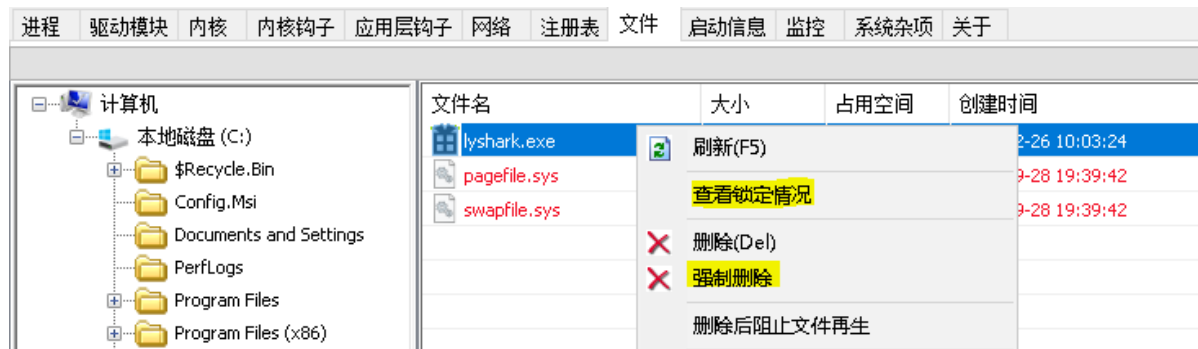


在某些时候我们的系统中会出现一些无法被正常删除的文件，如果想要强制删除则需要在驱动层面对其进行解锁后才可删掉，而所谓的解锁其实就是释放掉文件描述符（句柄表）占用，文件解锁的核心原理是通过调用 `ObSetHandleAttributes` 函数将特定句柄设置为可关闭状态，然后在调用 `ZwClose` 将其文件关闭，强制删除则是通过 `ObReferenceObjectByHandle` 在对象上提供相应的权限后直接调用 `ZwDeleteFile` 将其删除，虽此类代码较为普遍，但作为揭秘ARK工具来说也必须要将其分析并讲解一下。



首先封装 `lyshark.h` 通用头文件，并定义好我们所需要的结构体，以及特定未导出函数的声明，此处的定义部分是微软官方的规范，如果不懂结构具体含义可自行去微软官方查阅参考资料。

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

#include <ntddk.h>

// -----
// 引用微软结构
// -----
// 结构体定义
typedef struct _HANDLE_INFO
{
    UCHAR ObjectTypeIndex;
    UCHAR HandleAttributes;
    USHORT Handlevalue;
    ULONG GrantedAccess;
    ULONG64 Object;
    UCHAR Name[256];
} HANDLE_INFO, *PHANDLE_INFO;

HANDLE_INFO HandleInfo[1024];

typedef struct _SYSTEM_HANDLE_TABLE_ENTRY_INFO
{
    USHORT UniqueProcessId;
    USHORT CreatorBackTraceIndex;
    UCHAR ObjectTypeIndex;
    UCHAR HandleAttributes;
    USHORT Handlevalue;
    PVOID Object;
    ULONG GrantedAccess;
} SYSTEM_HANDLE_TABLE_ENTRY_INFO, *PSYSTEM_HANDLE_TABLE_ENTRY_INFO;

typedef struct _SYSTEM_HANDLE_INFORMATION
```

```

{
    ULONG64 NumberOfHandles;
    SYSTEM_HANDLE_TABLE_ENTRY_INFO Handles[1];
} SYSTEM_HANDLE_INFORMATION, *PSYSTEM_HANDLE_INFORMATION;

typedef enum _OBJECT_INFORMATION_CLASS
{
    ObjectBasicInformation,
    ObjectNameInformation,
    ObjectTypeInformation,
    ObjectAllInformation,
    ObjectDataInformation
} OBJECT_INFORMATION_CLASS, *POBJECT_INFORMATION_CLASS;

typedef struct _OBJECT_BASIC_INFORMATION
{
    ULONG                Attributes;
    ACCESS_MASK          DesiredAccess;
    ULONG                HandleCount;
    ULONG                ReferenceCount;
    ULONG                PagedPoolUsage;
    ULONG                NonPagedPoolUsage;
    ULONG                Reserved[3];
    ULONG                NameInformationLength;
    ULONG                TypeInformationLength;
    ULONG                SecurityDescriptorLength;
    LARGE_INTEGER        CreationTime;
} OBJECT_BASIC_INFORMATION, *POBJECT_BASIC_INFORMATION;

typedef struct _OBJECT_TYPE_INFORMATION
{
    UNICODE_STRING        TypeName;
    ULONG                TotalNumberOfHandles;
    ULONG                TotalNumberOfObjects;
    WCHAR                Unused1[8];
    ULONG                HighWaterNumberOfHandles;
    ULONG                HighWaterNumberOfObjects;
    WCHAR                Unused2[8];
    ACCESS_MASK          InvalidAttributes;
    GENERIC_MAPPING       GenericMapping;
    ACCESS_MASK          ValidAttributes;
    BOOLEAN              SecurityRequired;
    BOOLEAN              MaintainHandleCount;
    USHORT               MaintainTypeList;
    POOL_TYPE            PoolType;
    ULONG                DefaultPagedPoolCharge;
    ULONG                DefaultNonPagedPoolCharge;
} OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;

typedef struct _KAPC_STATE
{
    LIST_ENTRY ApcListHead[2];
    PVOID Process;
    BOOLEAN KernelApcInProgress;
    BOOLEAN KernelApcPending;
    BOOLEAN UserApcPending;
}

```

```

}KAPC_STATE, *PKAPC_STATE;

typedef struct _OBJECT_HANDLE_FLAG_INFORMATION
{
    BOOLEAN Inherit;
    BOOLEAN ProtectFromClose;
}OBJECT_HANDLE_FLAG_INFORMATION, *POBJECT_HANDLE_FLAG_INFORMATION;

typedef struct _LDR_DATA_TABLE_ENTRY64
{
    LIST_ENTRY64 InLoadOrderLinks;
    LIST_ENTRY64 InMemoryOrderLinks;
    LIST_ENTRY64 InInitializationOrderLinks;
    ULONG64 DllBase;
    ULONG64 EntryPoint;
    ULONG64 SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG Flags;
    USHORT LoadCount;
    USHORT TlsIndex;
    LIST_ENTRY64 HashLinks;
    ULONG64 SectionPointer;
    ULONG64 CheckSum;
    ULONG64 TimeDateStamp;
    ULONG64 LoadedImports;
    ULONG64 EntryPointActivationContext;
    ULONG64 PatchInformation;
    LIST_ENTRY64 ForwarderLinks;
    LIST_ENTRY64 ServiceTagLinks;
    LIST_ENTRY64 StaticLinks;
    ULONG64 ContextInformation;
    ULONG64 OriginalBase;
    LARGE_INTEGER LoadTime;
} LDR_DATA_TABLE_ENTRY64, *PLDR_DATA_TABLE_ENTRY64;

// -----
// 导出函数定义
// -----

NTKERNELAPI NTSTATUS ObSetHandleAttributes
(
    HANDLE Handle,
    POBJECT_HANDLE_FLAG_INFORMATION HandleFlags,
    KPROCESSOR_MODE PreviousMode
);

NTKERNELAPI VOID KeStackAttachProcess
(
    PEPROCESS PROCESS,
    PKAPC_STATE ApcState
);

NTKERNELAPI VOID KeUnstackDetachProcess
(
    PKAPC_STATE ApcState

```

```

);

NTKERNELAPI NTSTATUS PsLookupProcessByProcessId
(
    IN HANDLE ProcessId,
    OUT PEPROCESS *Process
);

NTSYSAPI NTSTATUS NTAPI ZwQueryObject
(
    HANDLE Handle,
    ULONG ObjectInformationClass,
    PVOID ObjectInformation,
    ULONG ObjectInformationLength,
    PULONG ReturnLength OPTIONAL
);

NTSYSAPI NTSTATUS NTAPI ZwQuerySystemInformation
(
    ULONG SystemInformationClass,
    PVOID SystemInformation,
    ULONG SystemInformationLength,
    PULONG ReturnLength
);

NTSYSAPI NTSTATUS NTAPI ZwDuplicateObject
(
    HANDLE SourceProcessHandle,
    HANDLE SourceHandle,
    HANDLE TargetProcessHandle OPTIONAL,
    PHANDLE TargetHandle OPTIONAL,
    ACCESS_MASK DesiredAccess,
    ULONG HandleAttributes,
    ULONG Options
);

NTSYSAPI NTSTATUS NTAPI ZwOpenProcess
(
    PHANDLE ProcessHandle,
    ACCESS_MASK AccessMask,
    POBJECT_ATTRIBUTES ObjectAttributes,
    PCLIENT_ID ClientId
);

#define STATUS_INFO_LENGTH_MISMATCH 0xC0000004

```

接下来将具体分析如何解锁指定文件的句柄表，强制解锁文件句柄表，大体步骤如下所示。

- 1.首先调用 `ZwQuerySystemInformation` 的16功能号 `SystemHandleInformation` 来枚举系统里的句柄。
- 2.通过 `ZwOpenProcess()` 打开拥有此句柄的进程，通过 `ZwDuplicateObject` 创建一个新的句柄，并把此句柄复制到自己的进程内。
- 3.通过调用 `ZwQueryObject` 并传入 `ObjectNameInformation` 查询到句柄的名称，并将其放入到 `pNameInfo` 变量内。

- 4.循环这个过程并在每次循环中通过 `strstr()` 判断是否是我们需要关闭的文件名，如果是则调用 `ForceCloseHandle` 强制解除占用。
- 5.此时会进入到 `ForceCloseHandle` 流程内，通过 `KeStackAttachProcess` 附加到进程内，并调用 `ObSetHandleAttributes` 将句柄设置为可关闭状态。
- 6.最后调用 `ZwClose` 关闭句柄占用，并 `KeUnstackDetachProcess` 脱离该进程。

实现代码流程非常容易理解，此类功能也没有其他别的写法了一般也就这种，但是还是需要注意这些内置函数的参数传递，这其中 `ZwQuerySystemInformation()` 一般用于查询系统进程等信息居多，但通过对 `SystemInformationClass` 变量传入不同的参数可实现对不同结构的枚举工作，具体的定义可去查阅微软定义规范；

```
NTSTATUS WINAPI ZwQuerySystemInformation(
    _In_      SYSTEM_INFORMATION_CLASS SystemInformationClass,    // 传入不同参数则
    输出不同内容
    _Inout_   PVOID SystemInformation,                          // 输出数据
    _In_      ULONG SystemInformationLength,                    // 长度
    _Out_opt_ PULONG ReturnLength                               // 返回长度
);
```

函数 `ZwDuplicateObject()`，该函数例程用于创建一个句柄，该句柄是指定源句柄的副本，此函数的具体声明部分如下；

```
NTSYSAPI NTSTATUS ZwDuplicateObject(
    [in]      HANDLE SourceProcessHandle,    // 要复制的句柄的源进程的句柄。
    [in]      HANDLE SourceHandle,          // 要复制的句柄。
    [in, optional] HANDLE TargetProcessHandle, // 要接收新句柄的目标进程的句柄。
    [out, optional] PHANDLE TargetHandle,    // 指向例程写入新重复句柄的
    HANDLE 变量的指针。
    [in]      ACCESS_MASK DesiredAccess,    // 一个 ACCESS_MASK 值，该值指定
    新句柄的所需访问。
    [in]      ULONG HandleAttributes,      // 一个 ULONG，指定新句柄的所需
    属性。
    [in]      ULONG Options                // 一组标志，用于控制重复操作的行为。
);
```

函数 `ZwQueryObject()` 其可以返回特定的一个对象参数，此函数尤为注意第二个参数，当下我们传入的是 `ObjectNameInformation` 则代表需要取出对象名称，而如果使用 `ObjectTypeInformation` 则是返回对象类型，该函数微软定义如下所示；

```

NTSYSAPI NTSTATUS ZwQueryObject(
    [in, optional] HANDLE Handle, // 要获取相关信息的对象句柄。
    [in] OBJECT_INFORMATION_CLASS ObjectInformationClass, // 该值确定 ObjectInformation 缓冲区中返回的信息的类型。
    [out, optional] PVOID ObjectInformation, // 指向接收请求信息的调用方分配缓冲区的指针。
    [in] ULONG ObjectInformationLength, // 指定 ObjectInformation 缓冲区的大小（以字节为单位）。
    [out, optional] PULONG ReturnLength // 指向接收所请求密钥信息的大小（以字节为单位）的变量的指针。
);

```

而对于 `ForceCloseHandle` 函数中，需要注意的只有一个 `ObSetHandleAttributes` 该函数微软并没有格式化文档，但是也并不影响我们使用它，如下最需要注意的是 `PreviousMode` 变量，该变量如果传入 `kernelMode` 则是内核模式，传入 `UserMode` 则代表用户模式，为了权限最大化此处需要写入 `kernelMode` 模式；

```

NTSYSAPI NTSTATUS ObSetHandleAttributes(
    HANDLE Handle, // 传入文件句柄
    POBJECT_HANDLE_FLAG_INFORMATION HandleFlags, //
    OBJECT_HANDLE_FLAG_INFORMATION标志
    KPROCESSOR_MODE PreviousMode // 指定运行级别kernelMode
)

```

实现文件解锁，该驱动程序不仅可用于解锁应用层程序，也可用于解锁驱动，如下代码中我们解锁 `pagefile.sys` 程序的句柄占用；

```

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

#include "lyshark.h"

// 根据PID得到EProcess
PEPROCESS LookupProcess(HANDLE Pid)
{
    PEPROCESS eprocess = NULL;
    if (NT_SUCCESS(PsLookupProcessByProcessId(Pid, &eprocess)))
        return eprocess;
    else
        return NULL;
}

// 将unicode转为char*
VOID UnicodeStringToCharArray(PUNICODE_STRING dst, char *src)
{
    ANSI_STRING string;
    if (dst->Length > 260)
    {
        return;
    }
}

```

```

        RtlUnicodeStringToAnsiString(&string, dst, TRUE);
        strcpy(src, string.Buffer);
        RtlFreeAnsiString(&string);
    }

    // 强制关闭句柄
VOID ForceCloseHandle(PEPROCESS Process, ULONG64 HandleValue)
{
    HANDLE h;
    KAPC_STATE ks;
    OBJECT_HANDLE_FLAG_INFORMATION ohfi;

    if (Process == NULL)
    {
        return;
    }
    // 验证进程是否可读写
    if (!MmIsAddressValid(Process))
    {
        return;
    }

    // 附加到进程
    KeStackAttachProcess(Process, &ks);
    h = (HANDLE)HandleValue;
    ohfi.Inherit = 0;
    ohfi.ProtectFromClose = 0;

    // 设置句柄为可关闭状态
    ObSetHandleAttributes(h, &ohfi, KernelMode);

    // 关闭句柄
    ZwClose(h);

    // 脱离附加进程
    KeUnstackDetachProcess(&ks);

    DbgPrint("EP = [ %d ] | HandleValue = [ %d ] 进程句柄已被关闭\n", Process, HandleValue);
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动卸载成功\n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("Hello LyShark.com\n");

    PVOID Buffer;
    ULONG BufferSize = 0x20000, rtl = 0;
    NTSTATUS Status, qost = 0;
    NTSTATUS ns = STATUS_SUCCESS;
    ULONG64 i = 0;

```

```

ULONG64 qwHandleCount;

SYSTEM_HANDLE_TABLE_ENTRY_INFO *p;
OBJECT_BASIC_INFORMATION BasicInfo;
POBJECT_NAME_INFORMATION pNameInfo;

ULONG ulProcessID;
HANDLE hProcess;
HANDLE hHandle;
HANDLE hDupObj;
CLIENT_ID cid;
OBJECT_ATTRIBUTES oa;
CHAR szFile[260] = { 0 };

Buffer = ExAllocatePoolWithTag(NonPagedPool, BufferSize, "LyShark");
memset(Buffer, 0, BufferSize);

// SystemHandleInformation
Status = ZwQuerySystemInformation(16, Buffer, BufferSize, 0);
while (Status == STATUS_INFO_LENGTH_MISMATCH)
{
    ExFreePool(Buffer);
    BufferSize = BufferSize * 2;
    Buffer = ExAllocatePoolWithTag(NonPagedPool, BufferSize, "LyShark");
    memset(Buffer, 0, BufferSize);
    Status = ZwQuerySystemInformation(16, Buffer, BufferSize, 0);
}

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

// 获取系统中所有句柄表
qwHandleCount = ((SYSTEM_HANDLE_INFORMATION *)Buffer)->NumberOfHandles;

// 得到句柄表的SYSTEM_HANDLE_TABLE_ENTRY_INFO结构
p = (SYSTEM_HANDLE_TABLE_ENTRY_INFO *)((SYSTEM_HANDLE_INFORMATION *)Buffer)->Handles;

// 初始化HandleInfo数组
memset(HandleInfo, 0, 1024 * sizeof(HANDLE_INFO));

// 开始枚举句柄
for (i = 0; i < qwHandleCount; i++)
{
    ulProcessID = (ULONG)p[i].UniqueProcessId;
    cid.UniqueProcess = (HANDLE)ulProcessID;
    cid.UniqueThread = (HANDLE)0;
    hHandle = (HANDLE)p[i].HandleValue;

    // 初始化对象结构
    InitializeObjectAttributes(&oa, NULL, 0, NULL, NULL);

    // 通过句柄信息打开占用进程
    ns = ZwOpenProcess(&hProcess, PROCESS_DUP_HANDLE, &oa, &cid);

```



```

        // 打开错误
        if (!NT_SUCCESS(ns))
        {
            continue;
        }

        // 创建一个句柄, 该句柄是指定源句柄的副本。
        ns = ZwDuplicateObject(hProcess, hHandle, NtCurrentProcess(), &hDupObj,
        PROCESS_ALL_ACCESS, 0, DUPLICATE_SAME_ACCESS);
        if (!NT_SUCCESS(ns))
        {
            continue;
        }

        // 查询对象句柄的信息并放入BasicInfo
        ZwQueryObject(hDupObj, ObjectBasicInformation, &BasicInfo,
        sizeof(OBJECT_BASIC_INFORMATION), NULL);

        // 得到对象句柄的名字信息
        pNameInfo = ExAllocatePool(PagedPool, 1024);
        RtlZeroMemory(pNameInfo, 1024);

        // 查询对象信息中的对象名, 并将该信息保存到pNameInfo中
        gost = ZwQueryObject(hDupObj, ObjectNameInformation, pNameInfo, 1024,
        &rtl);

        // 获取信息并关闭句柄
        unicodeStringToCharArray(&(pNameInfo->Name), szFile);
        ExFreePool(pNameInfo);
        ZwClose(hDupObj);
        ZwClose(hProcess);

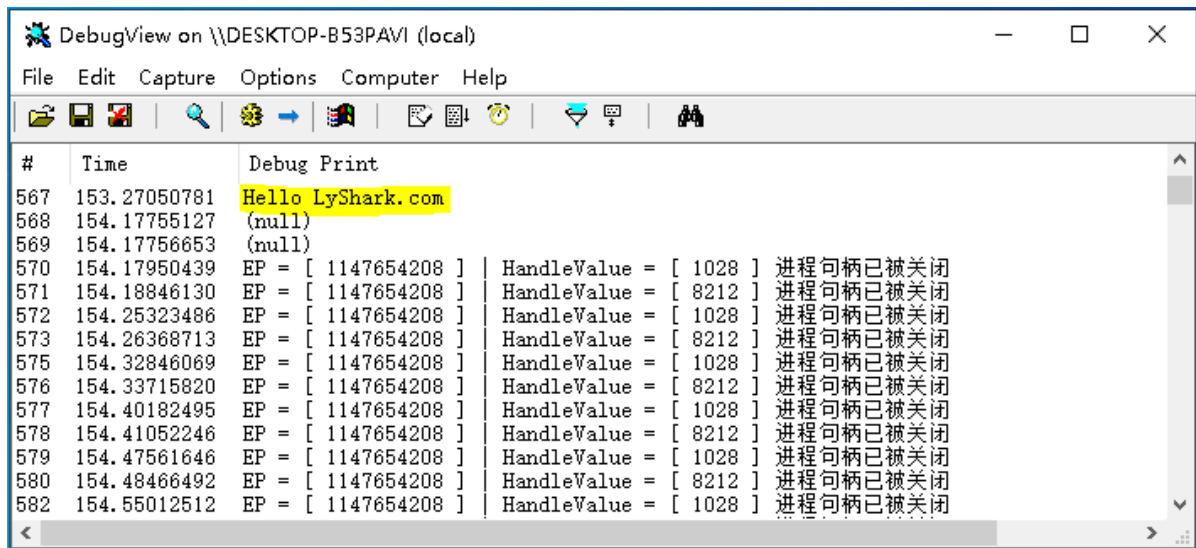
        // 检查句柄是否被占用, 如果被占用则关闭文件并删除
        if (strstr(_strlwr(szFile), "pagefile.sys"))
        {
            PEPROCESS ep = LookupProcess((HANDLE)(p[i].UniqueProcessId));

            // 占用则强制关闭
            ForceCloseHandle(ep, p[i].HandleValue);
            ObDereferenceObject(ep);
        }
    }

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}

```

编译并运行这段驱动程序, 则会将 `pagefile.sys` 内核文件进行解锁, 输出效果如下所示;



```
DebugView on \\DESKTOP-B53PAWI (local)
File Edit Capture Options Computer Help

# Time Debug Print
567 153.27050781 Hello LyShark.com
568 154.17755127 (null)
569 154.17756653 (null)
570 154.17950439 EP = [ 1147654208 ] | HandleValue = [ 1028 ] 进程句柄已被关闭
571 154.18846130 EP = [ 1147654208 ] | HandleValue = [ 8212 ] 进程句柄已被关闭
572 154.25323486 EP = [ 1147654208 ] | HandleValue = [ 1028 ] 进程句柄已被关闭
573 154.26368713 EP = [ 1147654208 ] | HandleValue = [ 8212 ] 进程句柄已被关闭
575 154.32846069 EP = [ 1147654208 ] | HandleValue = [ 1028 ] 进程句柄已被关闭
576 154.33715820 EP = [ 1147654208 ] | HandleValue = [ 8212 ] 进程句柄已被关闭
577 154.40182495 EP = [ 1147654208 ] | HandleValue = [ 1028 ] 进程句柄已被关闭
578 154.41052246 EP = [ 1147654208 ] | HandleValue = [ 8212 ] 进程句柄已被关闭
579 154.47561646 EP = [ 1147654208 ] | HandleValue = [ 1028 ] 进程句柄已被关闭
580 154.48466492 EP = [ 1147654208 ] | HandleValue = [ 8212 ] 进程句柄已被关闭
582 154.55012512 EP = [ 1147654208 ] | HandleValue = [ 1028 ] 进程句柄已被关闭
```

聊完了文件解锁功能，接下来将继续探讨如何实现 强制删除 文件的功能，文件强制删除的关键在于 `ObReferenceObjectByHandle` 函数，该函数可在对象句柄上提供访问验证，并授予访问权限返回指向对象的正文的相应指针，当有了指定的权限以后则可以直接调用 `ZwDeleteFile()` 将文件强制删除。

在调用初始化句柄前提之下需要先调用 `KeGetCurrentIrql()` 函数，该函数返回当前 IRQL 级别，那么什么是 IRQL 呢？

Windows 中系统中断请求 (IRQ) 可分为两种，一种外部中断 (硬件中断)，一种是软件中断 (INT3)，微软将中断的概念进行了扩展，提出了中断请求级别 (IRQL) 的概念，其中就规定了 32 个中断请求级别。

- 其中 0-2 级为软中断，顺序由小到大分别是：PASSIVE_LEVEL, APC_LEVEL, DISPATCH_LEVEL
- 其中 27-31 为硬中断，顺序由小到大分别是：PROFILE_LEVEL, CLOCK1_LEVEL, CLOCK2_LEVEL, IPI_LEVEL, POWER_LEVEL, HIGH_LEVEL

我们的代码中开头部分 `KeGetCurrentIrql() > PASSIVE_LEVEL` 则是在判断当前的级别不大于 0 级，也就是说必须要大于 0 才可以继续执行。

好开始步入正题，函数 `ObReferenceObjectByHandle` 需要传入一个文件句柄，而此句柄需要通过 `IoCreateFileSpecifyDeviceObjectHint` 对其进行初始化，文件系统筛选器驱动程序使用 `IoCreateFileSpecifyDeviceObjectHint` 函数创建，该函数的微软完整定义如下所示；

```
NTSTATUS IoCreateFileSpecifyDeviceObjectHint(
    [out] PHANDLE FileHandle, // 指向变量的指针，该
    // 变量接收文件对象的句柄。
    [in] ACCESS_MASK DesiredAccess, // 标志的位掩码，指定
    // 调用方需要对文件或目录的访问类型。
    [in] POBJECT_ATTRIBUTES ObjectAttributes, // 指向已由
    // InitializeObjectAttributes 例程初始化的 OBJECT_ATTRIBUTES 结构的指针。
    [out] PIO_STATUS_BLOCK IoStatusBlock, // 指向
    // IO_STATUS_BLOCK 结构的指针，该结构接收最终完成状态和有关所请求操作的信息。
    [in, optional] PLARGE_INTEGER AllocationSize, // 指定文件的初始分配
    // 大小（以字节为单位）。
    [in] ULONG FileAttributes, // 仅当文件创建、取代
    // 或在某些情况下被覆盖时，才会应用显式指定的属性。
    [in] ULONG ShareAccess, // 指定调用方希望的对
    // 文件的共享访问类型（为零或 1，或以下标志的组合）。
    [in] ULONG Disposition, // 指定一个值，该值确
    // 定要执行的操作，具体取决于文件是否存在。
```

```

    [in]                ULONG                CreateOptions,           // 指定要在创建或打开
文件时应用的选项。
    [in, optional] PVOID                EaBuffer,           // 指向调用方提供的
FILE_FULL_EA_INFORMATION结构化缓冲区的指针。
    [in]                ULONG                EaLength,           // EaBuffer 的长度
(以字节为单位)。
    [in]                CREATE_FILE_TYPE    CreateFileType,       // 驱动程序必须将此参
数设置为 CreateFileTypeNone。
    [in, optional] PVOID                InternalParameters,     // 驱动程序必须将此参
数设置为 NULL。
    [in]                ULONG                Options,           // 指定要在创建请求期
间使用的选项。
    [in, optional] PVOID                DeviceObject           // 指向要向其发送创建
请求的设备对象的指针。
);

```

当调用 `IoCreateFileSpecifyDeviceObjectHint()` 函数完成初始化并创建设备后，则下一步就是调用 `ObReferenceObjectByHandle()` 并传入初始化好的设备句柄到 `Handle` 参数上，

```

NTSTATUS ObReferenceObjectByHandle(
    [in]                HANDLE                Handle,           // 指定对象的打开句
柄。
    [in]                ACCESS_MASK            DesiredAccess,     // 指定对对象的请求
访问类型。
    [in, optional] POBJECT_TYPE                ObjectType,       // 指向对象类型的指
针。
    [in]                KPROCESSOR_MODE        AccessMode,       // 指定要用于访问检
查的访问模式。 它必须是 UserMode 或 KernelMode。
    [out]               PVOID                *Object,           // 指向接收指向对象
正文的指针的变量的指针。
    [out, optional] POBJECT_HANDLE_INFORMATION HandleInformation // 驱动程序将此设置
为 NULL。
);

```

通过调用如上两个函数将权限设置好以后，我们再手动将 `ImageSectionObject` 也就是映像节对象填充为0，然后再将 `DeleteAccess` 删除权限位打开，最后调用 `ZwDeleteFile()` 函数即可实现强制删除文件的效果，其核心代码如下所示：

```

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

#include "lyshark.h"

// 强制删除文件
BOOLEAN ForceDeleteFile(UNICODE_STRING pwzFileName)
{
    PEPROCESS pCurEprocess = NULL;
    KAPC_STATE kpc = { 0 };
    OBJECT_ATTRIBUTES fileob;
    HANDLE hFile = NULL;
    NTSTATUS status = STATUS_UNSUCCESSFUL;
    IO_STATUS_BLOCK iosta;

```

```

PDEVICE_OBJECT DeviceObject = NULL;
PVOID pHandleFileObject = NULL;

// 判断中断等级不大于0
if (KeGetCurrentIrql() > PASSIVE_LEVEL)
{
    return FALSE;
}
if (pwzFileName.Buffer == NULL || pwzFileName.Length <= 0)
{
    return FALSE;
}

__try
{
    // 读取当前进程的EProcess
    pCurEprocess = IoGetCurrentProcess();

    // 附加进程
    KeStackAttachProcess(pCurEprocess, &kapc);

    // 初始化结构
    InitializeObjectAttributes(&fileOb, &pwzFileName, OBJ_CASE_INSENSITIVE |
OBJ_KERNEL_HANDLE, NULL, NULL);

    // 文件系统筛选器驱动程序 仅向指定设备对象下面的筛选器和文件系统发送创建请求。
    status = IoCreateFileSpecifyDeviceObjectHint(&hFile,
        SYNCHRONIZE | FILE_WRITE_ATTRIBUTES | FILE_READ_ATTRIBUTES |
FILE_READ_DATA,
        &fileOb,
        &iosta,
        NULL,
        0,
        FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE,
        FILE_OPEN,
        FILE_NON_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT,
        0,
        0,
        CreateFileTypeNone,
        0,
        IO_IGNORE_SHARE_ACCESS_CHECK,
        DeviceObject);
    if (!NT_SUCCESS(status))
    {
        return FALSE;
    }

    // 在对象句柄上提供访问验证, 如果可以授予访问权限, 则返回指向对象的正文的相应指针。
    status = ObReferenceObjectByHandle(hFile, 0, 0, 0, &pHandleFileObject,
0);

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

```

```

        // 镜像节对象设置为0
        ((PFILE_OBJECT)(pHandleFileObject))->SectionObjectPointer-
>ImageSectionObject = 0;

        // 删除权限打开
        ((PFILE_OBJECT)(pHandleFileObject))->DeleteAccess = 1;

        // 调用删除文件API
        status = ZwDeleteFile(&fileOb);
        if (!NT_SUCCESS(status))
        {
            return FALSE;
        }
    }

    _finally
    {
        if (pHandleFileObject != NULL)
        {
            ObDereferenceObject(pHandleFileObject);
            pHandleFileObject = NULL;
        }
        KeUnstackDetachProcess(&kapc);

        if (hFile != NULL || hFile != (PVOID)-1)
        {
            ZwClose(hFile);
            hFile = (PVOID)-1;
        }
    }
    return TRUE;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动卸载成功 \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("Hello LyShark.com \n");

    UNICODE_STRING local_path;
    UNICODE_STRING file_path;
    BOOLEAN ref = FALSE;

    // 初始化被删除文件
    RtlInitUnicodeString(&file_path, L"\\??\\C:\\lyshark.exe");

    // 获取自身驱动文件
    local_path = ((PLDR_DATA_TABLE_ENTRY64)Driver->DriverSection)->FullDllName;

    // 删除lyshark.exe
    ref = ForceDeleteFile(file_path);
    if (ref == TRUE)
    {

```

```

        DbgPrint("[+] 已删除 %wZ \n",file_path);
    }

    // 删除WinDDK.sys
    ref = ForceDeleteFile(local_path);
    if (ref == TRUE)
    {
        DbgPrint("[+] 已删除 %wZ \n", local_path);
    }

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}

```

编译并运行如上程序，则会分别将 c://lyshark.exe 以及驱动程序自身删除，并输出如下图所示的提示信息；

