本章将探索驱动程序开发的基础部分，了解驱动对象 DRIVER\_OBJECT 结构体的定义，一般来说驱动程序

# DriverEntry 入口处都会存在这样一个驱动对象，该对象内所包含的就是当前所加载驱动自身的一些详细参数，例如驱动大小，驱动标志，驱动名，驱动节等等，每一个驱动程序都会存在这样的一个结构。

首先来看一下微软对其的定义，此处我已将重要字段进行了备注。

PDRIVER\_DISPATCH MajorFunction[IRP\_MJ\_MAXIMUM\_FUNCTION + 1];

} DRIVER\_OBJECT;

// 驱动对象的卸载地址

// 驱动类型

// 驱动大小

// 驱动对象

// 驱动的标志

// 驱动的起始位置

// 驱动的大小

// 指向驱动程序映像的内存区对象

// 驱动的扩展空间

// 驱动名字

typedef struct \_DRIVER\_OBJECT { CSHORT Type;

CSHORT Size;

PDEVICE\_OBJECT DeviceObject; ULONG Flags;

PVOID DriverStart; ULONG DriverSize; PVOID DriverSection;

PDRIVER\_EXTENSION DriverExtension; UNICODE\_STRING DriverName; PUNICODE\_STRING HardwareDatabase; PFAST\_IO\_DISPATCH FastIoDispatch; PDRIVER\_INITIALIZE DriverInit; PDRIVER\_STARTIO DriverStartIo;

PDRIVER\_UNLOAD DriverUnload;

DRIVER\_OBJECT结构体是 Windows 操作系统内核中用于表示驱动程序的基本信息的结构体。它包含了一系列的字段，用于描述驱动程序的特定属性。

以下是 DRIVER\_OBJECT 结构体中的一些重要字段：

# Type：该字段标识该结构体的类型，始终设置为DRIVER\_OBJECT\_TYPE。Size：该字段表示该结构体的大小，以字节为单位。

DeviceObject：该字段是一个指针，指向驱动程序所创建的设备对象链表的头部。每个设备对象代表着一个设备或者驱动程序创建的一种虚拟设备。

DriverStart：该字段是一个指针，指向驱动程序代码的入口点，也就是驱动程序的DriverEntry函数。 该函数会在驱动程序被加载时被调用。

DriverSize：该字段表示驱动程序代码的大小，以字节为单位。

DriverName：该字段是一个UNICODE\_STRING结构体，用于表示驱动程序的名称。

Flags：该字段是一个32位的位掩码，用于表示驱动程序的一些属性。例如，可以设置DO\_BUFFERED\_IO标志表示驱动程序支持缓冲I/O。

如果我们想要遍历出当前自身驱动的一些基本信息，我们只需要在驱动的头部解析 \_DRIVER\_OBJECT 即可得到全部的数据，这段代码可以写成如下样子，其中的 IRP\_MJ\_ 这一系列则是微软的调用号，不同的RIP代表着不同的涵义，但一般驱动也就会用到如下这几种调用号。

// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#include <ntifs.h>

VOID UnDriver(PDRIVER\_OBJECT driver)

{

DbgPrint(("Uninstall Driver Is OK \n"));

}

NTSTATUS DriverEntry(IN PDRIVER\_OBJECT Driver, PUNICODE\_STRING RegistryPath)

{

DbgPrint("hello lyshark \n");

Driver->DriverUnload = UnDriver;

DbgPrint("驱动名字 = %wZ \n", Driver->DriverName);

DbgPrint("驱动起始地址 = %p | 大小 = %x | 结束地址 %p \n",Driver-

>DriverStart,Driver->DriverSize,(ULONG64)Driver->DriverStart + Driver->DriverSize);

DbgPrint("卸载地址 = %p\n", Driver->DriverUnload);

DbgPrint("IRP\_MJ\_READ地址 = %p\n", Driver->MajorFunction[IRP\_MJ\_READ]);

DbgPrint("IRP\_MJ\_WRITE地址 = %p\n", Driver->MajorFunction[IRP\_MJ\_WRITE]);

DbgPrint("IRP\_MJ\_CREATE地址 = %p\n", Driver->MajorFunction[IRP\_MJ\_CREATE]);

DbgPrint("IRP\_MJ\_CLOSE地址 = %p\n", Driver->MajorFunction[IRP\_MJ\_CLOSE]);

DbgPrint("IRP\_MJ\_DEVICE\_CONTROL地址 = %p\n", Driver-

>MajorFunction[IRP\_MJ\_DEVICE\_CONTROL]);

// 输出完整的调用号

for (auto i = 0; i < IRP\_MJ\_MAXIMUM\_FUNCTION; i++)

{

DbgPrint("IRP\_MJ调用号 = %d | 函数地址 = %p \r\n", i, Driver-

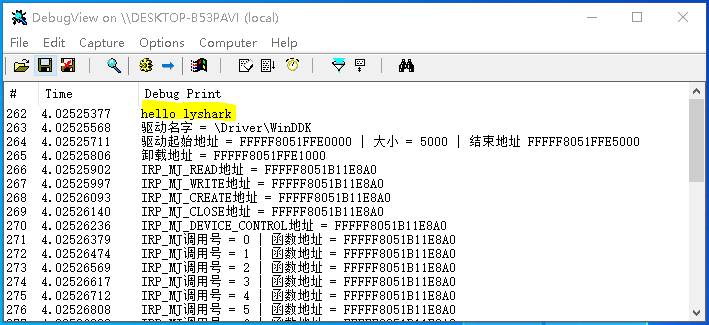
>MajorFunction[i]);

}

Driver->DriverUnload = UnDriver; return STATUS\_SUCCESS;

}

编译这段程序，签名并运行，我们即可看到如下输出信息，此时当前自身驱动的详细参数都可以被输出；



当然运用 \_DRIVER\_OBJECT 对象中的 DriverSection 字段我们完全可以遍历输出当前系统下所有的驱动程序的具体信息， DriverSection 结构指向了一个 \_LDR\_DATA\_TABLE\_ENTRY 结构，结构的微软定义如下；



typedef struct \_LDR\_DATA\_TABLE\_ENTRY { LIST\_ENTRY InLoadOrderLinks; LIST\_ENTRY InMemoryOrderLinks; LIST\_ENTRY InInitializationOrderLinks; PVOID DllBase;

PVOID EntryPoint;

ULONG SizeOfImage; UNICODE\_STRING FullDllName; UNICODE\_STRING BaseDllName; ULONG Flags;

USHORT LoadCount; USHORT TlsIndex; union {

LIST\_ENTRY HashLinks; struct {

PVOID SectionPointer;

ULONG CheckSum;

};

};

union {

struct {

ULONG TimeDateStamp;

};

struct {

PVOID LoadedImports;

};

};

}LDR\_DATA\_TABLE\_ENTRY, PLDR\_DATA\_TABLE\_ENTRY;

为了能够遍历出所有的系统驱动，我们需要得到 pLdr 结构，该结构可通过 Driver->DriverSection 的方式获取到，获取到之后通过 pLdr->InLoadOrderLinks.Flink 得到当前驱动的入口地址，而每一次调用pListEntry->Flink 都将会指向下一个驱动对象，通过不断地循环 CONTAINING\_RECORD 解析，即可输出当前系统内所有驱动的详细信息。这段程序的写法可以如下所示；

// 署名权



// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#include <ntifs.h>

typedef struct \_LDR\_DATA\_TABLE\_ENTRY { LIST\_ENTRY InLoadOrderLinks; LIST\_ENTRY InMemoryOrderLinks; LIST\_ENTRY InInitializationOrderLinks; PVOID DllBase;

PVOID EntryPoint; ULONG SizeOfImage;

UNICODE\_STRING FullDllName;

UNICODE\_STRING BaseDllName; ULONG Flags;

USHORT LoadCount; USHORT TlsIndex; union {

LIST\_ENTRY HashLinks; struct {

PVOID SectionPointer; ULONG CheckSum;

};

};

union {

struct {

ULONG TimeDateStamp;

};

struct {

PVOID LoadedImports;

};

};

}LDR\_DATA\_TABLE\_ENTRY, PLDR\_DATA\_TABLE\_ENTRY;

VOID UnDriver(PDRIVER\_OBJECT driver)

{

DbgPrint(("Uninstall Driver Is OK \n"));

}

NTSTATUS DriverEntry(IN PDRIVER\_OBJECT Driver, PUNICODE\_STRING RegistryPath)

{

DbgPrint("hello lyshark \n");

Driver->DriverUnload = UnDriver;

PLDR\_DATA\_TABLE\_ENTRY pLdr = NULL;

PLIST\_ENTRY pListEntry = NULL; PLIST\_ENTRY pCurrentListEntry = NULL;

PLDR\_DATA\_TABLE\_ENTRY pCurrentModule = NULL;

pLdr = (PLDR\_DATA\_TABLE\_ENTRY)Driver->DriverSection; pListEntry = pLdr->InLoadOrderLinks.Flink; pCurrentListEntry = pListEntry->Flink;

// 判断是否结束

while (pCurrentListEntry != pListEntry)

{

// 获取LDR\_DATA\_TABLE\_ENTRY结构

pCurrentModule = CONTAINING\_RECORD(pCurrentListEntry, LDR\_DATA\_TABLE\_ENTRY, InLoadOrderLinks);

if (pCurrentModule->BaseDllName.Buffer != 0)

{

DbgPrint("模块名 = %wZ | 模块基址 = %p | 模块入口 = %p | 模块时间戳 = %d \n", pCurrentModule->BaseDllName,

pCurrentModule->DllBase, pCurrentModule->EntryPoint, pCurrentModule->TimeDateStamp);

}

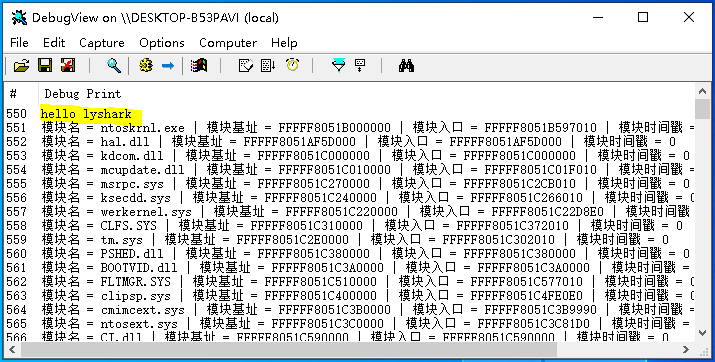
pCurrentListEntry = pCurrentListEntry->Flink;

}

Driver->DriverUnload = UnDriver; return STATUS\_SUCCESS;

}

# 编译这段程序，签名并运行，我们即可看到如下输出信息，此时当前自身驱动的详细参数都可以被输出；



通过使用上一篇文章 《驱动开发：内核字符串拷贝与比较》 中所介绍的的 RtlCompareUnicodeString 函数， 还可用于对比与过滤特定结果，以此来实现通过驱动名返回驱动基址的功能。

LONGLONG GetModuleBaseByName(PDRIVER\_OBJECT pDriverObj, UNICODE\_STRING ModuleName)

{

PLDR\_DATA\_TABLE\_ENTRY pLdr = NULL;

PLIST\_ENTRY pListEntry = NULL; PLIST\_ENTRY pCurrentListEntry = NULL;

PLDR\_DATA\_TABLE\_ENTRY pCurrentModule = NULL;

pLdr = (PLDR\_DATA\_TABLE\_ENTRY)pDriverObj->DriverSection; pListEntry = pLdr->InLoadOrderLinks.Flink; pCurrentListEntry = pListEntry->Flink;

while (pCurrentListEntry != pListEntry)

{

// 获取LDR\_DATA\_TABLE\_ENTRY结构

pCurrentModule = CONTAINING\_RECORD(pCurrentListEntry, LDR\_DATA\_TABLE\_ENTRY, InLoadOrderLinks);

if (pCurrentModule->BaseDllName.Buffer != 0)

{

// 对比模块名

if (RtlCompareUnicodeString(&pCurrentModule->BaseDllName, &ModuleName,

TRUE) == 0)

{

return (LONGLONG)pCurrentModule->DllBase;

}

}

pCurrentListEntry = pCurrentListEntry->Flink;

}

return 0;

}

上这段代码的使用也非常简单，通过传入一个 UNICODE\_STRING 类型的模块名，即可获取到模块基址并返 回，至于如何初始化 UNICODE\_STRING 则在 《驱动开发：内核字符串转换方法》 中有详细的介绍，此处你只需要这样来写。

NTSTATUS DriverEntry(IN PDRIVER\_OBJECT Driver, PUNICODE\_STRING RegistryPath)

{

DbgPrint("hello lyshark \n");

UNICODE\_STRING unicode;

// 获取WinDDK驱动基地址

RtlUnicodeStringInit(&unicode, L"WinDDK.sys");

LONGLONG winddk\_address = GetModuleBaseByName(Driver, unicode);

DbgPrint("WinDDK模块基址 = %p \n", winddk\_address);

// 获取ACPI驱动基地址

RtlUnicodeStringInit(&unicode, L"ACPI.sys");

LONGLONG acpi\_address = GetModuleBaseByName(Driver, unicode);

DbgPrint("ACPI模块基址 = %p \n", acpi\_address);

Driver->DriverUnload = UnDriver; return STATUS\_SUCCESS;

}

运行这段驱动程序，即可分别输出 WinDDK.sys 以及 ACPI.sys 两个驱动模块的基地址；

