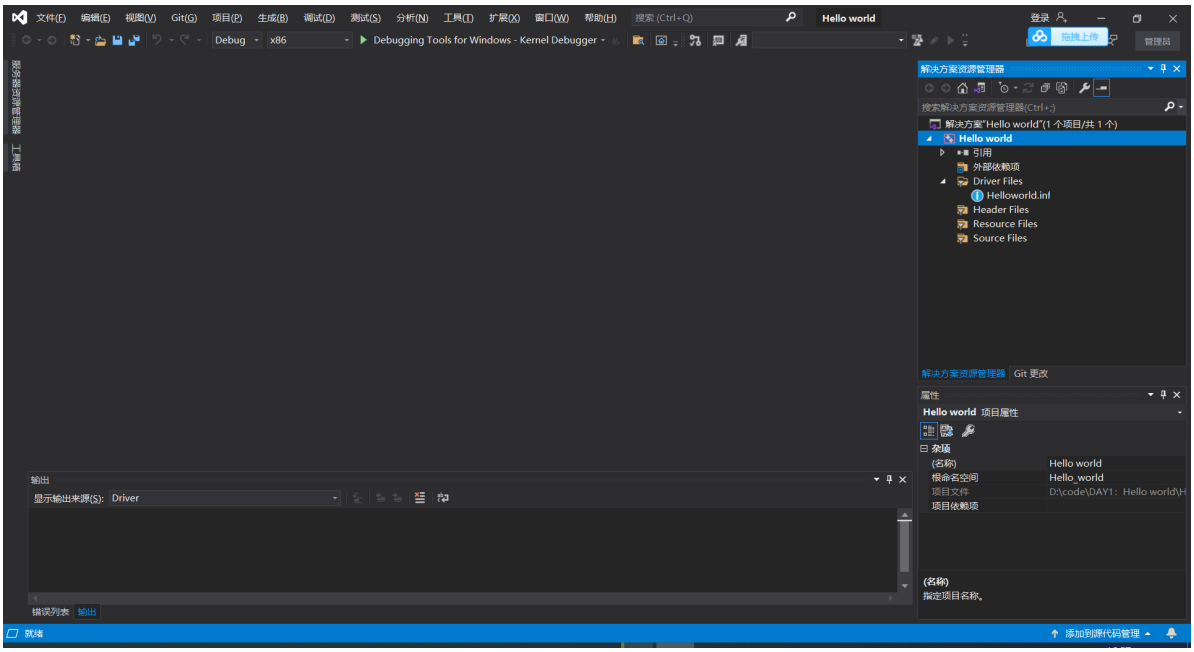
驱动开发

最近和火哥聊了一下，感觉很多知识点都没有学牢固，于是自己重新整理一下关于内核的笔记，重新学一下！温故而之心！

驱动 Hello World

打开vs2019-> 选择WDM，创建第一个驱动的项目



直接包含一个 `#include <ntifs.h>` 即可，因为这个头文件里包含了很多我们会经常用到的头文件，也不用考虑加载的顺序的问题，已经帮你准备好了

```
39  #pragma warning(push)
40  #endif
41
42  #pragma warning(disable:4115) // named type definition in parentheses
43  #pragma warning(disable:4201) // nameless struct/union
44  #pragma warning(disable:4214) // bit field types other than int
45
46
47  #include <ntddk.h>
48  #include <excpt.h>
49  #include <ntdef.h>
50  #include <ntnls.h>
51  #include <ntstatus.h>
52  #include <bugcodes.h>
53  #include <ntiologc.h>
54  #include <winapifamily.h>
55
56  //
57  // These macros are used to test, set and clear flags respectivly
58  //
```

考虑的问题：

- 驱动是进程吗?

当我们加载一个新的驱动的时候，他不是去创建了一个新的进程，而是有点像loadlibrary这样的操作，所以驱动本质来说是一个模块，就是给搞2G的内存驱动的空间来创建驱动

- NT类的驱动，他如果绑定了设备的话，不能卸载，只能进行一个重启的一个操作，所以如果我要更新的话，我只能进行一个重启的操作，也就不是一个热拔插的
- WDM类的驱动，是一个热拔插的方式，我卸载更新之后，不需要重启，也就是电源这一类处理的比较好，所以针对于一些服务器来说，不太能用NT类的驱动，实际上这两个差不多
- WDF类的驱动，是简化开发的，我们之前写那些驱动都是比较底层，是需要对原理比较熟悉的，WDF开发了一套框架，把WDF给封装了一下，做了一下架构，只需要调用封装体的东西，相当于事件驱动机制，这样做的话蓝屏的几率会大大的减少
- 那么WDF类的驱动里面还会有KWDF, UWDF KWDF主要是内核驱动的框架，UWDF是用户层的，就是在R3层会调用一些正常pe做不到的东西，所以如果想用这个，必须要对COM了解，COM是windows的一套组件，定义了一系列的接口

其实我们主要学的也是这种NT WDM这种的驱动

```
#include <ntifs.h>

VOID
DriverUnload(
    _In_ struct _DRIVER_OBJECT* DriverObject
)
{

}

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING Seg)
{
    DbgPrint("%d", *(int*)(0x9b26b000));
    pDriver->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}
```

```
struct _DRIVER_OBJECT
{
    SHORT Type;
//0x0
    SHORT Size;
//0x2
    struct _DEVICE_OBJECT* DeviceObject;
 //0x4
    ULONG Flags;
 //0x8
    VOID* DriverStart;
 //0xc
    ULONG DriverSize;
//0x10
    VOID* DriverSection;
 //0x14
    struct _DRIVER_EXTENSION* DriverExtension;
 //0x18
    struct _UNICODE_STRING DriverName;
 //0x1c
```

```
    struct _UNICODE_STRING* HardwareDatabase;
//0x24
    struct _FAST_IO_DISPATCH* FastIoDispatch;
//0x28
    LONG (*DriverInit)(struct _DRIVER_OBJECT* arg1, struct _UNICODE_STRING*
arg2); //0x2c
    VOID (*DriverStartIo)(struct _DEVICE_OBJECT* arg1, struct _IRP* arg2);
 //0x30
    VOID (*DriverUnload)(struct _DRIVER_OBJECT* arg1);
 //0x34
    LONG (*MajorFunction[28])(struct _DEVICE_OBJECT* arg1, struct _IRP* arg2);
//0x38
};
```

第一节总结来说，主要是讲了，一个基本的驱动的编写，以及驱动的入口点生成其实和我们写正常3层的c，是一个原理，都不是main，而是编译器默认的，又讲了当我们加载驱动后会在注册表`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services`下根据你驱动的名字进行一个注册

名称	类型	数据
(默认)	REG_SZ	(数值未设置)
DisplayName	REG_SZ	Text_address
ErrorControl	REG_DWORD	0x00000000 (0)
ImagePath	REG_EXPAN...	\??\C:\Users\Administrator\Desktop\Text_address.sys
Start	REG_DWORD	0x00000003 (3)
Type	REG_DWORD	0x00000001 (1)

当然了DisplayName是驱动的名称，ErrorControl是错误编号，ImagePath是全路径，Start代表的启动的方式

对于启动驱动来说，通过命令行`net start (name)`启动也可以

如果下了`DbgBreakPoint`断点相当于3层的`int 3`后，windbg会根据PE结构中的`Debug Directory RVA`去找

大概就是讲了这些第一节！

驱动基础

主要介绍了几个函数怎么使用

1.类型

2.字符串函数，申请内存函数

a)RtlInitString 初始化多字节ascii

b)RtlInitUnicodeString 初始化宽字符

c)RtlFreeUnicodeString 释放uncode字符串

d)RtlStringCbPrintfA 出格式化输 记得引用 #include <ntstrsafe.h>

e)RtlCompareUnicodeString 字符串比较

_2 申请内存
ExAllocatePool
ExFreePool

3.创建线程

```
HANDLE hThread = NULL;
    NTSTATUS status =PsCreateSystemThread(&hThread, THREAD_ALL_ACCESS, NULL,
NULL, NULL, work, NULL);
    if (NT_SUCCESS(status))
    {
        ZwClose(hThread);
    }
```

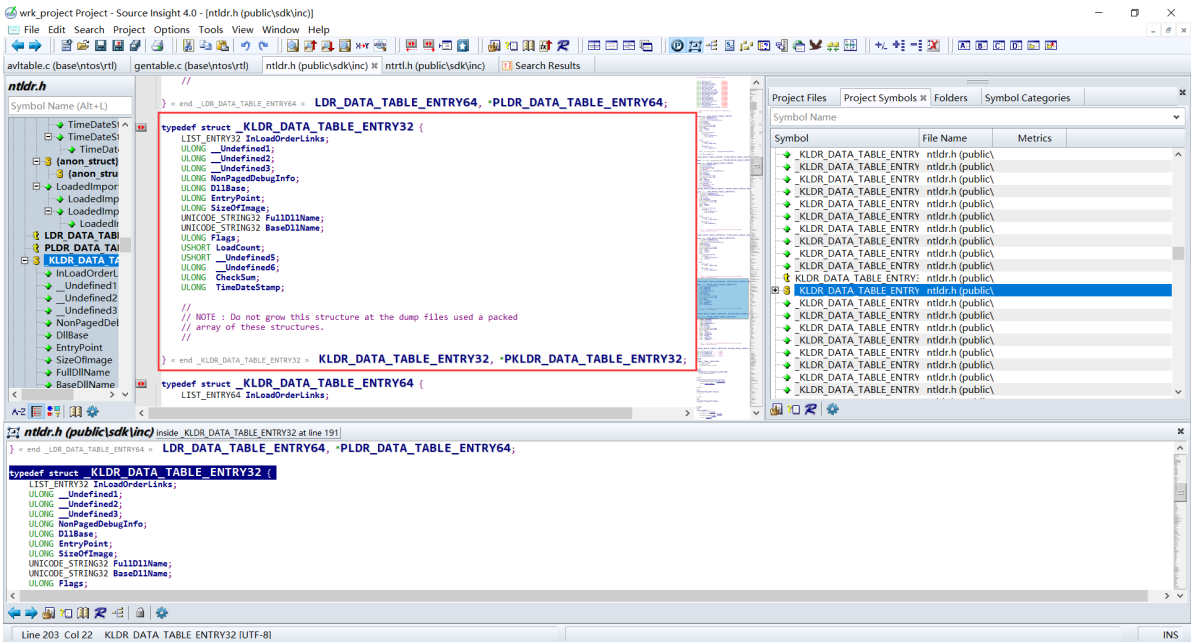
4.普通链表

5.内核链表

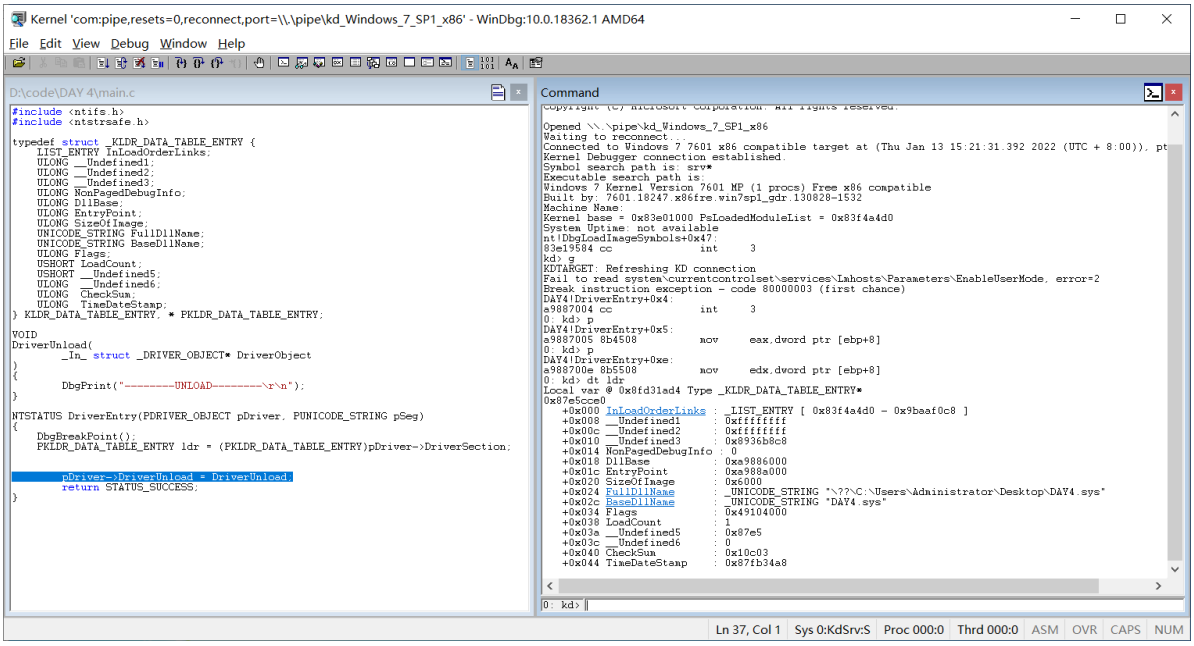
驱动断链

```
1: kd> dt _DRIVER_OBJECT
ntdll!_DRIVER_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
   +0x008 Flags            : Uint4B
   +0x00c DriverStart      : Ptr32 Void
   +0x010 DriverSize       : Uint4B
   +0x014 DriverSection    : Ptr32 Void
   +0x018 DriverExtension  : Ptr32 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING
   +0x024 HardwareDatabase : Ptr32 _UNICODE_STRING
   +0x028 FastIoDispatch   : Ptr32 _FAST_IO_DISPATCH
   +0x02c DriverInit       : Ptr32     long
   +0x030 DriverStartIo    : Ptr32     void
   +0x034 DriverUnload     : Ptr32     void
   +0x038 MajorFunction    : [28] Ptr32     long
```

主要的在DriverSection的结构里面，KLDR_DATA_TABLE_ENTRY



在windbg中看一下结构



- InLoadOrderLinks 是我们加载的链表
- DllBase 是我们的模块地址
- EntryPoint 是我们的入口点

观察一下pDriver和ldr的结构，发现PE头和入口点都能对上，说明没有什么问题

```
0: kd> dt ldr
Local var @ 0x8fd31ad4 Type _KLDR_DATA_TABLE_ENTRY*
0x87e5cce0
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x83f4a4d0 - 0x9baaf0c8 ]
   +0x008 __Undefined1     : 0xffffffff
   +0x00c __Undefined2     : 0xffffffff
   +0x010 __Undefined3     : 0x8936b8c8
   +0x014 NonPagedDebugInfo : 0
   +0x018 DllBase          : 0xa9886000
   +0x01c EntryPoint       : 0xa988a000
   +0x020 SizeOfImage      : 0x6000
   +0x024 FullDllName      : _UNICODE_STRING "\??\C:\Users\Administrator\Desktop\DAY4.sys"
   +0x02c BaseDllName      : _UNICODE_STRING "DAY4.sys"
   +0x034 Flags            : 0x49104000
   +0x038 LoadCount        : 1
   +0x03a __Undefined5     : 0x87e5
   +0x03c __Undefined6     : 0
   +0x040 CheckSum         : 0x10c03
   +0x044 TimeDateStamp    : 0x87fb34a8
0: kd> dt pDriver
Local var @ 0x8fd31ae0 Type _DRIVER_OBJECT*
0x9b53d498
   +0x000 Type             : 0n4
   +0x002 Size             : 0n168
   +0x004 DeviceObject     : (null)
   +0x008 Flags            : 2
   +0x00c DriverStart      : 0xa9886000 Void
   +0x010 DriverSize       : 0x6000
   +0x014 DriverSection    : 0x87e5cce0 Void
   +0x018 DriverExtension  : 0x9b53d540 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\DAY4"
   +0x024 HardwareDatabase : 0x8416e250 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYST
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xa988a000     long  DAY4!GsDriverEntry+0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : (null)
   +0x038 MajorFunction    : [28] 0x83eb60e5     long  nt!IopInvalidDeviceRequest+0
```

遍历LIST_ENTRY的双向循环链表 就可以把模块都遍历出来

```
0: kd> dt _KLDR_DATA_TABLE_ENTRY 0x86544c20
DAY4!_KLDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x86544ba0 - 0x86544c98 ]
   +0x008 __Undefined1     : 0x84215d50
   +0x00c __Undefined2     : 3
   +0x010 __Undefined3     : 0
   +0x014 NonPagedDebugInfo : 0
   +0x018 DllBase          : 0x84214000
   +0x01c EntryPoint       : 0x84214000
   +0x020 SizeOfImage      : 0x37000
```

```
   +0x024 FullDllName      : _UNICODE_STRING "\SystemRoot\system32\halmacpi.dll"
   +0x02c BaseDllName      : _UNICODE_STRING "hal.dll"
   +0x034 Flags            : 0x8004000
   +0x038 LoadCount        : 0x49
   +0x03a __Undefined5     : 0
   +0x03c __Undefined6     : 0
   +0x040 CheckSum         : 0x37fb1
   +0x044 TimeDateStamp    : 0
0: kd> dt _KLDR_DATA_TABLE_ENTRY 0x86544ba0
DAY4!_KLDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x86544b20 - 0x86544c20 ]
   +0x008 __Undefined1     : 0xffffffff
   +0x00c __Undefined2     : 0xffffffff
   +0x010 __Undefined3     : 0
   +0x014 NonPagedDebugInfo : 0
   +0x018 DllBase          : 0x80bd5000
   +0x01c EntryPoint       : 0x80bd7850
   +0x020 SizeOfImage      : 0x29000
   +0x024 FullDllName      : _UNICODE_STRING "\SystemRoot\system32\kdbazis.dll"
   +0x02c BaseDllName      : _UNICODE_STRING "kdcom.dll"
   +0x034 Flags            : 0x4000
   +0x038 LoadCount        : 3
   +0x03a __Undefined5     : 0
   +0x03c __Undefined6     : 0
   +0x040 CheckSum         : 0x6f7b
   +0x044 TimeDateStamp    : 0
```

驱动链表遍历：

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

typedef struct _KLDR_DATA_TABLE_ENTRY {
    LIST_ENTRY InLoadOrderLinks;
    LIST_ENTRY exp;
    ULONG un;
    ULONG NonPagedDebugInfo;
    ULONG DllBase;
    ULONG EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG Flags;
    USHORT LoadCount;
    USHORT __Undefined5;
    ULONG  __Undefined6;
    ULONG  CheckSum;
    ULONG  TimeDateStamp;
} KLDR_DATA_TABLE_ENTRY, * PKLDR_DATA_TABLE_ENTRY;

VOID
DriverUnload(
    _In_ struct _DRIVER_OBJECT* DriverObject
)
{
    DbgPrint("--------UNLOAD--------\r\n");
}
```

```
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pSeg)
{

    PKLDR_DATA_TABLE_ENTRY ldr = (PKLDR_DATA_TABLE_ENTRY)pDriver->DriverSection;
    PKLDR_DATA_TABLE_ENTRY pre = (PKLDR_DATA_TABLE_ENTRY)ldr-
>InLoadOrderLinks.Flink;
    PKLDR_DATA_TABLE_ENTRY next = (PKLDR_DATA_TABLE_ENTRY)pre-
>InLoadOrderLinks.Flink;
    int count = 0;
    while (pre != next)
    {
        DbgPrintEx(77, 0, "[%d] Driver name = %wZ\r\n",count, &next-
>FullDllName);
        next = (PKLDR_DATA_TABLE_ENTRY)next->InLoadOrderLinks.Flink;
        ++count;
    }

    pDriver->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}
```

```
[0] Driver name = \SystemRoot\system32\ntkrnlpa.exe
[1] Driver name = \SystemRoot\system32\halmacpi.dll
[2] Driver name = \SystemRoot\system32\kdbazis.dll
[3] Driver name = \SystemRoot\system32\mcupdate_GenuineIntel.dll
[4] Driver name = \SystemRoot\system32\PSHED.dll
[5] Driver name = \SystemRoot\system32\BOOTVID.dll
[6] Driver name = \SystemRoot\system32\CLFS.SYS
[7] Driver name = \SystemRoot\system32\CI.dll
[8] Driver name = \SystemRoot\system32\drivers\Wdf01000.sys
[9] Driver name = \SystemRoot\system32\drivers\WDFLDR.SYS
[10] Driver name = \SystemRoot\system32\drivers\ACPI.sys
[11] Driver name = \SystemRoot\system32\drivers\WMILIB.SYS
[12] Driver name = \SystemRoot\system32\drivers\msisadrv.sys
[13] Driver name = \SystemRoot\system32\drivers\pci.sys
[14] Driver name = \SystemRoot\system32\drivers\vdrvroot.sys
[15] Driver name = \SystemRoot\System32\drivers\partmgr.sys
[16] Driver name = \SystemRoot\system32\drivers\compbatt.sys
[17] Driver name = \SystemRoot\system32\drivers\BATTC.SYS
[18] Driver name = \SystemRoot\system32\drivers\volmgr.sys
[19] Driver name = \SystemRoot\System32\drivers\volmgrx.sys
[20] Driver name = \SystemRoot\system32\drivers\intelide.sys
[21] Driver name = \SystemRoot\system32\drivers\PCIIDEX.SYS
[22] Driver name = \SystemRoot\system32\DRIVERS\vmci.sys
[23] Driver name = \SystemRoot\System32\drivers\mountmgr.sys
[24] Driver name = \SystemRoot\system32\drivers\nvraid.sys
[25] Driver name = \SystemRoot\system32\drivers\CLASSPNP.SYS
[26] Driver name = \SystemRoot\system32\drivers\vsock.sys
[27] Driver name = \SystemRoot\system32\drivers\atapi.sys
[28] Driver name = \SystemRoot\system32\drivers\ataport.SYS
[29] Driver name = \SystemRoot\system32\drivers\lsi_sas.sys
[30] Driver name = \SystemRoot\system32\drivers\storport.sys
[31] Driver name = \SystemRoot\system32\drivers\amdsata.sys
[32] Driver name = \SystemRoot\system32\drivers\amdxata.sys
[33] Driver name = \SystemRoot\system32\drivers\amd_xata.sys
[34] Driver name = \SystemRoot\system32\drivers\fltmgr.sys
[35] Driver name = \SystemRoot\system32\drivers\fileinfo.sys
[36] Driver name = \SystemRoot\System32\Drivers\Ntfs.sys
[37] Driver name = \SystemRoot\System32\Drivers\msrpc.sys
[38] Driver name = \SystemRoot\System32\Drivers\ksecdd.sys
[39] Driver name = \SystemRoot\System32\Drivers\cng.sys
[40] Driver name = \SystemRoot\System32\drivers\pcw.sys
[41] Driver name = \SystemRoot\System32\Drivers\Fs_Rec.sys
[42] Driver name = \SystemRoot\system32\drivers\ndis.sys
[43] Driver name = \SystemRoot\system32\drivers\NETIO.SYS
[44] Driver name = \SystemRoot\System32\Drivers\ksecpkg.sys
[45] Driver name = \SystemRoot\System32\drivers\tcpip.sys
[46] Driver name = \SystemRoot\System32\drivers\fwpkclnt.sys
[47] Driver name = \SystemRoot\system32\drivers\vmstorfl.sys
[48] Driver name = \SystemRoot\system32\drivers\volsnap.sys
[49] Driver name = \SystemRoot\System32\Drivers\spldr.sys
[50] Driver name = \SystemRoot\System32\drivers\rdyboost.sys
[51] Driver name = \SystemRoot\System32\Drivers\mup.sys
[52] Driver name = \SystemRoot\system32\drivers\iaStorE.sys
```

驱动断链:

```
#include <ntifs.h>

extern POBJECT_TYPE *IoDriverObjectType;

NTKERNELAPI
NTSTATUS
ObReferenceObjectByName(
```

```
    __in PUNICODE_STRING ObjectName,
    __in ULONG Attributes,
    __in_opt PACCESS_STATE AccessState,
    __in_opt ACCESS_MASK DesiredAccess,
    __in POBJECT_TYPE ObjectType,
    __in KPROCESSOR_MODE AccessMode,
    __inout_opt PVOID ParseContext,
    __out PVOID* Object
);

typedef struct _KLDR_DATA_TABLE_ENTRY32 {
    LIST_ENTRY32 InLoadOrderLinks;
    ULONG __Undefined1;
    ULONG __Undefined2;
    ULONG __Undefined3;
    ULONG NonPagedDebugInfo;
    ULONG DllBase;
    ULONG EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING32 FullDllName;
    UNICODE_STRING32 BaseDllName;
    ULONG Flags;
    USHORT LoadCount;
    USHORT __Undefined5;
    ULONG  __Undefined6;
    ULONG  CheckSum;
    ULONG  TimeDateStamp;

    //
    // NOTE : Do not grow this structure at the dump files used a packed
    // array of these structures.
    //

} KLDR_DATA_TABLE_ENTRY32, * PKLDR_DATA_TABLE_ENTRY32;

VOID
DriverUnload(
    _In_ struct _DRIVER_OBJECT* DriverObject
)
{

}

void DirverHide(PDRIVER_OBJECT pDriverObj)
{
    LARGE_INTEGER large = { 0 };
    large.QuadPart = -10000 * 1000;
    KeDelayExecutionThread(KernelMode, TRUE, &large);
    PKLDR_DATA_TABLE_ENTRY32 ldr = (PKLDR_DATA_TABLE_ENTRY32)pDriverObj-
>DriverSection;
    pDriverObj->DriverSection = ldr->InLoadOrderLinks.Flink;
    RemoveEntryList(&ldr->InLoadOrderLinks);
    pDriverObj->DriverInit = 0;
    pDriverObj->Type = 0;
    pDriverObj->Size = 0;
}

PDRIVER_OBJECT FindDriverObject(wchar_t* Name)
```

```
{
    UNICODE_STRING DriverName = { 0 };
    PDRIVER_OBJECT pDriverObj = NULL;
    RtlInitUnicodeString(&DriverName, Name);
    NTSTATUS status = ObReferenceObjectByName(&DriverName, FILE_ALL_ACCESS, 0,
0, *IoDriverObjectType, KernelMode, 0, &pDriverObj);
    if (NT_SUCCESS(status))
    {
        ObDereferenceObject(pDriverObj);
        return pDriverObj;
    }
    else
    {
        DbgPrintEx(77, 0, "status = %d\r\n", status);
        return NULL;
    }
}


NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pSeg)
{
    DbgBreakPoint();
    PDRIVER_OBJECT pDriverObj =  FindDriverObject(L"\\Driver\\PCHunter32as");
    if (pDriverObj)
    {
       HANDLE hThread = NULL;
       NTSTATUS status = PsCreateSystemThread(&hThread, THREAD_ALL_ACCESS, 0, 0,
0, DirverHide, pDriverObj);
       if (NT_SUCCESS(status))
       {
           ZwClose(hThread);
       }
    }
    pDriver->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}
```

注意事项:

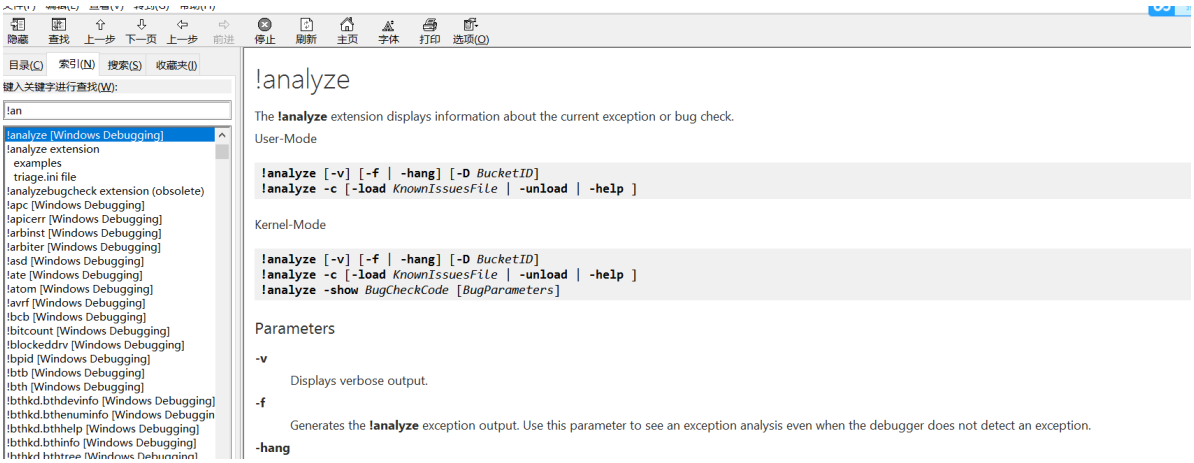
通过名字找寻`DriverObject`的时候`ObReferenceObjectByName`该函数是未导出的，`extern POBJECT_TYPE *IoDriverObjectType;`这句代码也要加上

要对自己的驱动进行隐藏，不能特别快因为还没有`loader`一起一些操作 会引发蓝屏
`KeDelayExecutionThread`该函数相当于`sleep`函数

测试的时候`pchunter`来说，他会对`DriverSection`进行一个读的操作，如果设置了`NULL`就会`crash`，所以要对该成员进行一个其他的赋值

蓝屏分析流程

主要也是根据`!analyze -v`输出的蓝屏错误码，从而定位蓝屏所在的问题



定位相关的问题后，再根据 `kv` 看看栈的问题来进一步判断

驱动通信

```
0: kd> dt _DRIVER_OBJECT
ntdll!_DRIVER_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT    //设备对象指针
   +0x008 Flags            : Uint4B
   +0x00c DriverStart      : Ptr32 Void
   +0x010 DriverSize       : Uint4B
   +0x014 DriverSection    : Ptr32 Void
   +0x018 DriverExtension  : Ptr32 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING
   +0x024 HardwareDatabase : Ptr32 _UNICODE_STRING
   +0x028 FastIoDispatch   : Ptr32 _FAST_IO_DISPATCH
   +0x02c DriverInit       : Ptr32     long
   +0x030 DriverStartIo    : Ptr32     void
   +0x034 DriverUnload     : Ptr32     void
   +0x038 MajorFunction    : [28] Ptr32     long
0: kd> dt _DEVICE_OBJECT
ntdll!_DEVICE_OBJECT
   +0x000 Type             : Int2B              //用它来表明该对象是一个设备对象，对设备对象来说，该成员的值为3， 它是一个只读成员
   +0x002 Size             : Uint2B             //表明设备对象的size (字节为单位)，该字节不包括设备扩展对象（如果有的话）。它是一个只读成员
   +0x004 ReferenceCount   : Int4B              //I/O管理器用它来追踪与该设备对象相关联的设备被打开的句柄数量
   +0x008 DriverObject     : Ptr32 _DRIVER_OBJECT   //驱动对象指针
   +0x00c NextDevice       : Ptr32 _DEVICE_OBJECT   //指向下一个由同一驱动程序创建的设备对象（如果有的话）
   +0x010 AttachedDevice   : Ptr32 _DEVICE_OBJECT   //挂载设备指针
   +0x014 CurrentIrp       : Ptr32 _IRP             //当前IRP
   +0x018 Timer            : Ptr32 _IO_TIMER        //计时器指针
   +0x01c Flags            : Uint4B
   +0x020 Characteristics  : Uint4B
   +0x024 Vpb              : Ptr32 _VPB
   +0x028 DeviceExtension  : Ptr32 Void             //拓展 一般为0
   +0x02c DeviceType       : Uint4B
   +0x030 StackSize        : Char                   //设备栈的大小
   +0x034 Queue            : <unnamed-tag>
   +0x05c AlignmentRequirement : Uint4B
   +0x060 DeviceQueue      : _KDEVICE_QUEUE
```

```
   +0x074 Dpc              : _KDPC                    //延迟处理调用
   +0x094 ActiveThreadCount : Uint4B
   +0x098 SecurityDescriptor : Ptr32 Void
   +0x09c DeviceLock       : _KEVENT
   +0x0ac SectorSize       : Uint2B
   +0x0ae Spare1           : Uint2Bc
   +0x0b0 DeviceObjectExtension : Ptr32 _DEVOBJ_EXTENSION
   +0x0b4 Reserved         : Ptr32 Void
```

什么是设备，比如我们的键盘和鼠标就是设备，那么设备是不是需要驱动，驱动是我们的代码层级的管理器而已，设备是需要被驱动来管理的（一个驱动对象可以管理多个设备）

```
ntdll!_IRP
   +0x000 Type             : Int2B
   +0x002 Size             : Uint2B
   +0x004 MdlAddress       : Ptr32 _MDL
   +0x008 Flags            : Uint4B
   +0x00c AssociatedIrp    : <unnamed-tag>
   +0x010 ThreadListEntry  : _LIST_ENTRY
   +0x018 IoStatus         : _IO_STATUS_BLOCK
   +0x020 RequestorMode    : Char
   +0x021 PendingReturned  : UChar
   +0x022 StackCount       : Char   //有多少设备
   +0x023 CurrentLocation  : Char   //当前设备栈的索引
   +0x024 Cancel           : UChar
   +0x025 CancelIrql       : UChar
   +0x026 ApcEnvironment   : Charc
   +0x027 AllocationFlags  : UChar
   +0x028 UserIosb         : Ptr32 _IO_STATUS_BLOCK
   +0x02c UserEvent        : Ptr32 _KEVENT
   +0x030 Overlay          : <unnamed-tag>
   +0x038 CancelRoutine    : Ptr32     void
   +0x03c UserBuffer       : Ptr32 Void
   +0x040 Tail             : <unnamed-tag>
```

设备对象拓展：

```
0: kd> dt _DEVOBJ_EXTENSION
ntdll!_DEVOBJ_EXTENSION
   +0x000 Type             : Int2B
   +0x002 Size             : Uint2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
   +0x008 PowerFlags       : Uint4B
   +0x00c Dope             : Ptr32 _DEVICE_OBJECT_POWER_EXTENSION
   +0x010 ExtensionFlags   : Uint4B
   +0x014 DeviceNode       : Ptr32 Void
   +0x018 AttachedTo       : Ptr32 _DEVICE_OBJECT
   +0x01c StartIoCount     : Int4B
   +0x020 StartIoKey       : Int4B
   +0x024 StartIoFlags     : Uint4Bc
   +0x028 Vpb              : Ptr32 _VPB
   +0x02c DependentList    : _LIST_ENTRY
   +0x034 ProviderList     : _LIST_ENTRY
```

R0层代码：

```
#include <ntifs.h>

#define DEVICE_NAME L"\\Device\\text"
#define DEVICE_SYM L"\\??\\text"

#define CODE_INDEX 0x800
#define TEXT
CTL_CODE(FILE_DEVICE_UNKNOWN,CODE_INDEX,METHOD_BUFFERED,FILE_ANY_ACCESS)

NTSTATUS
DefDispatch(
    _In_ struct _DEVICE_OBJECT* DeviceObject,
    _Inout_ struct _IRP* Irp
)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp, 0);
    return STATUS_SUCCESS;
}

NTSTATUS
Dispatch(
    _In_ struct _DEVICE_OBJECT* DeviceObject,
    _Inout_ struct _IRP* Irp
)
{
    DbgBreakPoint();
    //h
    PIO_STACK_LOCATION iostack = IoGetCurrentIrpStackLocation(Irp);
    if (iostack->MajorFunction == IRP_MJ_DEVICE_CONTROL)
    {
        int size = iostack->Parameters.DeviceIoControl.InputBufferLength;
        int OutputBufferLength = iostack-
>Parameters.DeviceIoControl.OutputBufferLength;
        ULONG IoControlCode = iostack->Parameters.DeviceIoControl.IoControlCode;

        switch (IoControlCode)
        {
            case TEXT:
            {
                int* x = (int *)Irp->AssociatedIrp.SystemBuffer;
                int y = 500;
                KdPrintEx((77, 0, "[db] = %d\r\n", *x));
                memcpy(Irp->AssociatedIrp.SystemBuffer, &y, 4);
                Irp->IoStatus.Information = OutputBufferLength;
            }
            break;
        }
    }
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp, 0);
    return STATUS_SUCCESS;
}

VOID
DriverUnload(
    _In_ struct _DRIVER_OBJECT* DriverObject
)
```

```
{
    UNICODE_STRING Symbol_Name = { 0 };
    RtlInitUnicodeString(&Symbol_Name, DEVICE_SYM);
    IoDeleteSymbolicLink(&Symbol_Name);
    IoDeleteDevice(DriverObject->DeviceObject);
}

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pSeg)
{
    UNICODE_STRING Device_Name = { 0 };
    RtlInitUnicodeString(&Device_Name, DEVICE_NAME);

    UNICODE_STRING Symbol_Name = { 0 };
    RtlInitUnicodeString(&Symbol_Name, DEVICE_SYM);

    PDEVICE_OBJECT pDevice = NULL;


    NTSTATUS status = IoCreateDevice(pDriver, 0, &Device_Name,
FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPEN, FALSE, &pDevice);
    if (!NT_SUCCESS(status))
    {
        KdPrintEx((77, 0, "[db] = %d\r\n", status));
        return status;
    }

    status = IoCreateSymbolicLink(&Symbol_Name, &Device_Name);
    if (!NT_SUCCESS(status))
    {
        IoDeleteDevice(pDevice);
        KdPrintEx((77, 0, "[db] = %d\r\n", status));
        return status;
    }

    pDevice->Flags &= ~DO_DEVICE_INITIALIZING;
    pDevice->Flags |= DO_BUFFERED_IO;

    pDriver->MajorFunction[IRP_MJ_CREATE] = DefDispatch;
    pDriver->MajorFunction[IRP_MJ_CLOSE] = DefDispatch;
    pDriver->MajorFunction[IRP_MJ_DEVICE_CONTROL] = Dispatch;

    pDriver->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}
```

R3层代码：

```
#include <Windows.h>
#include <winioctl.h>
#include <stdio.h>

#define DEVICE_SYM "\\\\.\\text"

#define CODE_INDEX 0x800
```

```
#define TEXT
CTL_CODE(FILE_DEVICE_UNKNOWN,CODE_INDEX,METHOD_BUFFERED,FILE_ANY_ACCESS)

int main()
{
    HANDLE hDevice = CreateFileA(DEVICE_SYM, GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
NULL);
    int x = 100;
    int y = 0;
    DWORD p = 0;
    BOOL b = DeviceIoControl(hDevice, TEXT, &x, 4, &y, 4, &p, NULL);
c
    CloseHandle(hDevice);
    printf("%d\r\n", y);
    system("pause");
    return 0;
}
```

由于DeviceIoControl很容易被检测 所以用Read也可以实现

```
#include <Windows.h>
#include <winioctl.h>
#include <stdio.h>

#define DEVICE_SYM "\\\\.\\text"

#define CODE_INDEX 0x800
#define TEXT
CTL_CODE(FILE_DEVICE_UNKNOWN,CODE_INDEX,METHOD_BUFFERED,FILE_ANY_ACCESS)

int main()
{
    HANDLE hDevice = CreateFileA(DEVICE_SYM, GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
NULL);
    int x = 100;
    DWORD p = 0;
    //BOOL b = DeviceIoControl(hDevice, TEXT, &x, 4, &y, 4, &p, NULL);
    ReadFile(hDevice, &x, 4, &p, NULL);

    CloseHandle(hDevice);
    printf("%d\r\n", x);
    system("pause");
    return 0;
}
```

```
#include <ntifs.h>

#define DEVICE_NAME L"\\Device\\text"
#define DEVICE_SYM L"\\??\\text"

#define CODE_INDEX 0x800
#define TEXT
CTL_CODE(FILE_DEVICE_UNKNOWN,CODE_INDEX,METHOD_BUFFERED,FILE_ANY_ACCESS)
```

```
NTSTATUS
ReadDispatch(
    _In_ struct _DEVICE_OBJECT* DeviceObject,
    _Inout_ struct _IRP* Irp
)
{
    DbgBreakPoint();
    PIO_STACK_LOCATION iostack = IoGetCurrentIrpStackLocation(Irp);
    int size = iostack->Parameters.Read.Length;
    LARGE_INTEGER ByteOffset = iostack->Parameters.Read.ByteOffset;
    int* xxx = Irp->AssociatedIrp.SystemBuffer;
    *xxx = 5000;
    Irp->IoStatus.Information = size;
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp, 0);
    return STATUS_SUCCESS;
}

NTSTATUS
DefDispatch(
    _In_ struct _DEVICE_OBJECT* DeviceObject,
    _Inout_ struct _IRP* Irp
)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp, 0);
    return STATUS_SUCCESS;
}

NTSTATUS
Dispatch(
    _In_ struct _DEVICE_OBJECT* DeviceObject,
    _Inout_ struct _IRP* Irp
)
{
    DbgBreakPoint();
    PIO_STACK_LOCATION iostack = IoGetCurrentIrpStackLocation(Irp);
    if (iostack->MajorFunction == IRP_MJ_DEVICE_CONTROL)
    {
        int size = iostack->Parameters.DeviceIoControl.InputBufferLength;
        int OutputBufferLength = iostack-
>Parameters.DeviceIoControl.OutputBufferLength;
        ULONG IoControlCode = iostack->Parameters.DeviceIoControl.IoControlCode;

        switch (IoControlCode)
        {
            case TEXT:
            {
                int* x = (int *)Irp->AssociatedIrp.SystemBuffer;
                int y = 500;
                KdPrintEx((77, 0, "[db] = %d\r\n", *x));
                memcpy(Irp->AssociatedIrp.SystemBuffer, &y, 4);
                Irp->IoStatus.Information = OutputBufferLength;
            }
            break;
        }
    }
```

```
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp, 0);
    return STATUS_SUCCESS;
}

VOID
DriverUnload(
    _In_ struct _DRIVER_OBJECT* DriverObject
)
{
    UNICODE_STRING Symbol_Name = { 0 };
    RtlInitUnicodeString(&Symbol_Name, DEVICE_SYM);
    IoDeleteSymbolicLink(&Symbol_Name);
    IoDeleteDevice(DriverObject->DeviceObject);
}

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pSeg)
{
    UNICODE_STRING Device_Name = { 0 };
    RtlInitUnicodeString(&Device_Name, DEVICE_NAME);

    UNICODE_STRING Symbol_Name = { 0 };
    RtlInitUnicodeString(&Symbol_Name, DEVICE_SYM);

    PDEVICE_OBJECT pDevice = NULL;


    NTSTATUS status = IoCreateDevice(pDriver, 0, &Device_Name,
FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPEN, FALSE, &pDevice);
    if (!NT_SUCCESS(status))
    {
        KdPrintEx((77, 0, "[db] = %d\r\n", status));
        return status;
    }

    status = IoCreateSymbolicLink(&Symbol_Name, &Device_Name);
    if (!NT_SUCCESS(status))
    {
        IoDeleteDevice(pDevice);
        KdPrintEx((77, 0, "[db] = %d\r\n", status));
        return status;
    }

    pDevice->Flags &= ~DO_DEVICE_INITIALIZING;
    pDevice->Flags |= DO_BUFFERED_IO;

    pDriver->MajorFunction[IRP_MJ_CREATE] = DefDispatch;
    pDriver->MajorFunction[IRP_MJ_CLOSE] = DefDispatch;
    pDriver->MajorFunction[IRP_MJ_DEVICE_CONTROL] = Dispatch;
    pDriver->MajorFunction[IRP_MJ_READ] = ReadDispatch;

    pDriver->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}
```

驱动封装

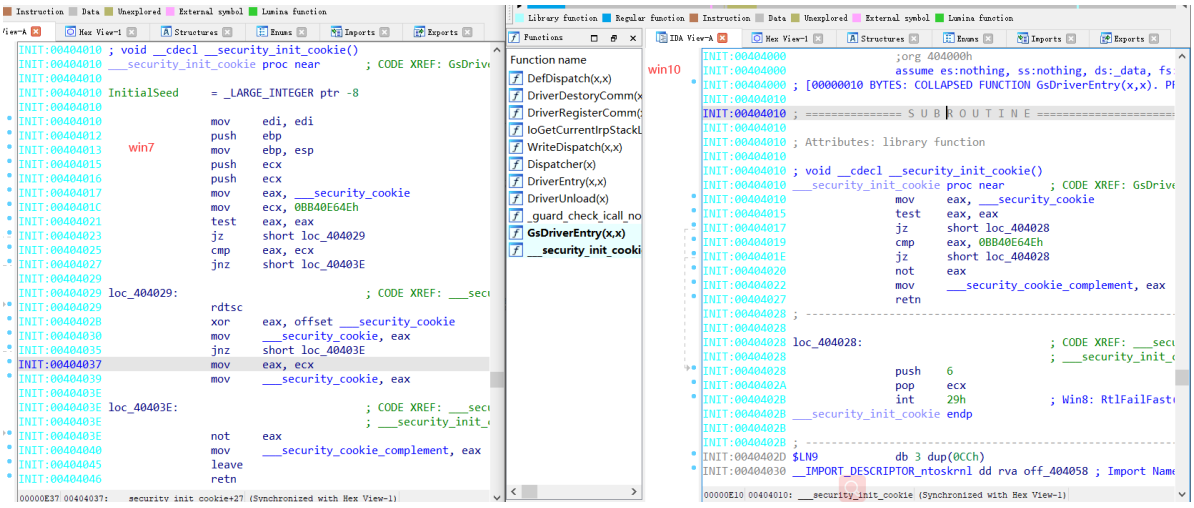
这里主要学习一个思想，就是对于驱动的封装，把要写的功能都封装起来，针对于main只留一个接口，便于扩展和使用

驱动内存加载 - 1

当我们双击exe的时候：

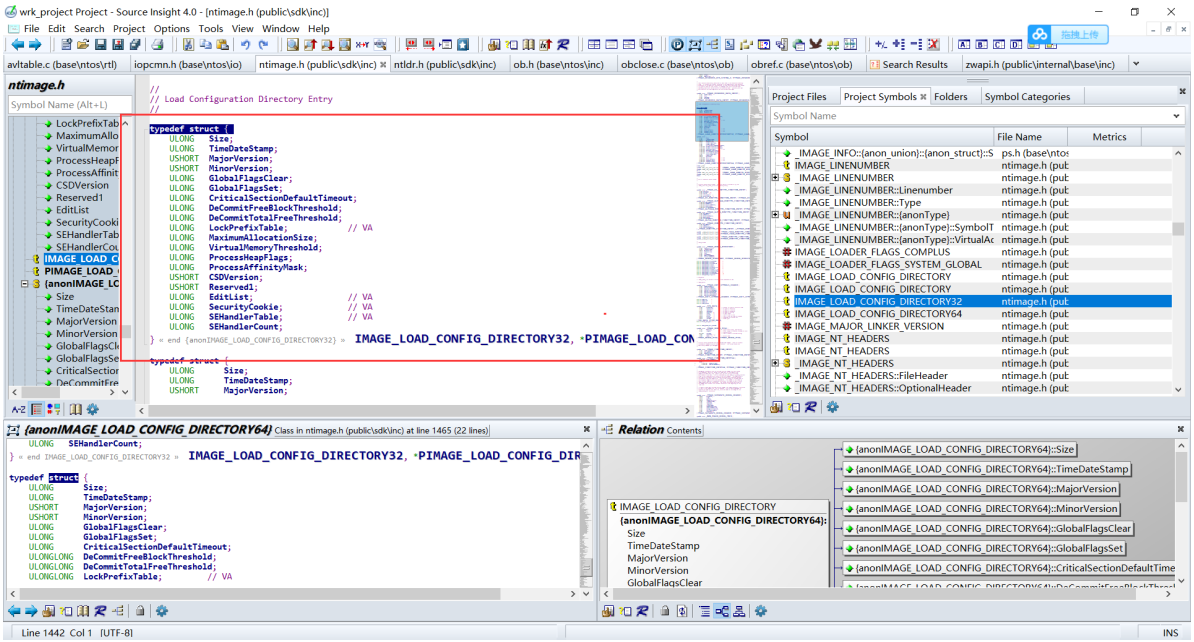
1. 会通过explorer.exe 去定位到文件
2. 得到这个文件后 去创建进程 CreateProcess 开辟一块空间（需要的地方挂上物理页（3环：PEB）（内核需要挂基址））
3. 通过文件路径载入exe（拉伸PE，解析PE头，把对应的数据目录，存放到内存相关偏移的地址）
4. 修复重定位 （有可能是随机基址/地址被占用）因为我们这里面的全局变量或者一些资源都是和imagebase进行一个关联的
5. 修复iat（导入表）
6. TLS（本地线程）（驱动里没有）
7. 延迟导入表（驱动里没有）
8. 修复异常（异常64位直接放在PE文件里面的所以要修复）修复 SEH异常
9. 获取入口点 CALL 入口点
10. 对于驱动来说还需要修复cookie

Windows10 与Windows7 之间cookie计算的区别：



如何修复cookie:

这个在PE代码结构里面



内存加载-2

我们不能通过文件的方式去载入，因为有文件就有痕迹，尽量避免这种，因为在磁盘上有个文件来说就代表我们的文件落地了，如果落地了就容易被查杀，所以这种文件的东西尽量不要被落地

主要的了解的知识还是要很了解相应的PE结构，因为对于一个filebuffer的来说，在内存需要延展成imagebuffer

主要的东西放在了代码中！可以看代码