x64内核：过滤驱动

Hook驱动的方法

1. 将驱动的回调替换掉——Hook驱动的IRP
    - 缺点：如果是异步操作，那么Hook的时候很大概率是拿不到数据的
2. 分层驱动 —— 给需要Hook的驱动加一层驱动（即过滤驱动）
    - `IoAttachDevice`附加到设备上，不管同步还是异步都可以收到数据
    - 如果在本层驱动完成了请求，那么下层驱动就收不到了
    - 需要设置一样的通信方式、跳过当前IRP堆栈`IoSkipCurrentIrpStackLocation`、调下层驱动`IoCallDriver`
    - 拿信息需要注册请求完成的回调`IoSetCompletionRoutine`，请求完成了会调用此回调，且在回调用要有引用计数，在驱动卸载的时候注销回调（先取消Hook驱动、判断所有派遣都没人调用之后，在返回）
    - 如果下层有的派遣函数，上层也必须实现派遣（实现全部派遣，派发给下层，感兴趣的单独实现）

关于IRP：http://www.cppblog.com/momoxiao/archive/2009/12/30/104470.html

过滤驱动

相关API

IoAttachDevice

将调用方的设备对象附加到一个命名的目标设备对象，以便为目标设备绑定的I/O请求首先路由到调用方

```
NTSTATUS IoAttachDevice(
  PDEVICE_OBJECT  SourceDevice,     // 指向调用者创建的设备对象的指针
  PUNICODE_STRING TargetDevice,     // 指向缓冲区的指针，该缓冲区包含要附加指定SourceDevice的设备对象的名称
  PDEVICE_OBJECT  *AttachedDevice   // 指向指针的调用者分配存储的指针。返回时，如果附加成功，则包含指向目标设备对象的指针
);
```

IoDetachDevice

在调用方的设备对象和下一个驱动程序的设备对象之间释放一个附加设备

```
void IoDetachDevice(
  PDEVICE_OBJECT TargetDevice   // 指向下层驱动程序的设备对象的指针。调用方先前成功调用了IoAttachDevice或IoAttachDeviceToDeviceStack以获取此指针
);
```

IoAttachDeviceToDeviceStack

将调用者的设备对象附加到链中最高的设备对象，并返回指向先前最高的设备对象的指针

```
PDEVICE_OBJECT IoAttachDeviceToDeviceStack(
  PDEVICE_OBJECT SourceDevice,
  PDEVICE_OBJECT TargetDevice
);
```

IoSetCompletionRoutine

```
void IoSetCompletionRoutine(
  PIRP                    Irp,                    // 指向驱动程序正在处理的IRP的指针
  PIO_COMPLETION_ROUTINE CompletionRoutine,      // 指定驱动程序提供的IoCompletion例程的入口点，它在下一个驱动程序完成包时被调用
  __drv_aliasesMem PVOID Context,                // 指向由驱动程序决定的上下文的指针，以传递到IoCompletion例程
  BOOLEAN                 InvokeOnSuccess,        // 指定在IRP的IO_STATUS_BLOCK结构中使用成功状态值完成IRP时是否调用完成例程
  BOOLEAN                 InvokeOnError,          // 指定如果IRP在IRP的IO_STATUS_BLOCK结构中以失败状态值完成
  BOOLEAN                 InvokeOnCancel          // 指定当驱动程序或内核调用IoCancelIrp来取消IRP时是否调用完成例程
);
```

实例

以键盘过滤驱动为例：

- 全局数据

```
typedef struct tagKEYBDINPUT {
    short wVk;
    short wScan;
    int dwFlags;
    int time;
} KEYBDINPUT, *PKEYBDINPUT;


PDRIVER_OBJECT driver_obj = NULL;            // 驱动对象
PDEVICE_OBJECT attached_device = NULL;       // attach设备

ULONG ref_count = 0;    // 引用计数
```

- 入口函数中注册全部派遣函数，针对读操作单独实现，并安装键盘过滤驱动`MyAttachDevice`

```
_Use_decl_annotations_
NTSTATUS
```

```
DriverEntry(struct _DRIVER_OBJECT* DriverObject, PUNICODE_STRING  RegistryPath)
{
    KdBreakPoint();
    DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_ERROR_LEVEL, "Hello Kernel! -
Install\n");

    // 注册卸载函数
    DriverObject->DriverUnload = DriverUnload;

    // 注册派遣函数
    for (ULONG i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
    {
        DriverObject->MajorFunction[i] = DispatchCommand;
    }
    DriverObject->MajorFunction[IRP_MJ_READ] = DispatchRead;

    // 设置IO模式
    DriverObject->Flags |= DO_BUFFERED_IO;    // 复制缓冲区，在R0中开辟空间，复制缓冲区
内容到此处

    NTSTATUS status;     // 返回状态

    // 创建设备对象，一般为IO管理器管理，内核设备FILE_DEVICE_UNKNOWN
    UNICODE_STRING driver_name;
    RtlInitUnicodeString(&driver_name, MY_DEVICE_NAME);
    PDEVICE_OBJECT device_object_ptr = NULL;
    status = IoCreateDevice(DriverObject, 0, &driver_name, FILE_DEVICE_UNKNOWN,
FILE_DEVICE_SECURE_OPEN, FALSE, &device_object_ptr);

    if(!NT_SUCCESS(status)) {
        DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_ERROR_LEVEL, "IoCreateDevice
Error: %p\n", status);
        return status;
    }

    // 键盘过滤驱动
    MyAttachDevice(device_object_ptr);

    // 创建符号连接
    UNICODE_STRING symbo_name;
    RtlInitUnicodeString(&symbo_name, MY_SYMBOL_NAME);
    status = IoCreateSymbolicLink(&symbo_name, &driver_name);

    if (!NT_SUCCESS(status)) {
        DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_ERROR_LEVEL,
"IoCreateSymbolicLink Error: %p\n", status);
        return status;
    }

    return STATUS_SUCCESS;
}
```

- 安装键盘过滤驱动`MyAttachDevice`，并设置同等的IO方式

```
void MyAttachDevice(PDEVICE_OBJECT source_dev)
{
    UNICODE_STRING dst_dev_name;
    RtlInitUnicodeString(&dst_dev_name, L"\\Device\\KeyboardClass0");

    IoAttachDevice(source_dev, &dst_dev_name, &attached_device);
    source_dev->Flags = attached_device->Flags;
}
```

- 相关派遣函数

```
NTSTATUS DispatchCommand(IN PDEVICE_OBJECT pDevObj, IN PIRP pIrp)
{
    // 跳过当前irp堆栈
    IoSkipCurrentIrpStackLocation(pIrp);

    // 调用下一层驱动
    NTSTATUS status = IoCallDriver(attached_device, pIrp);
    return status;
}

NTSTATUS DispatchRead(IN PDEVICE_OBJECT pDevObj, IN PIRP pIrp)
{
    InterlockedIncrement(&ref_count);

    // 跳过当前irp堆栈
    IoSkipCurrentIrpStackLocation(pIrp);

    // 注册请求完成的回调
    IoSetCompletionRoutine(pIrp, IoCompletion, NULL, TRUE, TRUE, TRUE);

    // 调用下一层驱动
    NTSTATUS status = IoCallDriver(attached_device, pIrp);

    if (pIrp->PendingReturned) {
        IoMarkIrpPending(pIrp);
    }

    return status;
}
```

- 派遣中注册的完成回调

```
NTSTATUS IoCompletion(
    __in PDEVICE_OBJECT  DeviceObject,
    __in PIRP  Irp,
    __in PVOID  Context
)
{
    // 获取缓冲区与缓冲区大小
    PKEYBDINPUT keybdinput = (PKEYBDINPUT)Irp->AssociatedIrp.SystemBuffer;
    ULONG_PTR keydb_size = Irp->IoStatus.Information;

    // 遍历打印
    for (ULONG_PTR i = 0; i < keydb_size / sizeof(KEYBDINPUT); i++) {
```

```
        DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_ERROR_LEVEL, "IoCompletion
IRQL:%d pInput:%p wVk:%04X wScan:%04X dwFlags:%d\n",
            KeGetCurrentIrql(), &keybdinput[i], keybdinput[i].wVk,
keybdinput[i].wScan, keybdinput[i].dwFlags);
    }

    InterlockedDecrement(&ref_count);

    return STATUS_SUCCESS;
}
```

- 卸载函数

```
// 卸载驱动
void DriverUnload(struct _DRIVER_OBJECT* DriverObject)
{
    // 删除设备
    IoDeleteDevice(DriverObject->DeviceObject);

    // 删除符号链接，如果有的话
    UNICODE_STRING symbo_name;
    RtlInitUnicodeString(&symbo_name, MY_SYMBOL_NAME);
    IoDeleteSymbolicLink(&symbo_name);

    // Detach设备
    if (attached_device) {
        IoDetachDevice(attached_device);
        IoDeleteDevice(attached_device);
    }

    while (ref_count != 0) {
        LARGE_INTEGER interval;
        interval.QuadPart = -10 * 1000 * 3000;
        KeDelayExecutionThread(KernelMode, 0, &interval);
    }

    DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_ERROR_LEVEL, "Hello Kernel! -
UnInstall\n");
}
```