation == 1)
        {
                ULONG ReturnedInformation = 0;
                status = STATUS_INVALID_DEVICE_REQUEST;
                Irp->IoStatus.Status = status;
                Irp->IoStatus.Information = ReturnedInformation;
                IoCompleteRequest(Irp, IO_NO_INCREMENT);
                return status;
        }
        // 得到设备扩展。目的是之后为了获得下一个设备的指针。
        devExt = DeviceObjet->DeviceExtension;
        pLowDevice = devExt->LowerDeviceObject;
        currentIrpStack = IoGetCurrentIrpStackLocation(Irp);
        // 复制 IRP栈
        IoCopyCurrentIrpStackLocationToNext(Irp);
        // 设置 IRP完成回调函数
        DbgBreakPoint();
        IoSetCompletionRoutine(Irp, c2pReadComplete, DeviceObjet, TRUE, TRUE, TRUE)
        status = IoCallDriver(pLowDevice, Irp);
        return status;
}
```

通过设置了回调函数，也就是例程起始地址，下面就是捕获IRP栈中的数据，看到键盘MakeCode= 0x1e如下所示：

windbg g运行，结束这个函数IRP，发现立刻会在下发Read函数中断下来，这也就是说，一旦完成后会立刻调用ZwReadFile向驱动要求读入数据.

```
NTSTATUS c2pDispatchRead(
        IN PDEVICE_OBJECT DeviceObjet,
        IN PIRP Irp
)
{
        NTSTATUS status = STATUS_SUCCESS;
        PC2P_DEV_EXT devExt;
        PIO_STACK_LOCATION currentIrpStack;
        PDEVICE_OBJECT pLowDevice;
        KEVENT waitEvent;
        KeInitializeEvent(&waitEvent, NotificationEvent, FALSE);

        if (Irp->CurrentLocation == 1)
        {
                ULONG ReturnedInformation = 0;
```


HOOK手段：

替换分发函数指针：

键盘HOOK这种方式，有很多帖子叫FSD键盘钩子？个人认为FSD

HOOK应该是指FileSystemHOOK，也就是设备\FileSystem\Ntfs，后续文章中会说到。HOOK派遣函数指针其实本质是替换，与上述那种键盘过滤都是针对派遣函数调

定义全局变量先保存，这里只HOOK IRP_MJ_READ

```
PDRIVER_DISPATCH *OldReadAddress = NULL;
```

绑定过滤设备之后，也就是调用ObReferenceObjectByName之后，进行派遣函数保存:

```
OldReadAddress = KbdDriverObj->MajorFunction[IRP_MJ_READ];
```

然后派遣设置成自己的MyHook()

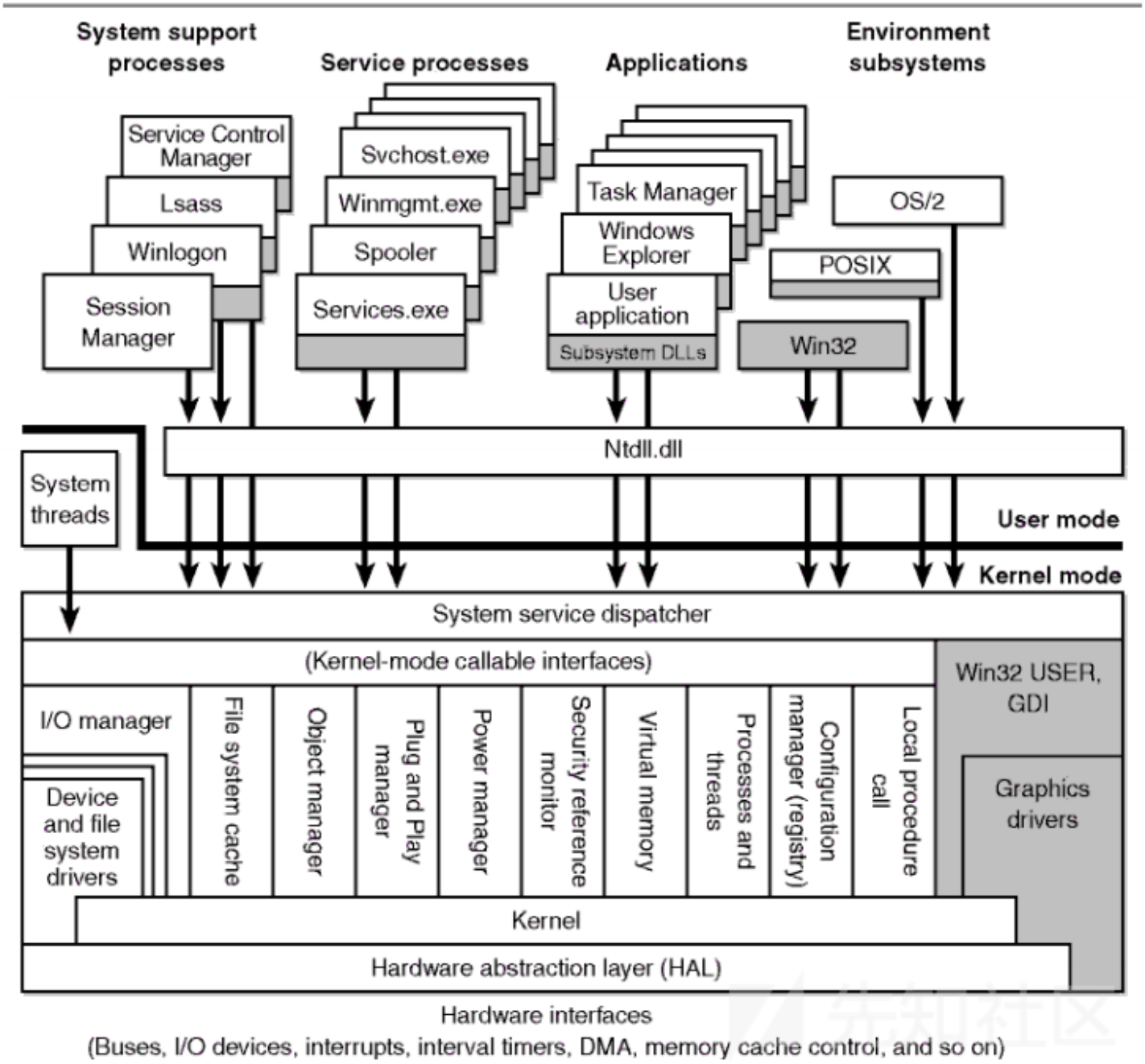
```
KbdDriverObj->MajorFunction[IRP_MJ_READ] = MyHook();
```

卸载驱动时候UnDriver时候还原指针:

```
kbdDriver->MajorFunction[IRP_MJ_READ] = OldReadAddress;
```

6. ##### 类驱动下端口指针HOOK：

   内核曾又分为：执行体层、微内核层、还有HAL层，打个比方EPROCESS属于执行体层，而内嵌的KPROCESS属于微内核层。那么EPROCESS信息包含句柄表、虚拟内存



端口驱动是根硬件打交道，一般都在HAL层，PS/2键盘端口驱动是i8042prt，USB是Kbdhid，键盘驱动工作就是接收中断请求、端口读写扫描码数据，数据传输给IRP完成

对于i8024ptr来说缓冲区来说，按下按键产生通码MakeCode，按键弹起BreakCode断码，都会有中断调用键盘中断服务例程，调用这些端口驱动。i8042ptr会调用I8042K

谭文老师书中的这块就是对层KeyboardInterruptService做HOOK，总的来说谁HOOK越底层谁就能把谁反了，你应用层HOOK我内核层反你，微内核HOOK我HAL在做手

这个KeyboardInterruptService地址没有公开，这里就按照书中方式动态调试的找一找这个函数地址，这里本想贴代码动态调试，复现二次没成功，代码被重构乱了，第一次

反过滤手段：

基础知识铺垫：

对于win可执行来说，有很多反调试手段，如检测窗口是否有OD、x64等窗口，获取PEB的数据，利用winApi检测等，而反HOOK显示要检验，比较常见的都是更早获取数据
对于键盘反过滤来说经典的就是中断HOOK，软中断有除零（0号中断）、断点（3号中断）、系统调用（2e号中断）以及异常处理等，当发生异常时候，系统就会通过中断
（Interrupt Descriptor Table），而硬中断被称为IRQ，这里不做细说。那么int
0x93，根据中断码去IDT找对应的中断处理函数，我们只需要HOOK处理IDT处理int 0x93中断的函数地址即可。
先来看看IDA表，windbg下用!pcr指令，就是查看当前KPCR结构，处理器控制域信息，这里不做多扩展，我们就可以发现IDT的基址，同样r idtr也可以读取：

```
kd> !pcr
KPCR for Processor 0 at 84131c00:
    Major 1 Minor 1
       NtTib.ExceptionList: 8412e32c
           NtTib.StackBase: 00000000
          NtTib.StackLimit: 00000000
        NtTib.SubSystemTib: 801e5000
             NtTib.Version: 0003703a
         NtTib.UserPointer: 00000001
             NtTib.SelfTib: 00000000

                   SelfPcr: 84131c00
                      Prcb: 84131d20
                      Irql: 0000001f
                       IRR: 00000000
                       IDR: ffffffff
             InterruptMode: 00000000
                       IDT: 80b95400
                       GDT: 80b95000
                       TSS: 801e5000

             CurrentThread: 8413b280
                NextThread: 00000000
                IdleThread: 8413b280

                 DpcQueue:
```

查看一下0x80b95400内存中的数据：

```
kd> dd 80b95400
80b95400  0008c200 84048e00 0008c390 84048e00
80b95410  00580000 00008500 0008c800 8404ee00
80b95420  0008c988 8404ee00 0008cae8 84048e00
80b95430  0008cc5c 84048e00 0008d258 84048e00
80b95440  00500000 00008500 0008d6b8 84048e00
80b95450  0008d7dc 84048e00 0008d91c 84048e00
80b95460  0008db7c 84048e00 0008de6c 84048e00
80b95470  0008e51c 84048e00 0008e8d0 84048e00
```

IDT表中每一项都是一个门描述符，包含了任务门、中断门、陷阱门这些，而我们键盘int 0x93HOOK就是中断例程入口，IDT记录了0~255的中断号和调用函数之间的关系。

63 48	47	46 45	44	43 40	39 32
	P	DPL	S	type	
Offset 16 : 31	access				必须为0
31 16	15 0				
Selector 15 : 0	Offset 15 : 0				

```
typedef struct _IDTENTRY
{
   unsigned short LowOffset;
   unsigned short selector;
   unsigned char retention : 5;
```

```
   unsigned char zero1 : 3;
   unsigned char gate_type : 1;
   unsigned char zero2 : 1;
   unsigned char interrupt_gate_size : 1;
   unsigned char zero3 : 1;
   unsigned char zero4 : 1;
   unsigned char DPL : 2;
   unsigned char P : 1;
   unsigned short HiOffset;
} IDTENTRY, *PIDTENTRY;
```

如何用汇编获取IDTR呢？汇编指令sidt

SIDT - 存储中断描述符表格寄存器

请参阅项目：SGDT/SIDT - 存储全局/中断描述符表格寄存器。

IDT HOOK（过PCHunter）：

本文不用修改IDT中断处理表中的例程函数来做键盘HOOK，而介绍另一种IDT
HOOK的方式，我们上述提到了GDT/LDT，这两个叫做全局描述符表/局部描述符表，GDT表中每项都是一个段描述符，因为索引号只有13bit，所以GDT数组最多有8192个
如何运作的呢，如下图所示，通过段选择子Segment Selector的TI标志位，如果是0意味着是GDT，如果是1意味着LDT表，GDTR Registe读取表基地址：

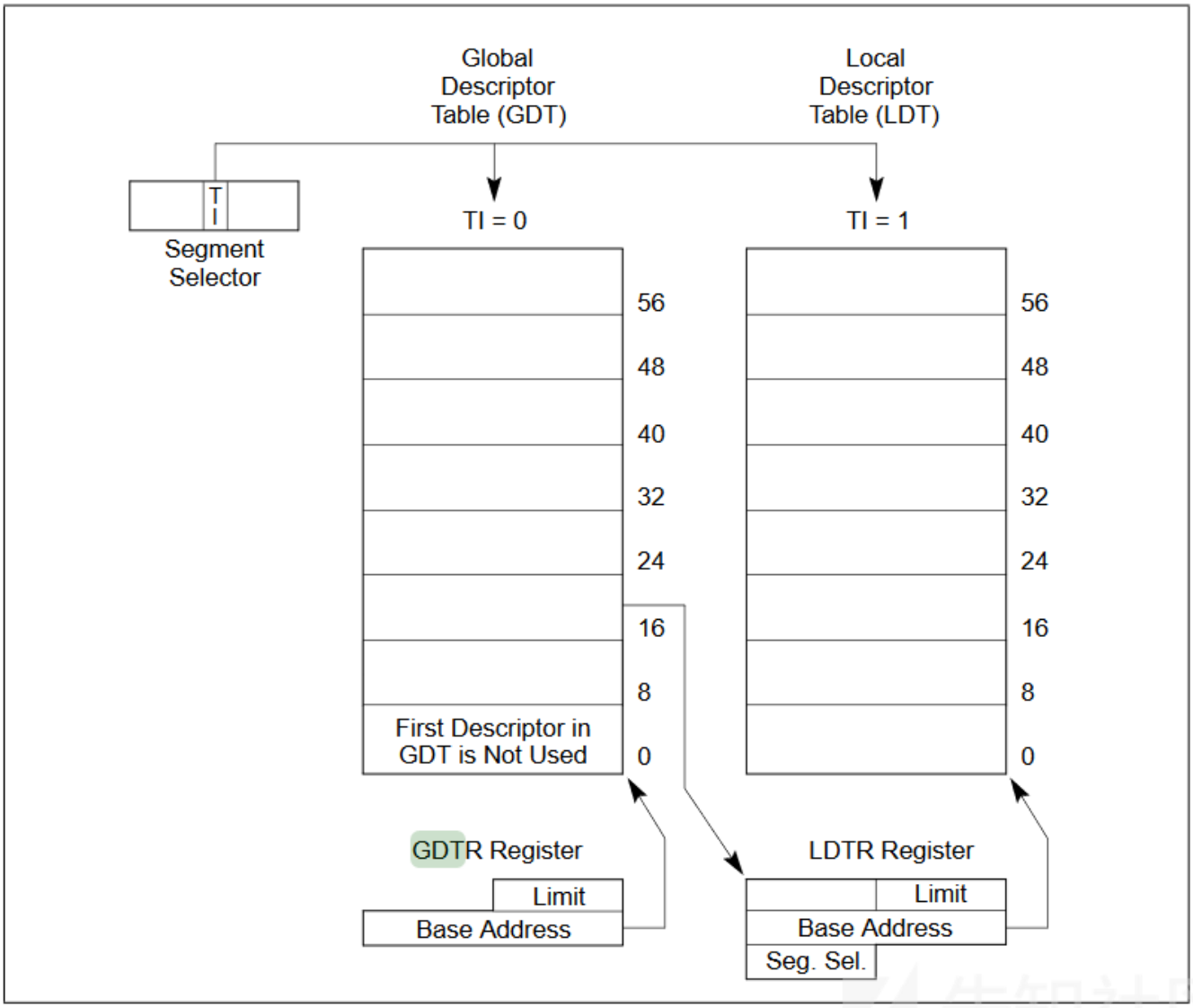


Figure 3-10. Global and Local Descriptor Tables

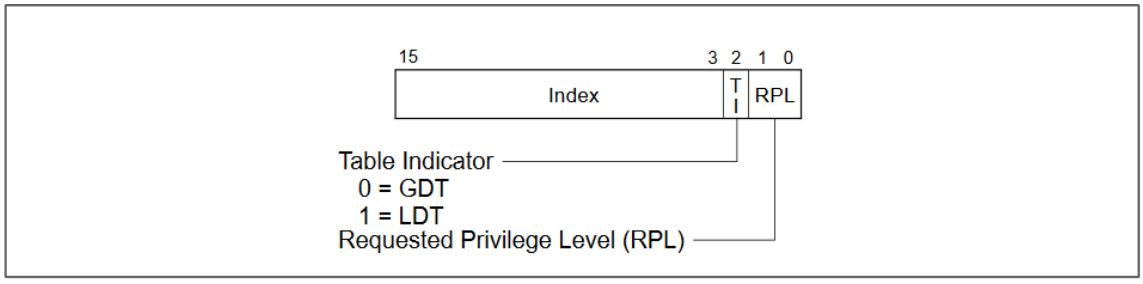


Figure 3-6. Segment Selector

因为段描述符又分为系统段、代码段、数据段，根据标志位，下述贴出一个标准IA-32e下的Descriptor:

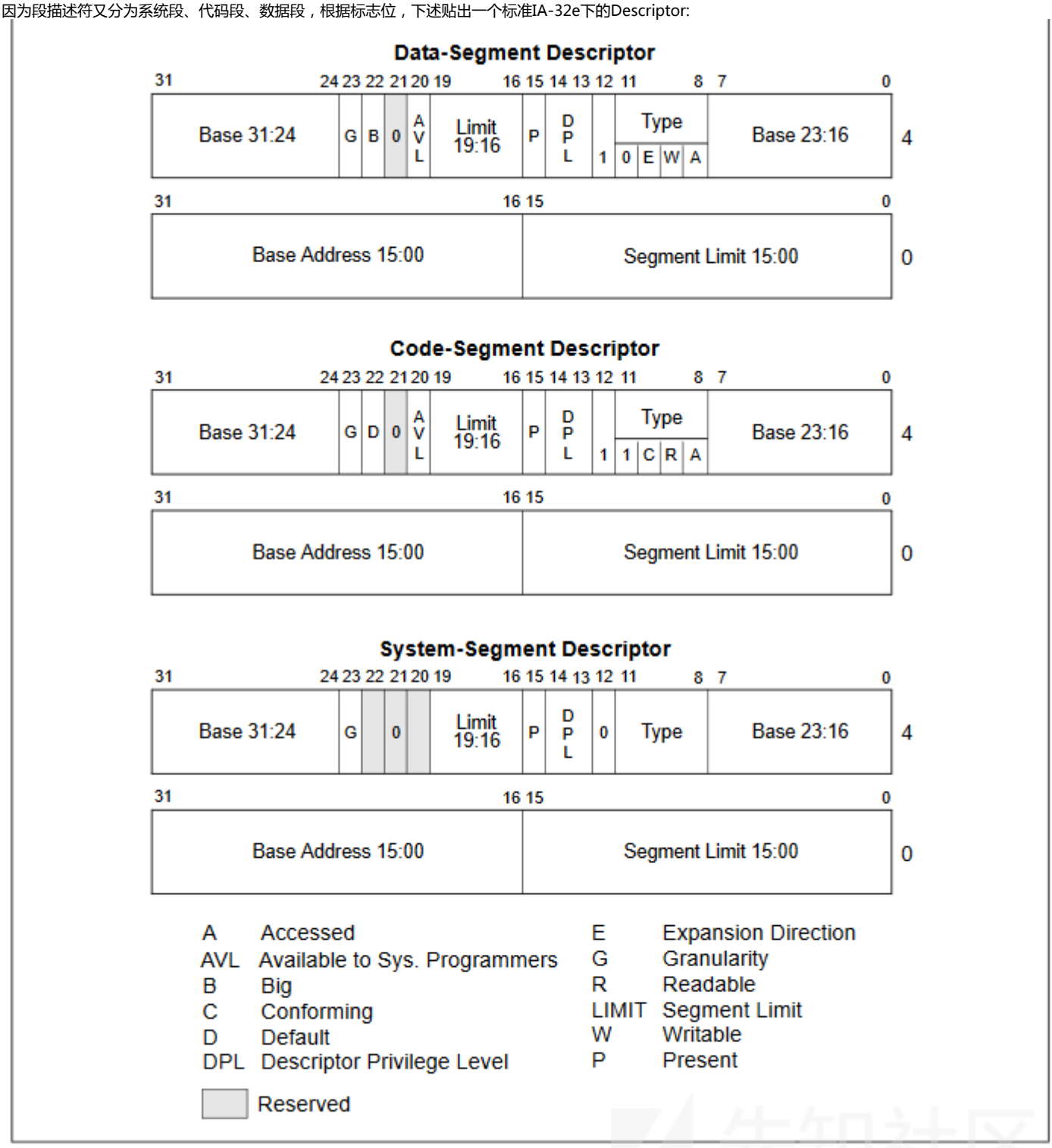
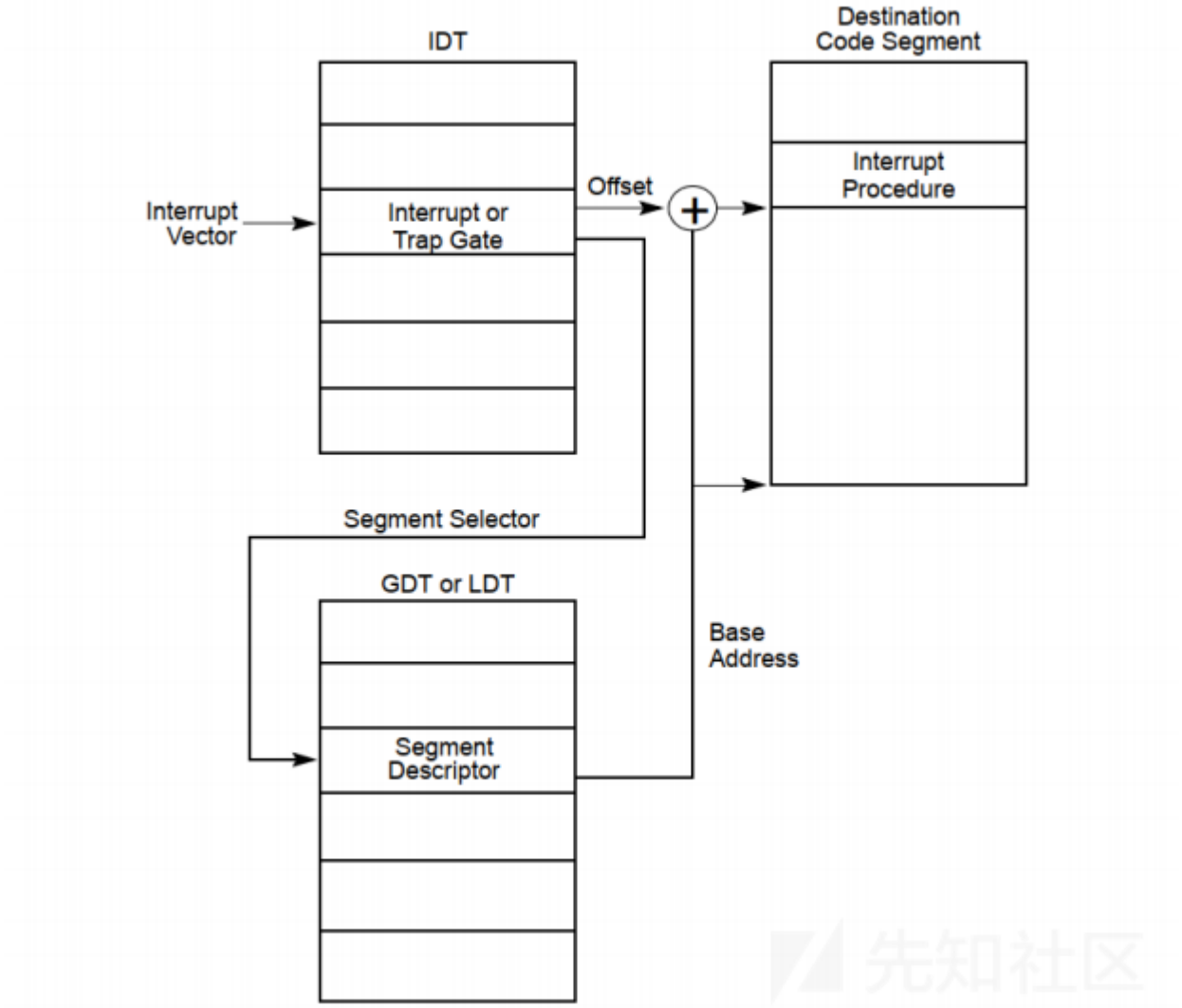


Figure 5-1. Descriptor Fields Used for Protection

有了上述知识的铺垫，来说一说键盘IDT HOOK如何实现，先明确思路，对于IDT HOOK来说，中断描述符修改符号表中索引地址就可以了，因为端口与处理中断是一一对应。而针对GDT来说我们不可以直接修改段描述符中的基地址，也就是Base Address直接修改，因为GDT会被其它的操作调用，贸然更改则会蓝屏崩溃。

产生中断或异常后：

1. CPU[redacted] IDT [redacted] -- [redacted]HOOK
2. [redacted]. -- [redacted]HOOK
3. [redacted] GDT [redacted] -- [redacted]HOOK
4. [redacted] + [redacted].
5. [redacted]



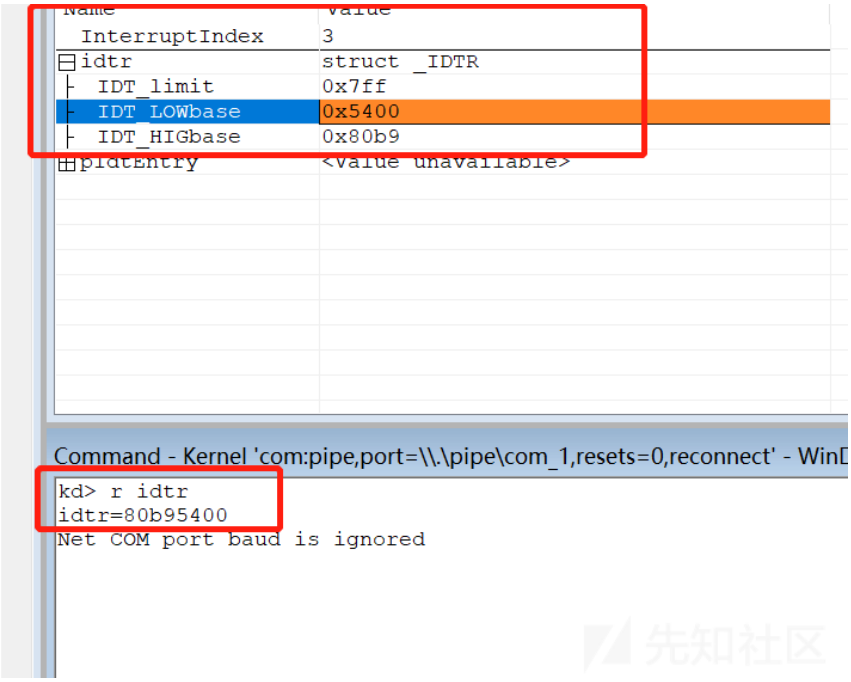

```
kd> r gdtr
gdtr=80b95000
kd> dq gdtr
80b95000  00000000`00000000 00cf9b00`0000ffff
80b95010  00cf9300`0000ffff 00cffb00`0000ffff
80b95020  00cff300`0000ffff 80008b1e`500020ab
80b95030  84409316`6c003748 0040f300`00000fff
80b95040  0000f200`0400ffff 00000000`00000000
80b95050  84008916`40000068 84008916`40680068
80b95060  00000000`00000000 00000000`00000000
80b95070  800092b9`500003ff 00000000`00000000
........................................
```

（1）首先我们还是要获取idt[0x93]，也就是键盘中断处理例程函数地址，如下所示，IDTR的寄存器48bit，其中32bit是基址，后16bit是IDT长度，我们定义下述结构体：

```
typedef struct _IDTR {
   USHORT   IDT_limit;
   USHORT   IDT_LOWbase;
   USHORT   IDT_HIGbase;
}IDTR, *PIDTR;

ULONG GetkeyIdtAddress()
{
   IDTR        idtr;
```

```
    IDTENTRY    *pIdtr;

    __asm    SIDT    idtr;

    /*
        MAKELONG
        idtr.IDT_LOWbase;  // ■■■    IDT_LOWbase | IDT_HIGbase << 16
        idtr.IDT_HIGbase;  // << 16bit
        minwindef.h■■■■■
    */
    pIdtr = (IDTENTRY *)MAKELONG(idtr.IDT_LOWbase, idtr.IDT_HIGbase);

    // ■■0x93■■■■■■■■■■■■
    return MAKELONG(pIdtr[0x93].LowOffset, pIdtr[0x93].HiOffset);
}
```

动态结果如下：




```
ndex)

tIndex);
```

```
与操作    IDT_LOWbase | IDT_HIGbase << 16
<< 16bit

itr.IDT_LOWbase, idtr.IDT_HIGbase);
```

注意的地方，IDT 表有时候没有通过IDTR来读取，多核CPU来说可能有多个IDT表，汇编指令idtr只能读取其中一个.

(2) 计算函数偏移，获取到了IDT中键盘处理中断的函数地址，用新得减去原地址，就可以得到偏移，韦伪代码如下：

```
// ■■■■■
VOID __declspec(naked) FilterFunction();
// ■■IDT■■■■■■■■■■
g_OldDescriptAddressBase = GetkeyIdtAddress(Index);
// ■■■■ + g_uOrigInterruptFunc ■ NewInterruptFunc
OffsetBase = NewInterruptFunc - g_OldDescriptAddressBase;
// ■■■■■■■■■■■■
*(ULONG*)g_Jmp = (ULONG)FilterFunction;
```

(3) 关于CR0~CR4，这里不多过介绍，写保护开启与关闭如下所示:

Following power-up, The state of control register CR0 is 60000010H (see Figure 9-1). This places the processor is in real-address mode with paging disabled.

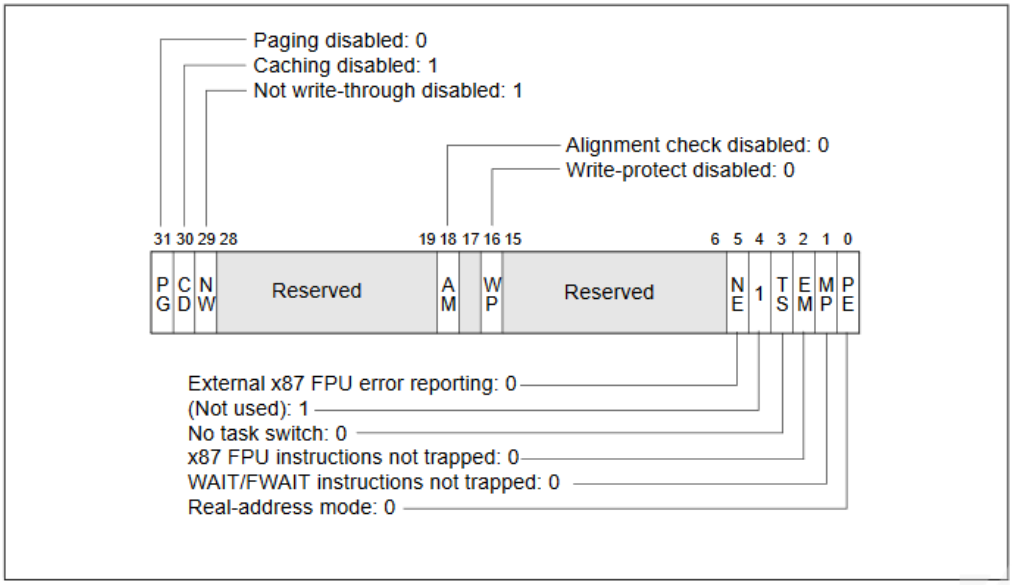


Figure 9-1. Contents of CR0 Register after Reset

```
// ■■■■■■■■■
NTSTATUS MemoryPageProtectOff()
{
   __asm
   {
       pushad;
       pushfd;

       mov eax, cr0;
       // ■■■■■■■■■■■■■ WP = 1 ■■..■■■■■■
       and eax, ~0x10000;
       mov cr0, eax;

       popfd;
       popad;

   }
}

// ■■■■■■■■■
NTSTATUS MemoryPageProtectOn()
{
   __asm
   {
       pushad;
       pushfd;

       mov eax, cr0;
       or eax, 0x10000;
       mov cr0, eax;

       popfd;
       popad;

   }
}
```

（4）构造一个新得段描述符，修改门描述符中的段选择子，跳转到我们构造得段描述符中，触发我们自定义得函数，完成IDT HOOK：

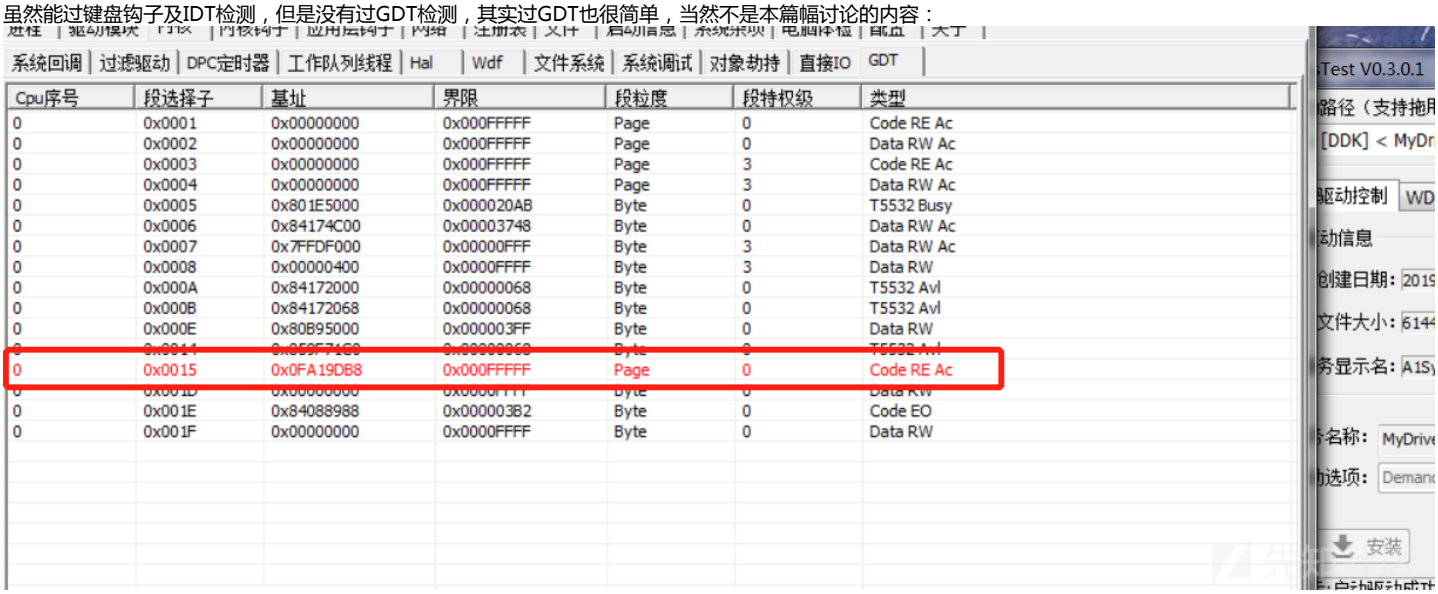
构造新得有两种方式:

一个手动填充中段描述符的各类属性，第二个是直接拷贝GDT[1]段属性描述符在修改，拷贝时候最好先看IDT段选择子对应的GDT中描述符，然后根据HOOK的函数在做拷贝

```
// ■■■■■■■■
void __declspec(naked) MyFinter()
{
```

```
    // GDT■■■■■■■■■■HOOK
    KdPrint(("Process: %s\n", (char*)PsGetCurrentProcess() + 0x16c));
    DbgBreakPoint();
    // ■■■■■■■■■■■■
    __asm CALL g_OldDescriptorAddress;
}

// ■■■■gdtr[xx]■■■gdtr[xx].BaseAddress = MyFunction ■IDT[0x93].selector■■■■■■■■■
NTSTATUS InstallIDT()
{
    // ■■IDT[0x93].selector■■■■■■■■■■■■■■■■■■
    g_OldDescriptorAddress = GetkeyIdtAddress(1);
    ULONG AddrNew = ((unsigned int)MyFinter - g_OldDescriptorAddress);
    DbgBreakPoint();
    // ■■gdtr■■■■
    char sgdtr[6] = { 0, };
    PKGDTENTRY sgdtrDataArr = NULL;
    // ■■■■■■■■■■PIDTR■■■■■■■■IDTR■■
    PIDTR TempgdtrBaseaddress = NULL;
    __asm SGDT sgdtr
    // sgdt dgt
    ULONG gdtrBaseAddr = 0;
    sgdtrDataArr = (PKGDTENTRY)sgdtr;
    TempgdtrBaseaddress = (PIDTR)sgdtr;
    ULONG gdtrBase = MAKELONG(TempgdtrBaseaddress->IDT_LOWbase, TempgdtrBaseaddress->IDT_HIGbase);
    // 500003ff`807f2abc
    DbgBreakPoint();
    // 1. ■■■■■
    MemoryPageProtectOff();
    // ■■GDT[21]■■■■■ 8■Bit 0■■■
    ULONG gdtrBase21 = (gdtrBase + sizeof(KGDTENTRY) * 0x15);
    DbgBreakPoint();
    // ■GDT[1]■■■GDT[21],■■■■■■■■
    RtlCopyMemory((PVOID)gdtrBase21, (PVOID)(gdtrBase + sizeof(KGDTENTRY)), sizeof(KGDTENTRY));
    DbgBreakPoint();
    // ■■■■■■■■■BaseAddress■■■■■■■■■■■■
    sgdtrDataArr = (PKGDTENTRY)gdtrBase21;
    sgdtrDataArr->HighWord.Bytes.BaseMid = (UCHAR)(((unsigned int)AddrNew >> 16) & 0xff);
    sgdtrDataArr->HighWord.Bytes.BaseHi = (UCHAR)((unsigned int)AddrNew >> 24);
    sgdtrDataArr->BaseLow = (USHORT)((unsigned int)AddrNew & 0x0000FFFF);
    DbgBreakPoint();
    MemoryPageProtectOn();
}
```

虽然能过键盘钩子及IDT检测，但是没有过GDT检测，其实过GDT也很简单，当然不是本篇幅讨论的内容：



上述中构建新GDT描述符位置索引0xA8或者0x4B（第九项）都可以，保证0~3Bit为0，涉及指令的权限检测，有兴趣的可以查一下Inter手册。其实中断门、调用门、任务门

点击收藏 | 0 关注 | 1

上一篇：Joomla3.0.0-3.4.6... 下一篇：记一次webshell的获取

1. 2 条回复

 wonderkun 2019-10-30 11:10:42

在windows10上 hook KeyboardInterruptService 的操作跟书上说的还有些区别，我大概记录了一下我的代码。
https://blog.wonderkun.cc/2019/04/02/Hook%20KeyboardClassServiceCallback%20实现内核态按键记录和模拟/

0 回复Ta

 一半人生 2019-10-30 17:51:24

@wonderkun 对的谢谢，上述过IDT HOOK或说GDT HOOK都是基于win7测试的，win10上 KeyboardInterruptService HOOK没有尝试过。

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板