在之前的文章中`LyShark`一直都在教大家如何让驱动程序与应用层进行`正向通信`，而在某些时候我们不仅仅只需要正向通信，也需要反向通信，例如杀毒软件如果驱动程序拦截到恶意操作则必须将这个请求动态的转发到应用层以此来通知用户，而这种通信方式的实现有多种，通常可以使用创建Socket套接字的方式实现，亦或者使用本章所介绍的通过`事件同步`的方法实现反向通信。

基于事件同步方式实现的通信需要用的最重要函数`IoCreateNotificationEvent()`这是微软定为开发者提供的。

IoCreateNotificationEvent 例程创建或打开一个命名通知事件，用于通知一个或多个执行线程已发生事件。

```
PKEVENT IoCreateNotificationEvent(
  [in]  PUNICODE_STRING EventName,
  [out] PHANDLE         EventHandle
);
```

其中的第二个参数`EventHandle`指向返回事件对象的内核句柄的位置的指针。此处为了能更好的接收和管理指针与进程之间的关系我们最好定义一个`DEVICE_EXTEN`结构体。

```
// 自定义设备扩展
typedef struct
{
    HANDLE  hProcess;         // HANDLE
    PKEVENT pkProcessEvent;   // 事件对象
    HANDLE  hProcessId;       // PID
    HANDLE  hpParProcessId;   // PPID
    BOOLEAN bIsCreateMark;    // Flag
}DEVICE_EXTEN, *PDEVICE_EXTEN;
```

驱动入口处`PsSetCreateProcessNotifyRoutine`则用于创建一个进程回调，该回调函数被指定为`pMyCreateProcessThreadRoutine`当回调函数被创建后，一旦有新进程创建则会执行函数内部的具体流程。

如代码所示，在`pMyCreateProcessThreadRoutine`函数内首先通过`(PDEVICE_EXTEN)GlobalDevObj->DeviceExtension`得到了自定义设备扩展指针，接着将当前进程的`PID,PPID,isCreate`等属性赋予到扩展指针中，当一切准备就绪后，通过调用`KeSetEvent`将当前进程事件设置为有信号状态，设置后重置为默认值。

```
// 自定义回调函数
VOID pMyCreateProcessThreadRoutine(IN HANDLE pParentId, HANDLE hProcessId,
BOOLEAN bisCreate)
{
    PDEVICE_EXTEN pDeviceExten = (PDEVICE_EXTEN)GlobalDevObj->DeviceExtension;

    // 将进行信息依次复制到结构体中
    pDeviceExten->hProcessId = hProcessId;
    pDeviceExten->hpParProcessId = pParentId;
    pDeviceExten->bIsCreateMark = bisCreate;

    // 设置为有信号
    KeSetEvent(pDeviceExten->pkProcessEvent, 0, FALSE);

    // 重置状态信号
```

```
    KeResetEvent(pDeviceExten->pkProcessEvent);
}
```

此时由于客户端中通过`OpenEventW(SYNCHRONIZE, FALSE, EVENT_NAME)`打开了内核对象，并通过`WaitForSingleObject(hProcessEvent, INFINITE)`一直在等待事件，一旦内核驱动`KeSetEvent(pDeviceExten->pkProcessEvent, 0, FALSE)`设置为有信号状态，则应用层会通过`DeviceIoControl`向内核层发送`IOCTL`控制信号并等待接收数据。

此时主派遣函数`DisPatchIoControl`被触发执行，通过`(PPROCESS_PTR)pUserOutPutBuffer`获取到应用层缓冲区设备指针，并依次通过`pBuffer->`的方式设置参数，最后返回状态码，此时应用层`PROCESS_PTR`中也就得到了进程创建的相关信息。

```
pBuffer = (PPROCESS_PTR)pUserOutPutBuffer;
uIoControl = pIrpStack->Parameters.DeviceIoControl.IoControlCode;
uReadLen = pIrpStack->Parameters.DeviceIoControl.InputBufferLength;
uWriteLen = pIrpStack->Parameters.DeviceIoControl.OutputBufferLength;

switch (uIoControl)
{
case IOCTL:
    pDeviceExten = (PDEVICE_EXTEN)GlobalDevObj->DeviceExtension;
    pBuffer->hProcessId = pDeviceExten->hProcessId;
    pBuffer->hpParProcessId = pDeviceExten->hpParProcessId;
    pBuffer->bIsCreateMark = pDeviceExten->bIsCreateMark;
    break;
default:
    ntStatus = STATUS_INVALID_PARAMETER;
    uWriteLen = 0;
    break;
}
```

如上就是内核层与应用层的部分代码功能分析，接下来我将完整代码分享出来，大家可以自行测试效果。

驱动程序WinDDK.sys完整代码；

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include <ntifs.h>
#include <ntstrsafe.h>

#define IOCTL CTL_CODE(FILE_DEVICE_UNKNOWN,0x801,METHOD_BUFFERED,FILE_ANY_ACCESS)

UNICODE_STRING GlobalSym = { 0 };
PDEVICE_OBJECT GlobalDevObj;

// 自定义设备扩展
typedef struct
{
    HANDLE  hProcess;          // HANDLE
    PKEVENT pkProcessEvent;    // 事件对象
```

```
    HANDLE  hProcessId;       // PID
    HANDLE  hpParProcessId;   // PPID
    BOOLEAN bIsCreateMark;    // Flag
}DEVICE_EXTEN, *PDEVICE_EXTEN;

// 自定义结构体(临时)
typedef struct
{
    HANDLE hProcessId;
    HANDLE hpParProcessId;
    BOOLEAN bIsCreateMark;
}PROCESS_PTR, *PPROCESS_PTR;

// 自定义回调函数
VOID pMyCreateProcessThreadRoutine(IN HANDLE pParentId, HANDLE hProcessId,
BOOLEAN bisCreate)
{
    PDEVICE_EXTEN pDeviceExten = (PDEVICE_EXTEN)GlobalDevObj->DeviceExtension;

    // 将进行信息依次复制到结构体中
    pDeviceExten->hProcessId = hProcessId;
    pDeviceExten->hpParProcessId = pParentId;
    pDeviceExten->bIsCreateMark = bisCreate;

    // 设置为有信号
    KeSetEvent(pDeviceExten->pkProcessEvent, 0, FALSE);

    // 重置状态信号
    KeResetEvent(pDeviceExten->pkProcessEvent);
}

// 默认派遣函数
NTSTATUS DisPatchComd(PDEVICE_OBJECT pDeviceObject, PIRP pIrp)
{
    pIrp->IoStatus.Information = 0;
    pIrp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);

    return pIrp->IoStatus.Status;
}

// 主派遣函数
NTSTATUS DisPatchIoControl(PDEVICE_OBJECT pDeviceObject, PIRP pIrp)
{
    NTSTATUS ntStatus;
    PIO_STACK_LOCATION pIrpStack;
    PVOID pUserOutPutBuffer;
    PPROCESS_PTR pBuffer;
    ULONG uIoControl = 0;
    ULONG uReadLen;
    ULONG uWriteLen;
    PDEVICE_EXTEN pDeviceExten;

    pIrpStack = IoGetCurrentIrpStackLocation(pIrp);
    pUserOutPutBuffer = pIrp->AssociatedIrp.SystemBuffer;
```

```
    pBuffer = (PPROCESS_PTR)pUserOutPutBuffer;
    uIoControl = pIrpStack->Parameters.DeviceIoControl.IoControlCode;
    uReadLen = pIrpStack->Parameters.DeviceIoControl.InputBufferLength;
    uWriteLen = pIrpStack->Parameters.DeviceIoControl.OutputBufferLength;

    switch (uIoControl)
    {
    case IOCTL:
        pDeviceExten = (PDEVICE_EXTEN)GlobalDevObj->DeviceExtension;
        pBuffer->hProcessId = pDeviceExten->hProcessId;
        pBuffer->hpParProcessId = pDeviceExten->hpParProcessId;
        pBuffer->bIsCreateMark = pDeviceExten->bIsCreateMark;
        break;
    default:
        ntStatus = STATUS_INVALID_PARAMETER;
        uWriteLen = 0;
        break;
    }

    pIrp->IoStatus.Information = uWriteLen;
    pIrp->IoStatus.Status = ntStatus;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    return ntStatus;
}

// 卸载驱动
VOID DriverUnLoad(PDRIVER_OBJECT pDriverObject)
{
    NTSTATUS ntStatus;
    UNICODE_STRING SymboLicLinkStr = { 0 };

    // 删除符号链接
    ntStatus = RtlUnicodeStringInit(&SymboLicLinkStr,
L"\\DosDevices\\Symboliclnk");
    ntStatus = IoDeleteSymbolicLink(&SymboLicLinkStr);

    // 注销进程消息回调
    PsSetCreateProcessNotifyRoutine(pMyCreateProcessThreadRoutine, TRUE);
    IoDeleteDevice(pDriverObject->DeviceObject);
}

// 入口函数
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING RegistryPath)
{
    NTSTATUS ntStatus = NULL;
    PDEVICE_OBJECT pDeviceObject = NULL;
    UNICODE_STRING uDeviceName = { 0 };
    UNICODE_STRING uEventName = { 0 };

    // 初始化字符串
    ntStatus = RtlUnicodeStringInit(&uDeviceName, L"\\Device\\ProcLook");
    ntStatus = RtlUnicodeStringInit(&GlobalSym, L"\\DosDevices\\Symboliclnk");
    ntStatus = RtlUnicodeStringInit(&uEventName,
L"\\BaseNamedObjects\\ProcLook");
    DbgPrint("hello lyshark.com");
    if (!NT_SUCCESS(ntStatus))
```

```
    {
        return ntStatus;
    }

    // 创建设备
    ntStatus =
IoCreateDevice(pDriver,sizeof(DEVICE_EXTEN),&uDeviceName,FILE_DEVICE_UNKNOWN,FILE
_DEVICE_SECURE_OPEN,FALSE,&pDeviceObject);
    DbgPrint("[LyShark] 创建设备对象");

    if (!NT_SUCCESS(ntStatus))
    {
        IoDeleteDevice(pDeviceObject);
        return ntStatus;
    }
    pDriver->Flags |= DO_BUFFERED_IO;
    GlobalDevObj = pDeviceObject;

    // 获取设备扩展指针,并IoCreateNotificationEvent创建一个R3到R0的事件
    PDEVICE_EXTEN pDeviceExten = (PDEVICE_EXTEN)pDeviceObject->DeviceExtension;
    pDeviceExten->pkProcessEvent = IoCreateNotificationEvent(&uEventName,
&pDeviceExten->hProcess);
    KeClearEvent(pDeviceExten->pkProcessEvent);
    DbgPrint("[LyShark] 创建事件回调");

    // 创建符号链接
    ntStatus = IoCreateSymbolicLink(&GlobalSym, &uDeviceName);
    if (!NT_SUCCESS(ntStatus))
    {
        IoDeleteDevice(pDeviceObject);
        return ntStatus;
    }

    // 注册进程创建回调
    ntStatus = PsSetCreateProcessNotifyRoutine(pMyCreateProcessThreadRoutine,
FALSE);
    DbgPrint("[LyShark] 创建进程回调");
    if (!NT_SUCCESS(ntStatus))
    {
        IoDeleteDevice(pDeviceObject);
        return ntStatus;
    }

    // 初始化派遣函数
    for (ULONG i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
    {
        pDriver->MajorFunction[i] = DisPatchComd;
    }
    pDriver->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DisPatchIoControl;

    // 关闭驱动
    pDriver->DriverUnload = DriverUnLoad;
    return ntStatus;
}
```

应用层客户端程序 `lyshark.exe` 完整代码;

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include <iostream>
#include <Windows.h>

#define EVENT_NAME L"Global\\ProcLook"
#define IOCTL CTL_CODE(FILE_DEVICE_UNKNOWN,0x801,METHOD_BUFFERED,FILE_ANY_ACCESS)

typedef struct
{
    HANDLE hProcessId;
    HANDLE hpParProcessId;
    BOOLEAN bIsCreateMark;
}PROCESS_PTR, *PPROCESS_PTR;

int main(int argc, char* argv[])
{
    PROCESS_PTR Master = { 0 };
    PROCESS_PTR Slave = { 0 };
    DWORD dwRet = 0;
    BOOL bRet = FALSE;

    // 打开驱动设备对象
    HANDLE hDriver = CreateFile(L"\\\\.\\Symboliclnk",GENERIC_READ |
GENERIC_WRITE,NULL,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
    if (hDriver == INVALID_HANDLE_VALUE)
    {
        return 0;
    }

    // 打开内核事件对象
    HANDLE hProcessEvent = OpenEventW(SYNCHRONIZE, FALSE, EVENT_NAME);

    if (NULL == hProcessEvent)
    {
        return 0;
    }

    while (TRUE)
    {
        // 等待事件
        WaitForSingleObject(hProcessEvent, INFINITE);

        // 发送控制信号
        bRet =
DeviceIoControl(hDriver,IOCTL,NULL,0,&Master,sizeof(Master),&dwRet,NULL);
        if (!bRet)
        {
            return 0;
        }
```

```
        if (bRet)
        {
            if (Master.hpParProcessId != Slave.hpParProcessId ||
Master.hProcessId != Slave.hProcessId || Master.bIsCreateMark !=
Slave.bIsCreateMark)
            {
                if (Master.bIsCreateMark)
                {
                    printf("[LyShark] 父进程: %d | 进程PID: %d | 状态: 创建进程 \n",
Master.hpParProcessId, Master.hProcessId);
                }
                else
                {
                    printf("[LyShark] 父进程: %d | 进程PID: %d | 状态: 退出进程 \n",
Master.hpParProcessId, Master.hProcessId);
                }
                Slave = Master;
            }
        }
        else
        {
            break;
        }
    }

    CloseHandle(hDriver);
    return 0;
}
```

手动编译这两个程序，将驱动签名后以管理员身份运行`lyshark.exe`客户端，此时屏幕中即可看到滚动输出效果，如此一来就实现了循环传递参数的目的。