



```

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
#ifdef MIDL_PASS
    [size_is(MaximumLength / 2), length_is((Length) / 2)] USHORT * Buffer;
#else // MIDL_PASS
    _Field_size_bytes_part_opt_(MaximumLength, Length) PWCH Buffer;
#endif // MIDL_PASS
} UNICODE_STRING;

```

z ANSI\_STRING或UNICODE\_STRING结构

的地址ANSI\_STRING或UNICODE\_STRING结构作为参数传递，显示缓冲区中包含的字符串。Buffer结构场。用大小修饰语前缀w若要指定UNICODE\_STRING争论--例如，%wZ。这个Length结构的字段必须设置为字符串的长度(以字节为单位)。这个MaximumLength结构的字段必须设置为缓冲区的长度(以字节为单位)。

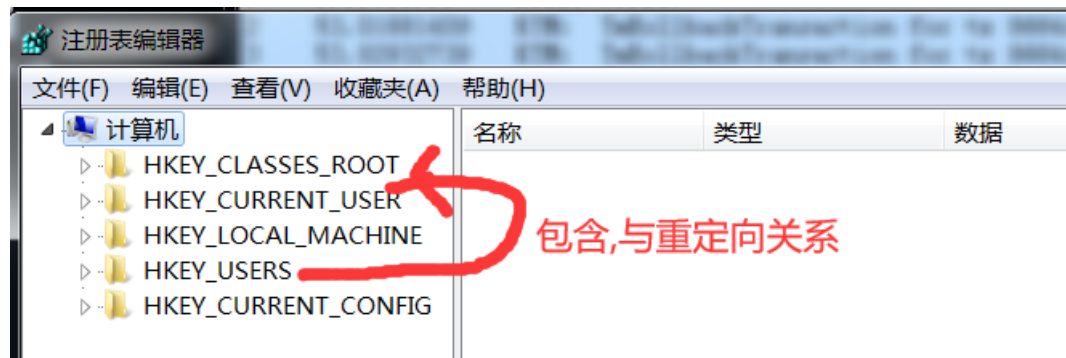
通常，z类型字符仅用于使用转换规范的驱动程序调试函数，如dbgPrint和kdPrint。

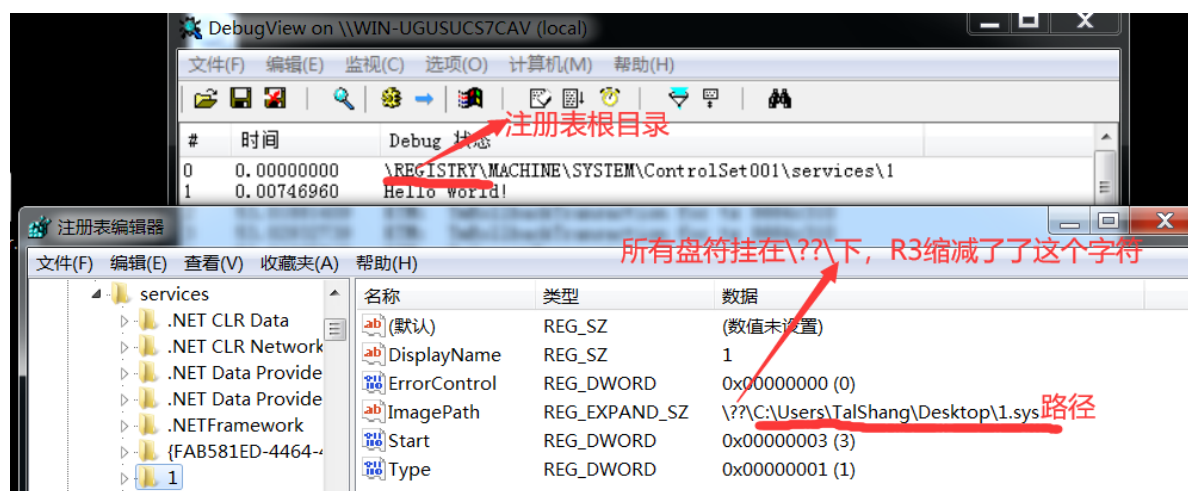
## 不引用参数

```
#define UNREFERENCED_PARAMETER(P) (P)
```

```
UNREFERENCED_PARAMETER(pDriver);
```

## PUNICODE\_STRING pReg





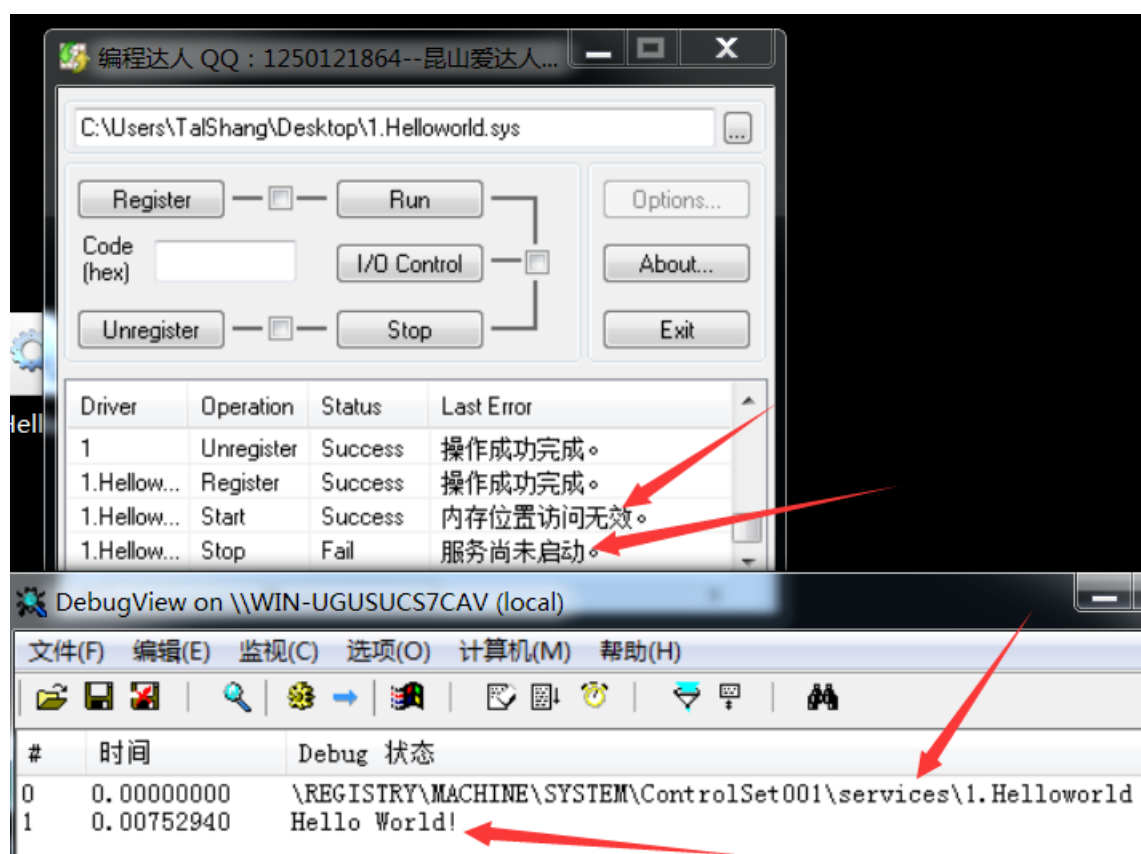
Start: 0 和 1 代表开机自启动, 3代表手动启动。可以写个 2 的开机自启。

Type: 代表当前是驱动还是服务。

## DriverEntry返回值

```
return STATUS_ACCESS_VIOLATION;
```

驱动运行后会自动卸载, 但仍会执行入口点函数, 不执行卸载函数。内存中不停留



## Register和Unregister的意义

就是操作注册表里的1子健。

# 驱动服务启动流程

注册服务 (注册到注册表)

运行服务 (加载到内存 停止服务 (内存中卸载 注销服务 (删注册表)

CreateService

StartService

system, Csrss等系统进程干活

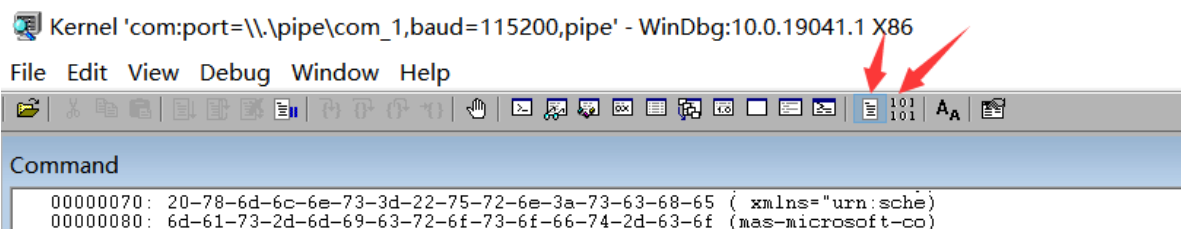
ZwLoadDriver 自己加载驱动  
前提是自己写好了注册表注册服务。  
对应ZwUnloadDriver 停止驱动  
注销服务仍需要自己在注册表清理。

## PDRIVER\_OBJECT

```
typedef struct _DRIVER_OBJECT {
    USHORT Type;
    USHORT Size;
    PDEVICE_OBJECT DeviceObject; //驱动程序创建的设备对象的指针
    ULONG Flags;
    PVOID DriverStart;
    ULONG DriverSize;
    PVOID DriverSection;
    PDRIVER_EXTENSION DriverExtension; //驱动程序的扩展指针(-->AddDevice)
    UNICODE_STRING DriverName;
    PUNICODE_STRING HardwareDatabase; //硬件配置信息, \Registry\Machine\Hardware中保存
    PFAST_IO_DISPATCH FastIoDispatch; //FSD和网络传输驱动程序使用, 指向快速I/O入口点
    PDRIVER_INITIALIZE DriverInit; //入口点
    PDRIVER_STARTIO DriverStartIo;
    PDRIVER_UNLOAD DriverUnload; //卸载函数
    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1]; //IRP
} DRIVER_OBJECT;
```

## 2.驱动调试

### 1.windbg源码调试



注意: windbg会通过pe文件格式找pdb路径, 然后通过真机的pdb加上符号。

### 2.LARGE\_INTEGER

```
typedef union _LARGE_INTEGER {
    struct {
        ULONG LowPart;
        LONG HighPart;
    } DUMMYSTRUCTNAME;
```

```

    struct {
        ULONG LowPart;
        LONG HighPart;
    } u;
    LONGLONG QuadPart;
} LARGE_INTEGER;
//驱动开发中，我们除了可以使用LONGLONG这个表示64位结构的数据外。
//还可以使用一个叫做LARGE_INTEGER的数据结构来表示64位数据。
LARGE_INTEGER value;
value.LowPart = 100;
value.HighPart = 0;
//在小端的情况下。低32位数字在前。高32位在后

```

### 3.创建线程与线程延时

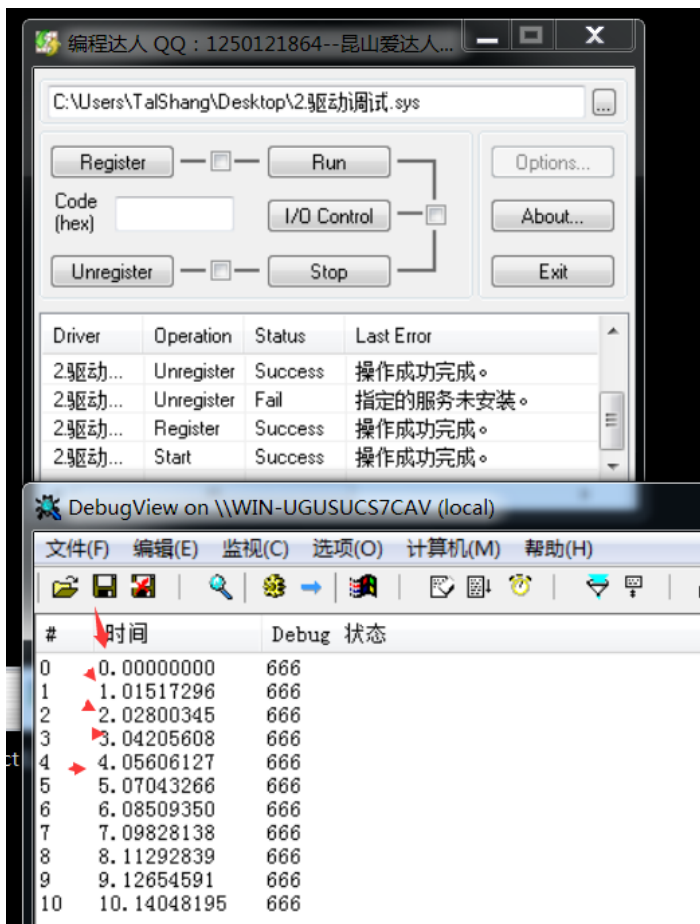
注意：创建的内核线程没有指定进程时，默认跑在System进程下。

没有退出线程就停止驱动会造成 蓝屏。

```

#include<ntifs.h>
VOID DRIVERUNLOAD(_In_ struct _DRIVER_OBJECT* DriverObject)
{
    DbgPrint("%z,%d,%d\r\n", __FUNCTION__, __LINE__, __TIME__);
}
VOID StartThread(_In_ PVOID StartContext)
{
    LARGE_INTEGER a = { 0 };
    a.QuadPart = -10000*1000; //按100纳秒的时间计算,也就是10^-4次方毫秒
    //表示相对时间要加负号,绝对时间要加+号。
    while (1)
    {
        KeDelayExecutionThread(KernelMode,FALSE,&a); //Alertable 是否允许被其他唤醒线程
        KdPrint(("666"));
    }
}
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pReg)
{
    pDriver->DriverUnload = DRIVERUNLOAD;
    HANDLE Hthread = 0;
    PsCreateSystemThread(&Hthread,THREAD_ALL_ACCESS,0,0,0, StartThread,0);
    return STATUS_SUCCESS;
}

```



## 4.蓝屏分析

```
kd>!an //加TAB建 自动补全
kd>!analyze -v
```

1.

```

*****
*
*                               Bugcheck Analysis                               *
*
*****
DRIVER_UNLOADED_WITHOUT_CANCELLING_PENDING_OPERATIONS (ce)
A driver unloaded without cancelling timers, DPCs, worker threads, etc.
The broken driver's name is displayed on the screen and saved in
KiBugCheckDriver.
Arguments:
Arg1: 96b3c093, memory referenced
Arg2: 00000008, value 0 = read operation, 1 = write operation
Arg3: 96b3c093, If non-zero, the instruction address which referenced the bad memory
      address.
Arg4: 00000000, Mm internal code.

Debugging Details:
-----

KEY_VALUES_STRING: 1

    Key  : Analysis.CPU.Sec
    Value: 6

    Key  : Analysis.DebugAnalysisProvider.CPP
    Value: Create: 8007007e on DESKTOP-HF7IEV9

    Key  : Analysis.DebugData
    Value: CreateObject

    Key  : Analysis.DebugModel
    Value: CreateObject

    Key  : Analysis.Elapsed.Sec
    Value: 87

    Key  : Analysis.Memory.CommitPeak.Mb
    Value: 128

```

蓝屏编号

错误描述

内存引用错误

保留

读  
执行

写

EIP

2.

Bug Check 0xCE

bug\_code

DRIVER\_UNLOADED\_WITHOUT\_CANCELLING\_PENDING\_OPERATIONS

3.

```

WRITE_ADDRESS: 96b3c093
IMAGE_NAME: 2.Çýµ÷ÊÏ.sys
MODULE_NAME: 2.Çýµ÷ÊÏ
FAULTING_MODULE: 00000000
PROCESS_NAME: System

TRAP_FRAME: 97967bd4 -- (.trap 0xffffffff97967bd4)
ErrCode = 00000010
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=8783e3f0 edi=00000000
eip=96b3c093 esp=97967c48 ebp=97967c50 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010246
<Unloaded_2.驱动调试.sys>+0x1093:
96b3c093 ??             ???
Resetting default scope

IP_MODULE_UNLOADED:
2.Çýµ÷ÊÏ.sys+1093
96b3c093 ??             ???

STACK_TEXT:
9796771c 83f20083 00000003 1e75b2ec 00000065 nt!RtlpBreakWithStatusInstruction
9796776c 83f20b81 00000003 8783e3f0 00000000 nt!KiBugCheckDebugBreak+0x1c
97967b30 83ecf41b 00000050 96b3c093 00000008 nt!KeBugCheck2+0x68b
97967bbc 83e823d8 00000008 96b3c093 00000000 nt!MmAccessFault+0x106
97967bbc 96b3c093 00000008 96b3c093 00000000 nt!KiTrap0E+0xdc
WARNING: Frame IP not in any known module. Following frames may be wrong.
97967c44 ff676980 ffffffff 97967c90 8404af5e <Unloaded_2.Çýµ÷ÊÏ.sys>+0x1093
97967c50 8404af5e 00000000 1e75b910 00000000 0xff676980
97967c90 83ef2219 96b3c060 00000000 00000000 nt!PspSystemThreadStartup+0x9e
00000000 00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x19

```

4.

Kd>K //查看调用堆栈

```

kd> k
# ChildEBP RetAddr
00 9796771c 83f20083 nt!RtlpBreakWithStatusInstruction
01 9796776c 83f20b81 nt!KiBugCheckDebugBreak+0x1c
02 97967b30 83ecf41b nt!KeBugCheck2+0x68b
03 97967bbc 83e823d8 nt!MmAccessFault+0x106
04 97967bbc 96b3c093 nt!KiTrap0E+0xdc
WARNING: Frame IP not in any known module. Following frames may be wrong.
05 97967c44 ff676980 <Unloaded_2.驱动调试.sys>+0x1093
06 97967c50 8404af5e 0xff676980
07 97967c90 83ef2219 nt!PspSystemThreadStartup+0x9e
08 00000000 00000000 nt!KiThreadStartup+0x19

```

Kd>KV //查看的堆栈更详细

```

kd> kv
# ChildEBP RetAddr Args to Child
00 9796771c 83f20083 00000003 1e75b2ec 00000065 nt!RtlpBreakWithStatusInstruction (FP0: [1,0,0])
01 9796776c 83f20b81 00000003 8783e3f0 00000000 nt!KiBugCheckDebugBreak+0x1c
02 97967b30 83ecf41b 00000050 96b3c093 00000008 nt!KeBugCheck2+0x68b
03 97967bbc 83e823d8 00000008 96b3c093 00000000 nt!MmAccessFault+0x106
04 97967bbc 96b3c093 00000008 96b3c093 00000000 nt!KiTrap0E+0xdc (FP0: [0,0] TrapFrame @ 97967bd4)
WARNING: Frame IP not in any known module. Following frames may be wrong.
05 97967c44 ff676980 ffffffff 97967c90 8404af5e <Unloaded_2.驱动调试.sys>+0x1093
06 97967c50 8404af5e 00000000 1e75b910 00000000 0xff676980
07 97967c90 83ef2219 96b3c060 00000000 00000000 nt!PspSystemThreadStartup+0x9e
08 00000000 00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x19

```

5. Trap Frame



```

kd> dt _KTRAP_FRAME 97967bd4
nt!_KTRAP_FRAME
+0x000 DbgEbp          : 0x97967c50
+0x004 DbgEip          : 0x96b3c093
+0x008 DbgArgMark      : 0xbadb0d00
+0x00c DbgArgPointer   : 0
+0x010 TempSegCs       : 0xe4b0
+0x012 Logging         : 0x83 ''
+0x013 Reserved       : 0x87 ''
+0x014 TempEsp         : 0x36
+0x018 Dr0             : 0x97967c48
+0x01c Dr1             : 0
+0x020 Dr2             : 0x8783e3f0
+0x024 Dr3             : 0
+0x028 Dr6             : 1
+0x02c Dr7             : 0
+0x030 SegGs           : 0
+0x034 SegEs           : 0x23
+0x038 SegDs           : 0x23
+0x03c Edx             : 0
+0x040 Ecx             : 0
+0x044 Eax             : 0
+0x048 PreviousPreviousMode : 0x83f0607f
+0x04c ExceptionList    : 0x97967c80 _EXCEPTION_REGISTRATION_RECORD
+0x050 SerrFo          : 0x30

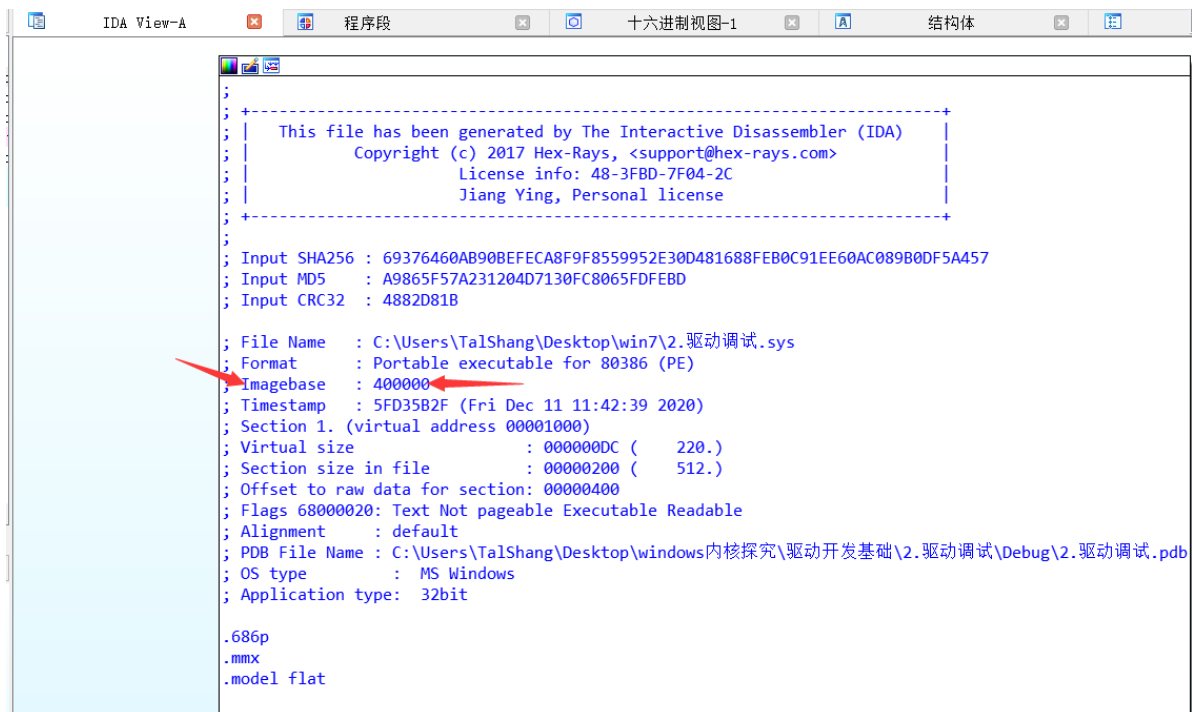
kd> .trap 97967bd4
ErrCode = 00000010
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=8783e3f0 edi=00000000
eip=96b3c093 esp=97967c48 ebp=97967c50 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010246
<Unloaded_2.驱动调试.sys>+0x1093:
96b3c093  ???

```

## 分析流程

通过 <驱动调试.sys>+0x1093，可以定位到是哪个驱动文件出错。

0x1093 是基于 ImageBase 的偏移。IDA 的默认 ImageBase 是 0x00400000。



所以，定位的代码地址为：401093

```

00401085 lea     edx, [ebp+a]
00401088 push    edx           ; Interval
00401089 push    0             ; Alertable
0040108B push    0             ; WaitMode
0040108D call    ds:__imp__KeDelayExecutionThread@12 ; KeDelayExecutionThread(x,x,x)
00401093 push    offset a666    ; "666"
00401098 call    _DbgPrint
0040109D add     esp, 4
004010A0 jmp     short loc_40107C

```

```

1 void __stdcall StartThread(void *StartContext)
2 {
3     _LARGE_INTEGER a; // [esp+0h] [ebp-8h]
4
5     a.QuadPart = -10000000i64;
6     while ( 1 )
7     {
8         KeDelayExecutionThread(0, 0, &a);
9         DbgPrint("666");
10    }
11}

```

## 无符号时In指令

当没有pdb符号时。

```

kd> kv
# ChildEBP RetAddr  Args to Child
00 9796771c 83f20083 00000003 1e75b2ec 00000065 nt!DbgBreakWithStatusInstruction (F7F7 [1,3,5])
01 9796776c 83f20b81 00000003 8783e3f0 00000000 nt!DbgCheckDebugBreak (F7F7 [1,3,5])
02 97967b30 83ecf41b 00000050 96b3c093 00000008 nt!KDBG_Sig3Lock+0x8b
03 97967bbc 83e823d8 00000008 96b3c093 00000000 nt!MmAccessFault+0x106

```

Kd>ln 返回地址 //解析函数 //windbg自动解析 正确率一半。

```

kd> ln 83ecf41b
Browse module
Set by breakpoint
(83ecf315) nt!MmAccessFault+0x106 | (83ed22f7) nt!CcCanIWrite

```

## 当蓝屏错误在系统函数地址

如果，windbg上没有暴出驱动相关的函数调用的痕迹，而是爆了一个系统函数的错误。

可能是，栈溢出，内核线程堆栈x86只有12k,x64有24k。

可能是，给与的内存地址错误，例如空指针。

可能是，结构体未初始化好。

注意：系统函数出错，跟了一个0x0E号异常，就是堆栈出错。

## 5.断点调试

```

DbgBreakPoint();
#define DbgBreakPoint __debugbreak

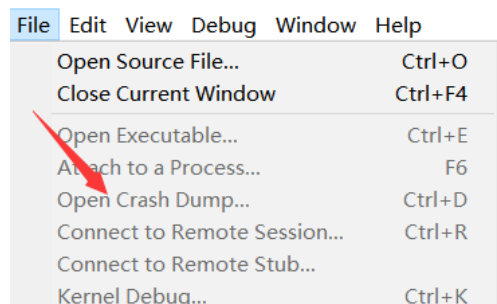
```

## 6.蓝屏dump

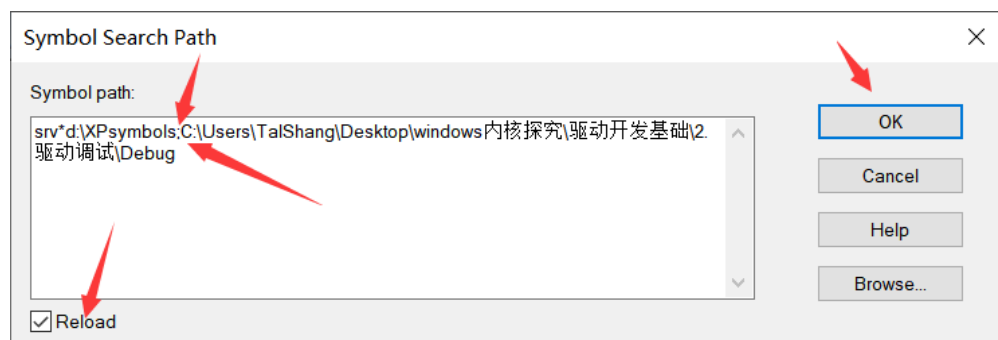
注意：自动定位不一定成功，先上基本符号，添加dump文件，再加自己的符号,reload;

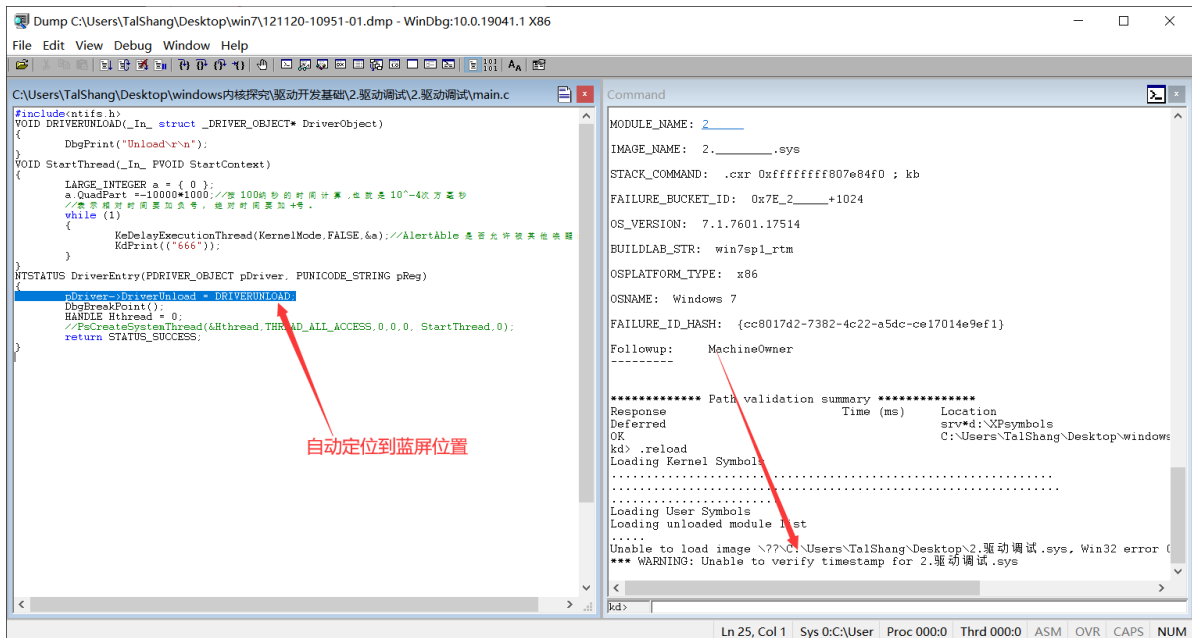


Dump C:\Users\TalShang\Desktop\win7\12112



## 给dump加pdb路径





### 3.驱动断链

#### 1.windows内核结构



图 2.3 Windows 内核的组成结构

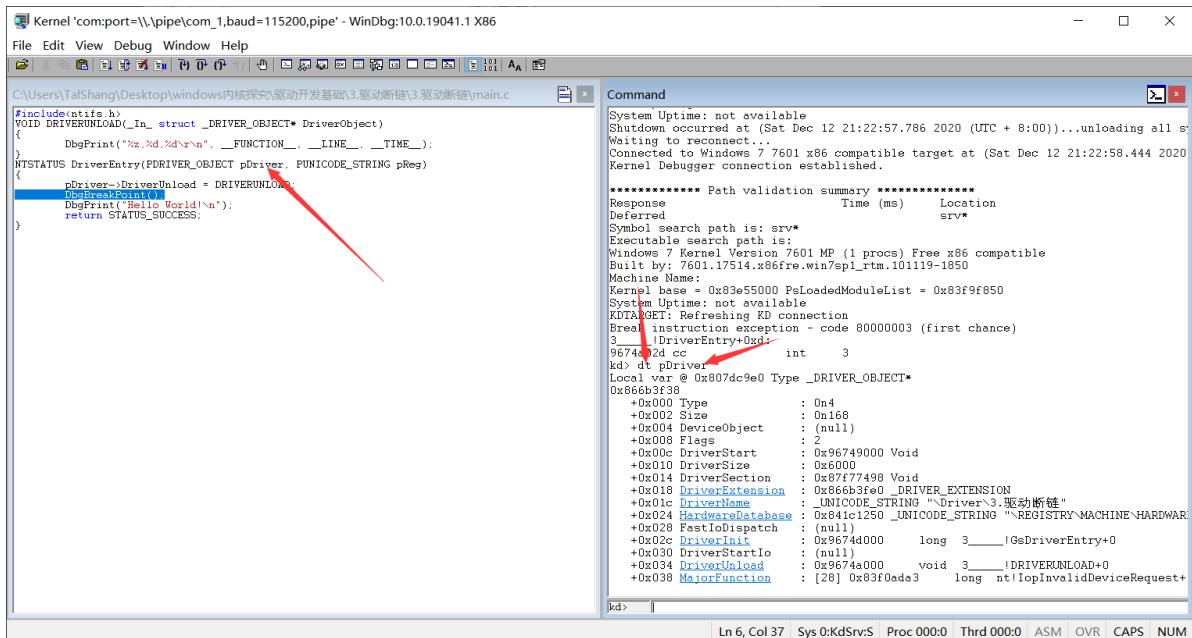
例程(Routine): 从OpenProcess到ZwOpenProcess通过SSDT表到NtOpenProcess, 这样完整的一条线, 叫做例程。

#### 2.宏内核与微内核

EPROCESS, 是执行体层, 和三环交流。(E开头都是

KPROCESS, 是微内核或内核层, 不和三环交流。(K开头都是

#### 3.再次解析DRIVER\_OBJECT



```
kd> dt pDriver
Local var @ 0x807dc9e0 Type _DRIVER_OBJECT*
0x87dde838
+0x000 Type          : 0n4//这里解析错误，应该是4，代表驱动类型。
+0x002 Size          : 0n168//168，代表这个结构体的大小。
+0x004 DeviceObject  : (null) //设备对象
+0x008 Flags         : 2
+0x00c DriverStart   : 0x9519e000 Void//驱动加载后的起始地址，ImageBase。
+0x010 DriverSize    : 0x6000//内存中，驱动的大小。
+0x014 DriverSection : 0x867a70b0 Void //_KLDLDR_，类似TEB中的_LDR。
+0x018 DriverExtension : 0x87dde8e0 _DRIVER_EXTENSION//驱动扩展结构
+0x01c DriverName     : _UNICODE_STRING "\Driver\3.驱动断链"//驱动名。
+0x024 HardwareDatabase : 0x841b2250 _UNICODE_STRING
"\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"//数据库的目录。和pReg差不多，但是要简
略的多。
+0x028 FastIoDispatch : (null) //驱动通信，不经过IRP
+0x02c DriverInit     : 0x951a2000 long 3_____!GsDriverEntry+0//驱动
DriverEntry函数，是驱动入口点。
+0x030 DriverStartIo  : (null)
+0x034 DriverUnload   : 0x9519f000 void 3_____!DRIVERUNLOAD+0
+0x038 MajorFunction  : [28] 0x83efbda3 long
nt!IopInvalidDeviceRequest+0//IRP驱动通信。

kd> dt _DRIVER_EXTENSION 0x87dde8e0
ntdll!_DRIVER_EXTENSION
+0x000 DriverObject   : 0x87dde838 _DRIVER_OBJECT //指向驱动对象。（可以获取驱动
对象地址。
+0x004 AddDevice      : (null)
+0x008 Count          : 0
+0x00c ServiceKeyName : _UNICODE_STRING "3.驱动断链"//服务名。
+0x014 ClientDriverExtension : (null)
+0x018 FsFilterCallbacks : (null) //minifilter 文件过滤
```

## 4.解析DriverSection

```
typedef struct _NON_PAGED_DEBUG_INFO {
    USHORT      Signature;
    USHORT      Flags;
    ULONG       Size;
    USHORT      Machine;
    USHORT      Characteristics;
    ULONG       TimeDateStamp;
    ULONG       CheckSum;
    ULONG       SizeOfImage;
    ULONGLONG   ImageBase;
} NON_PAGED_DEBUG_INFO, * PNON_PAGED_DEBUG_INFO;
typedef struct _KLDLDR_DATA_TABLE_ENTRY {
    LIST_ENTRY  InLoadOrderLinks;
    PVOID       ExceptionTable;
    ULONG       ExceptionTableSize;
    // ULONG padding on IA64
    PVOID       GpValue;
    PNON_PAGED_DEBUG_INFO NonPagedDebugInfo;
    PVOID       DllBase;
    PVOID       EntryPoint;
    ULONG       SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG       Flags;
    USHORT      LoadCount;
    USHORT      __Unused5;
    PVOID       SectionPointer;
    ULONG       CheckSum;
    // ULONG padding on IA64
    PVOID       LoadedImports;
    PVOID       PatchInformation;
} KLDLDR_DATA_TABLE_ENTRY, * PKLDLDR_DATA_TABLE_ENTRY;
```

```
kd> dt _KLDLDR_DATA_TABLE_ENTRY 0x867a70b0
```

```
0!e32!_KLDLDR_DATA_TABLE_ENTRY
```

```
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x83f90850 - 0x872f5eb0 ]//双向链表
+0x008 ExceptionTable   : 0xffffffff Void
+0x00c ExceptionTableSize : 0xffffffff
+0x010 GpValue          : 0x0000001c Void
+0x014 NonPagedDebugInfo : (null)
+0x018 DllBase           : 0x9519e000 Void//DriverStart
+0x01c EntryPoint        : 0x951a2000 Void//驱动DriverEntry函数,是驱动入口点。
```

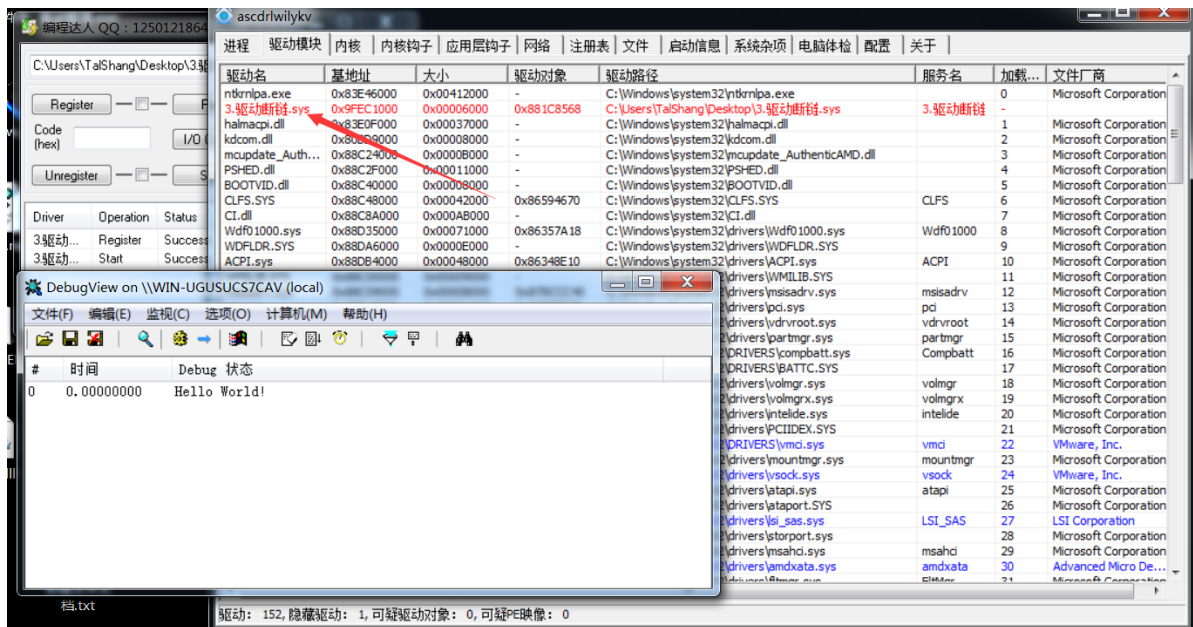
```
DriverInit
```

```
+0x020 SizeOfImage       : 0x6000//DriverSize
+0x024 FullDllName       : _UNICODE_STRING "C:\Users\Ta1Shang\Desktop\3.驱动断链.sys"
+0x02c BaseDllName       : _UNICODE_STRING "3.驱动断链.sys"
+0x034 Flags              : 0x49104000//权限
+0x038 LoadCount         : 1//驱动加载次数。
+0x03a __Unused5         : 0x69
+0x03c SectionPointer    : (null)
+0x040 CheckSum           : 0x10e16//校验和,来自PE头。
+0x044 CoverageSectionSize : 0x5c0073
+0x048 CoverageSection   : (null)
+0x04c LoadedImports      : 0x86341c99 Void
```

```
+0x050 PatchInformation : (null)
+0x054 SizeOfImageNotRounded : 0x6000
+0x058 TimeDateStamp      : 0x5fd77db8
```

## 断链

```
#include<ntifs.h>
typedef struct _NON_PAGED_DEBUG_INFO {
    USHORT      Signature;
    USHORT      Flags;
    ULONG       Size;
    USHORT      Machine;
    USHORT      Characteristics;
    ULONG       TimeDateStamp;
    ULONG       CheckSum;
    ULONG       SizeOfImage;
    ULONGLONG   ImageBase;
} NON_PAGED_DEBUG_INFO, * PNON_PAGED_DEBUG_INFO;
typedef struct _KLDR_DATA_TABLE_ENTRY {
    LIST_ENTRY  InLoadOrderLinks;
    PVOID       ExceptionTable;
    ULONG       ExceptionTableSize;
    // ULONG padding on IA64
    PVOID       GpValue;
    PNON_PAGED_DEBUG_INFO NonPagedDebugInfo;
    PVOID       DllBase;
    PVOID       EntryPoint;
    ULONG       SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG       Flags;
    USHORT      LoadCount;
    USHORT      __Unused5;
    PVOID       SectionPointer;
    ULONG       CheckSum;
    // ULONG padding on IA64
    PVOID       LoadedImports;
    PVOID       PatchInformation;
} KLDR_DATA_TABLE_ENTRY, * PKLDR_DATA_TABLE_ENTRY;
VOID DRIVERUNLOAD(_In_ struct _DRIVER_OBJECT* DriverObject)
{
    DbgPrint("%Z,%d,%d\r\n", __FUNCTION__, __LINE__, __TIME__);
}
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pReg)
{
    pDriver->DriverUnload = DRIVERUNLOAD;
    RemoveEntryList(pDriver->DriverSection); //库函数，无导入表。
    DbgPrint("Hello world!\n");
    return STATUS_SUCCESS;
}
```



## 遍历

```
#include<ntifs.h>

typedef struct _NON_PAGED_DEBUG_INFO {
    USHORT      Signature;
    USHORT      Flags;
    ULONG       Size;
    USHORT      Machine;
    USHORT      Characteristics;
    ULONG       TimeDateStamp;
    ULONG       CheckSum;
    ULONG       SizeOfImage;
    ULONGLONG   ImageBase;
} NON_PAGED_DEBUG_INFO, * PNON_PAGED_DEBUG_INFO;

typedef struct _KLDL_DATA_TABLE_ENTRY {
    LIST_ENTRY InLoadOrderLinks;
    PVOID ExceptionTable;
    ULONG ExceptionTableSize;
    // ULONG padding on IA64
    PVOID GpValue;
    PNON_PAGED_DEBUG_INFO NonPagedDebugInfo;
    PVOID DllBase;
    PVOID EntryPoint;
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BasedDllName;
    ULONG Flags;
    USHORT LoadCount;
    USHORT __Unused5;
    PVOID SectionPointer;
    ULONG CheckSum;
    // ULONG padding on IA64
    PVOID LoadedImports;
    PVOID PatchInformation;
} KLDL_DATA_TABLE_ENTRY, * PKLDL_DATA_TABLE_ENTRY;

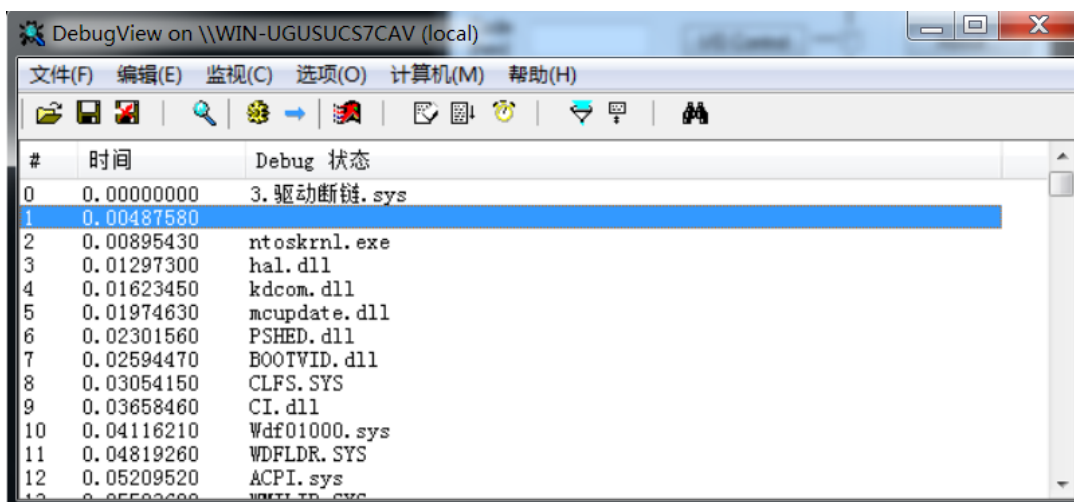
VOID DRIVERUNLOAD(_In_ struct _DRIVER_OBJECT* DriverObject)
{
    DbgPrint("%z,%d,%d\\r\\n", __FUNCTION__, __LINE__, __TIME__);
}
```



```

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pReg)
{
    pDriver->DriverUnload = DRIVERUNLOAD;
    PKLDR_DATA_TABLE_ENTRY CurrentHeadr=(PKLDR_DATA_TABLE_ENTRY)pDriver->DriverSection;
    PKLDR_DATA_TABLE_ENTRY FilnkHeadr= ((PKLDR_DATA_TABLE_ENTRY)pDriver->DriverSection)->InLoadOrderLinks.Flink;
    KdPrint(("wZ\n", &CurrentHeadr->BaseDllName));//注意wZ输出UNICODE_STRING时，要加上取地址。PUNICODE不用。
    while (CurrentHeadr != FilnkHeadr)
    {
        KdPrint(("wZ\n",&FlnkHeadr->BaseDllName));//注意wZ输出UNICODE_STRING时，要加上取地址。PUNICODE不用。
        FilnkHeadr = FilnkHeadr->InLoadOrderLinks.Flink;
    }
    return STATUS_SUCCESS;
}

```



## 完整断链

```

#include<ntifs.h>
typedef struct _NON_PAGED_DEBUG_INFO {
    USHORT      Signature;
    USHORT      Flags;
    ULONG       Size;
    USHORT      Machine;
    USHORT      Characteristics;
    ULONG       TimeDateStamp;
    ULONG       CheckSum;
    ULONG       SizeOfImage;
    ULONGLONG   ImageBase;
} NON_PAGED_DEBUG_INFO, * PNON_PAGED_DEBUG_INFO;
typedef struct _KLDR_DATA_TABLE_ENTRY {
    LIST_ENTRY InLoadOrderLinks;
    PVOID ExceptionTable;
    ULONG ExceptionTableSize;
    // ULONG padding on IA64
    PVOID GpValue;
    PNON_PAGED_DEBUG_INFO NonPagedDebugInfo;
    PVOID DllBase;
    PVOID EntryPoint;
    ULONG SizeOfImage;
}

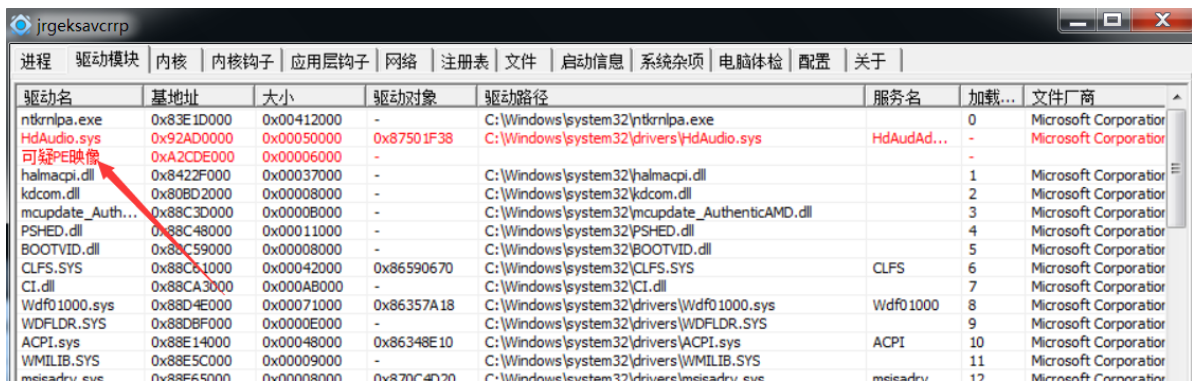
```

```

    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG Flags;
    USHORT LoadCount;
    USHORT __Unused5;
    PVOID SectionPointer;
    ULONG CheckSum;
    // ULONG padding on IA64
    PVOID LoadedImports;
    PVOID PatchInformation;
} KLDLDR_DATA_TABLE_ENTRY, * PKLDLDR_DATA_TABLE_ENTRY;
VOID DRIVERUNLOAD(_In_ struct _DRIVER_OBJECT* DriverObject)
{
    DbgPrint("%Z,%d,%d\\r\\n", __FUNCTION__, __LINE__, __TIME__);
}
VOID start(_In_ PVOID StartContext)
{
    LARGE_INTEGER time= { 0 };
    time.QuadPart = -10000 * 1000; //延时一秒，先让DriverEntry跑完。
    KeDelayExecutionThread(KernelMode, FALSE, &time);
    PDRIVER_OBJECT pDriver = StartContext;
    RemoveEntryList(pDriver->DriverSection); //断链
    pDriver->Type = 0; //抹类型
    pDriver->Size = 0; //抹掉结构体大小
    pDriver->DriverSection = 0; //抹掉地址
}
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pReg)
{
    pDriver->DriverUnload = DRIVERUNLOAD;
    HANDLE hthread = 0;
    PsCreateSystemThread(&hthread, THREAD_ALL_ACCESS, 0, 0, 0, start, pDriver); //不在
    DriverEntry中断链，是因为DriverEntry执行完后会返回，然后执行其他。
    return STATUS_SUCCESS;
}

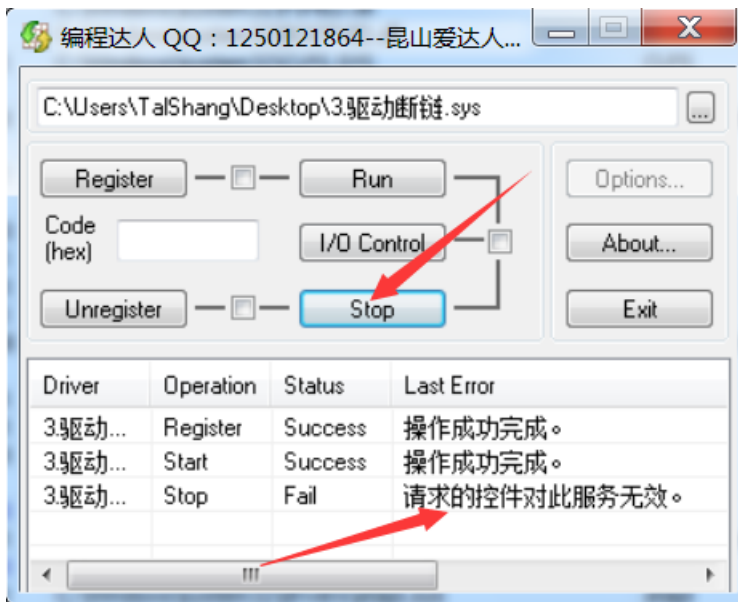
```

注意：新版本Pchunter仍能遍历出驱动，可见断链没啥用。



驱动名	基址	大小	驱动对象	驱动路径	服务名	加载...	文件厂商
ntkrnlpa.exe	0x83E1D000	0x00412000	-	C:\Windows\system32\ntkrnlpa.exe		0	Microsoft Corporation
HdAudio.sys	0x92AD0000	0x00050000	0x87501F38	C:\Windows\system32\drivers\HdAudio.sys	HdAudAd...	-	Microsoft Corporation
可疑PE映像	0xA2CDE000	0x00006000	-			-	
halmacpi.dll	0x8422F000	0x00037000	-	C:\Windows\system32\halmacpi.dll		1	Microsoft Corporation
kdcom.dll	0x80BD2000	0x00008000	-	C:\Windows\system32\kdcom.dll		2	Microsoft Corporation
mcupdate_Auth...	0x88C3D000	0x00008000	-	C:\Windows\system32\mcupdate_AuthenticAMD.dll		3	Microsoft Corporation
PSHED.dll	0x88C48000	0x00011000	-	C:\Windows\system32\PSHED.dll		4	Microsoft Corporation
BOOTVID.dll	0x88C59000	0x00008000	-	C:\Windows\system32\BOOTVID.dll		5	Microsoft Corporation
CLFS.SYS	0x88C61000	0x00042000	0x86590670	C:\Windows\system32\CLFS.SYS	CLFS	6	Microsoft Corporation
CI.dll	0x88CA3000	0x000AB000	-	C:\Windows\system32\CI.dll		7	Microsoft Corporation
Wdf01000.sys	0x88D4E000	0x00071000	0x86357A18	C:\Windows\system32\drivers\Wdf01000.sys	Wdf01000	8	Microsoft Corporation
WDFLDR.SYS	0x88DBF000	0x0000E000	-	C:\Windows\system32\drivers\WDFLDR.SYS		9	Microsoft Corporation
ACPI.sys	0x88E14000	0x00048000	0x86348E10	C:\Windows\system32\drivers\ACPI.sys	ACPI	10	Microsoft Corporation
WMILIB.SYS	0x88E5C000	0x00009000	-	C:\Windows\system32\drivers\WMILIB.SYS		11	Microsoft Corporation
meicadrv.exe	0x88F55000	0x00008000	0x870C4D70	C:\Windows\system32\drivers\meicadrv.exe	meicadrv	12	Microsoft Corporation

注意：断链后的驱动不能被正常暂停和卸载。



## 字符串比较细节

```
UNICODE_STRING DriverName;
RtlInitUnicodeString(&DriverName, L"要比较的字符串");
if(RtlCompareUnicodeString(&UNICODE_STRING, &DriverName) == 0)
{
}
//本质是Strcmp, 但是Strcmp容易蓝屏。
//同理, memset, memcpy等C语言库函数, 都容易蓝屏, 建议使用Rtl系列代替。
```

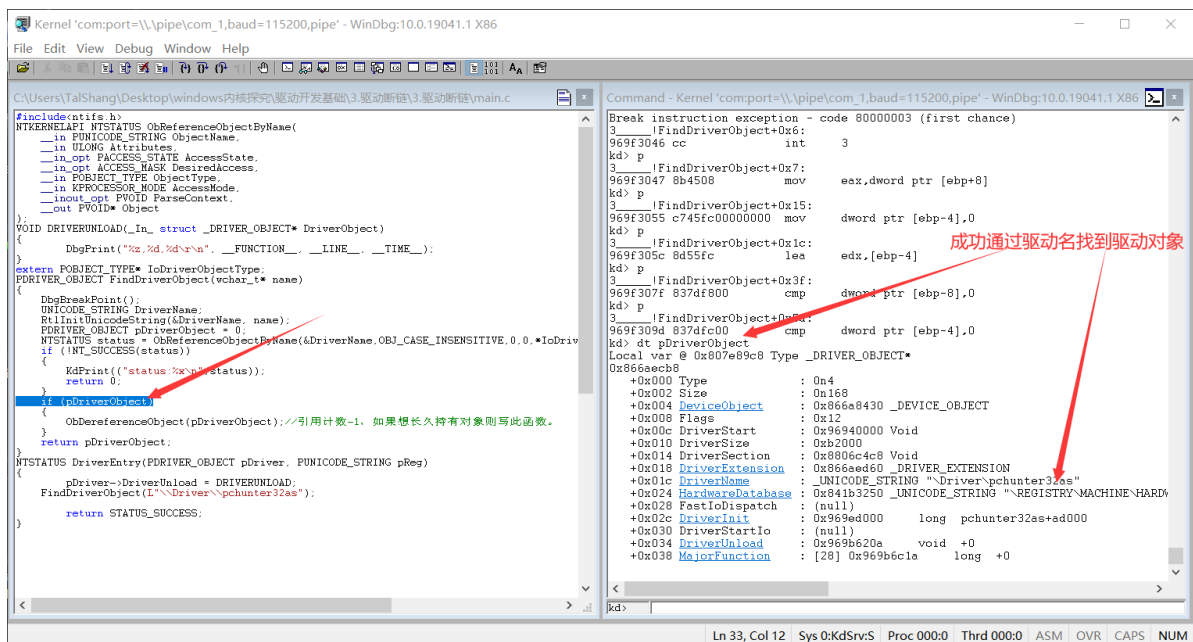
## 通过驱动名找驱动对象

```
#include<ntifs.h>
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
);
VOID DRIVERUNLOAD(_In_ struct _DRIVER_OBJECT* DriverObject)
{
    DbgPrint("%Z,%d,%d\r\n", __FUNCTION__, __LINE__, __TIME__);
}
extern POBJECT_TYPE* IoDriverObjectType; //全局变量。
PDRIVER_OBJECT FindDriverObject(wchar_t* name)
{
    UNICODE_STRING DriverName;
    RtlInitUnicodeString(&DriverName, name);
    PDRIVER_OBJECT pDriverObject = 0;
    NTSTATUS status =
    ObReferenceObjectByName(&DriverName, OBJ_CASE_INSENSITIVE, 0, 0, *IoDriverObjectType,
    , KernelMode, 0, &pDriverObject); //未文档化函数。
    if (!NT_SUCCESS(status))
    {
    }
```

```

    KdPrint(("status:%x\n",status));
    return 0;
}
if (pDriverObject)
{
    ObDereferenceObject(pDriverObject); //引用计数-1, 如果想长久持有对象则不加此函数。
}
return pDriverObject;
}
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pReg)
{
    pDriver->DriverUnload = DRIVERUNLOAD;
    FindDriverObject(L"\\Driver\\pchunter32as"); //注意: 没有.sys后缀
    return STATUS_SUCCESS;
}

```



## 导出未文档化函数

定义：IDA里查看导出表能找到的函数，但是在<ntifs.h>的头文件里找不到定义的函数。

名称	地址	序号
ObReferenceObjectByHandle	005FC45C	1198
ObReferenceObjectByHandleWithTag	005FC486	1199
ObReferenceObjectByName	0061DF8D	1200
ObReferenceObjectByPointer	004A60B9	1201
ObReferenceObjectByPointerWithTag	004755C0	1202
ObReferenceSecurityDescriptor	00666A3E	1203
ObRegisterCallbacks	006CB949	1204
ObReleaseObjectSecurity	00657C2E	1205

```

UNICODE_STRING DriverName;
RtlInitUnicodeString(&DriverName, name);
NTSTATUS status=ObReferenceObjectByName()

```

没找到

```

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
);
//WRK中定义。

```

## IoDriverObjectType

```
extern POBJECT_TYPE* IoDriverObjectType; //全局变量。
```

3. 驱动断链.sys

模块名称	导入	OFTs	时间日期戳	转发链	名称
00000CF4	N/A	00000C48	00000C4C	00000C50	00000C54
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword
ntoskrnl.exe	5	00004070	00000000	00000000	00000000

导入表里不止有函数，还有变量

OFTs	FTs (IAT)	Hint	Name
Dword	Dword	Word	szAnsi
000040A0	000040A0	005D	DbgPrint
000040AC	000040AC	0646	ObfDereferenceObject
000040C4	000040C4	0638	ObReferenceObjectByName
000040DE	000040DE	02DD	IoDriverObjectType
00004088	00004088	084B	RtlInitUnicodeString

## PE: INIT段

INIT：顾名思义，初始化段，在初始化完成后，内存被释放。

DriverEntry函数在INIT段中。

名称	虚拟大小	虚拟地址	Raw 大小	Raw 地址	重定位地址
00000238	00000240	00000244	00000248	0000024C	00000250
Byte[8]	Dword	Dword	Dword	Dword	Dword
.text	0000011E	00001000	00000200	00000400	00000000
.rdata	00000284	00002000	00000400	00000600	00000000
.data	00000008	00003000	00000200	00000A00	00000000
INIT	00000102	00004000	00000200	00000C00	00000000
.reloc	00000044	00005000	00000200	00000E00	00000000

3.驱动断链.sys				
成员	偏移量	大小	值	内涵
Magic	000000E0	Word	010B	PE32
MajorLinkerVersion	000000E2	Byte	0E	
MinorLinkerVersion	000000E3	Byte	1B	
SizeOfCode	000000E4	Dword	00000400	
SizeOfInitializedData	000000E8	Dword	00000800	
SizeOfUninitializedData	000000EC	Dword	00000000	
AddressOfEntryPoint	000000F0	Dword	00004000	INIT

## 隐藏

通过给系统驱动的INIT段线性地址挂上物理页，写shellcode。绕过常规检测。

寻找init段地址-->MdlMapMemory-->memcpy-->MdlUnMapMemory-->替换对象钩子

# 4.驱动通信

## 1.ioCreateDevice

一个驱动管理多个设备对象，设备对象与R3通信。

```

NTKERNELAPI
NTSTATUS
IoCreateDevice(
    _In_   PDRIVER_OBJECT DriverObject, //驱动对象
    _In_   ULONG DeviceExtensionSize, //设备扩展对象的大小。一般为0
    _In_opt_ PUNICODE_STRING DeviceName, //设备对象名 注意:"\\Device\\xxx"
    _In_   DEVICE_TYPE DeviceType, //设备类型，例如鼠标，键盘，耳机等，一般为
    FILE_DEVICE_UNKNOWN (0x22
    _In_   ULONG DeviceCharacteristics, //驱动地附加信息，大多数是
    FILE_DEVICE_SECURE_OPEN
    _In_   BOOLEAN Exclusive, //独占，如果为True，则R3一次性只能打开一个设备对象的句柄，大多数是 FALSE
                                //设为True的好处是 检测打不开就放过去了。
    _Outptr_result_nullonfailure_
    _At_(*DeviceObject,
        __drv_allocatesMem(Mem)
        _when((((_In_function_class_(DRIVER_INITIALIZE))
                ||(_In_function_class_(DRIVER_DISPATCH))))),
        __drv_aliasesMem))
    PDEVICE_OBJECT *DeviceObject //返回一个设备对象。注意是二级指针。
);

```

```

kd> dt _DEVICE_OBJECT
ntdll!_DEVICE_OBJECT
+0x000 Type           : Int2B
+0x002 Size           : Uint2B
+0x004 ReferenceCount : Int4B //打开次数 猜测可能和FALSE 和TRUE有关
+0x008 DriverObject   : Ptr32 _DRIVER_OBJECT //驱动对象
+0x00c NextDevice     : Ptr32 _DEVICE_OBJECT
+0x010 AttachedDevice : Ptr32 _DEVICE_OBJECT

```

```

+0x014 CurrentIrp      : Ptr32 _IRP //当前IRP
+0x018 Timer          : Ptr32 _IO_TIMER
+0x01c Flags           : Uint4B
+0x020 Characteristics : Uint4B
+0x024 Vpb            : Ptr32 _VPB
+0x028 DeviceExtension : Ptr32 Void
+0x02c DeviceType      : Uint4B
+0x030 StackSize       : Char
+0x034 Queue           : <unnamed-tag>
+0x05c AlignmentRequirement : Uint4B
+0x060 DeviceQueue     : _KDEVICE_QUEUE
+0x074 Dpc              : _KDPC
+0x094 ActiveThreadCount : Uint4B
+0x098 SecurityDescriptor : Ptr32 Void
+0x09c DeviceLock       : _KEVENT
+0x0ac SectorSize       : Uint2B
+0x0ae Spare1           : Uint2B
+0x0b0 DeviceObjectExtension : Ptr32 _DEVOBJ_EXTENSION
+0x0b4 Reserved         : Ptr32 Void

```

## 2. IoCreateSymbolicLink

```

UNICODE_STRING SymbolLink;
RtlInitUnicodeString(&SymbolLink, L"\\?\\SybmolLink");
sta=IoCreateSymbolicLink(&SymbolLink, &Device_1);
if (!NT_SUCCESS(sta))
{
    IoDeleteDevice(pDevice); //设备对象挂符号链接失败后删除设备对象。
    KdPrint(("SymbolicLink Create Fail!\n"));
    return STATUS_SYMLINK_CLASS_DISABLED;
}

```

## 3. IRP

```

#define IRP_MJ_CREATE          0x00 //R3打开设备对象    //通信需要
#define IRP_MJ_CREATE_NAMED_PIPE 0x01 //R3打开命名管道
#define IRP_MJ_CLOSE          0x02 //R3关闭设备对象句柄  //通信需要
#define IRP_MJ_READ            0x03 //R3的读
#define IRP_MJ_WRITE           0x04 //R3的写
#define IRP_MJ_DEVICE_CONTROL 0x0e //主要分发函数。 //通信需要

```

```

kd> dt _IRP -r1
ntdll!_IRP
+0x000 Type          : Int2B
+0x002 Size          : Uint2B
+0x004 MdlAddress     : Ptr32 _MDL //映射内存
    +0x000 Next       : Ptr32 _MDL
    +0x004 Size       : Int2B
    +0x006 MdlFlags   : Int2B
    +0x008 Process    : Ptr32 _EPROCESS
    +0x00c MappedSystemVa : Ptr32 Void
    +0x010 StartVa    : Ptr32 Void
    +0x014 ByteCount  : Uint4B
    +0x018 ByteOffset : Uint4B

```



```

+0x008 Flags : Uint4B
+0x00c AssociatedIrp : <unnamed-tag> //IRP的参数
    +0x000 MasterIrp : Ptr32 _IRP
    +0x000 IrpCount : Int4B
    +0x000 SystemBuffer : Ptr32 Void //系统缓冲区，这里是R3传过来的参数。
+0x010 ThreadListEntry : _LIST_ENTRY
    +0x000 Flink : Ptr32 _LIST_ENTRY
    +0x004 Blink : Ptr32 _LIST_ENTRY
+0x018 IoStatus : _IO_STATUS_BLOCK //状态
    +0x000 Status : Int4B
    +0x000 Pointer : Ptr32 Void
    +0x004 Information : Uint4B //返回给R3的数据的字节数
+0x020 RequestorMode : Char //请求模式，区别R3和R0来的请求。1代表R3，0代表R0
+0x021 PendingReturned : UChar //等待一下再返回
+0x022 StackCount : Char //栈的个数，因为每个过滤FDO都需要一个栈，所以在这里，栈
的个数对应FDO的个数。
//如果有3个设备栈，这里填 2
+0x023 CurrentLocation : Char
+0x024 Cancel : UChar
+0x025 CancelIrql : UChar //取消等级
+0x026 ApcEnvironment : Char //Apc环境
+0x027 AllocationFlags : UChar
+0x028 UserIosb : Ptr32 _IO_STATUS_BLOCK //没有用到，在<windows驱动开发技术
详解>中有过介绍。
    +0x000 Status : Int4B
    +0x000 Pointer : Ptr32 Void
    +0x004 Information : Uint4B
+0x02c UserEvent : Ptr32 _KEVENT //用户事件，只是异步IO才需要使用。异步的例子：
详见于ReadFileEx
    +0x000 Header : _DISPATCHER_HEADER
+0x030 overlay : <unnamed-tag>
    +0x000 AsynchronousParameters : <unnamed-tag>
    +0x000 AllocationSize : _LARGE_INTEGER
+0x038 CancelRoutine : Ptr32 void
+0x03c UserBuffer : Ptr32 Void //用户缓冲区，直写IO
+0x040 Tail : <unnamed-tag>
    +0x000 overlay : <unnamed-tag> //这里放
IoGetCurrentIrpStackLocation 的设备栈
    +0x000 Apc : _KAPC
    +0x000 CompletionKey : Ptr32 Void

```

```

kd> dt _IRP
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 //IO状态，默认是0 (STATUS_SUCCESS)
+0x020 RequestorMode : Char
+0x021 PendingReturned : UChar
+0x022 StackCount : Char //栈的个数
+0x023 CurrentLocation : Char
+0x024 Cancel : UChar
+0x025 CancelIrql : UChar
+0x026 ApcEnvironment : Char

```



```

+0x027 AllocationFlags : UChar
+0x028 UserIosb        : Ptr32 _IO_STATUS_BLOCK
+0x02c UserEvent       : Ptr32 _KEVENT
+0x030 Overlay         : <unnamed-tag>
+0x038 CancelRoutine   : Ptr32 void //取消函数
+0x03c UserBuffer      : Ptr32 Void
+0x040 Tail            : <unnamed-tag>

```

## 4.设备栈与控制码

### 控制码

```

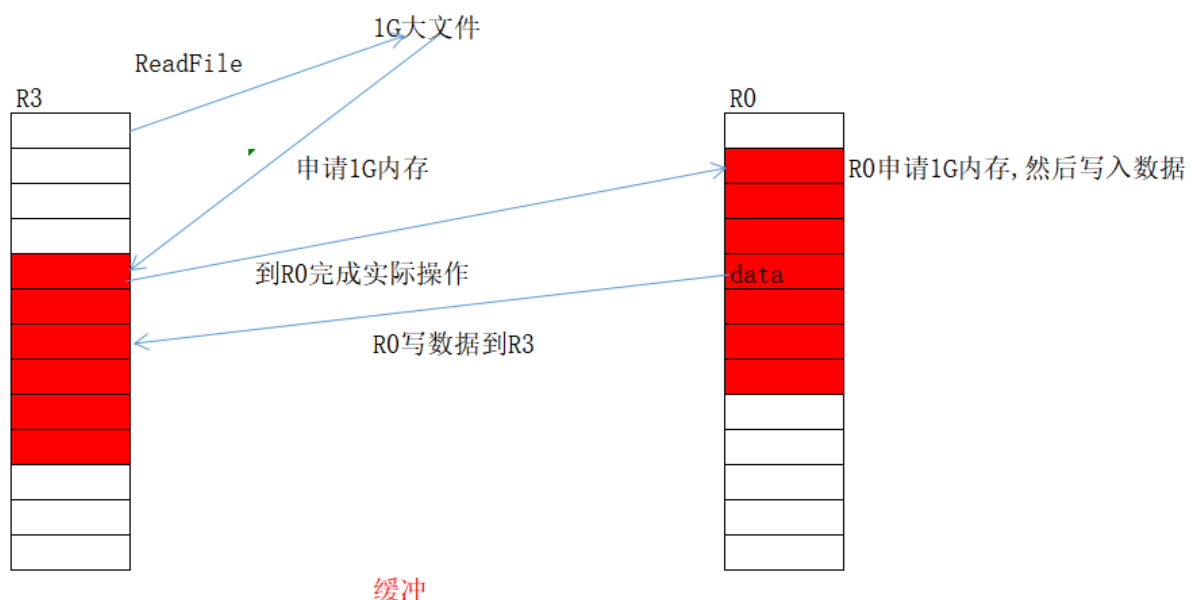
#define METHOD_BUFFERED          0 //缓冲
#define METHOD_IN_DIRECT        1 //直写
#define METHOD_OUT_DIRECT       2 //直写
#define METHOD_NEITHER          3 //基本不用

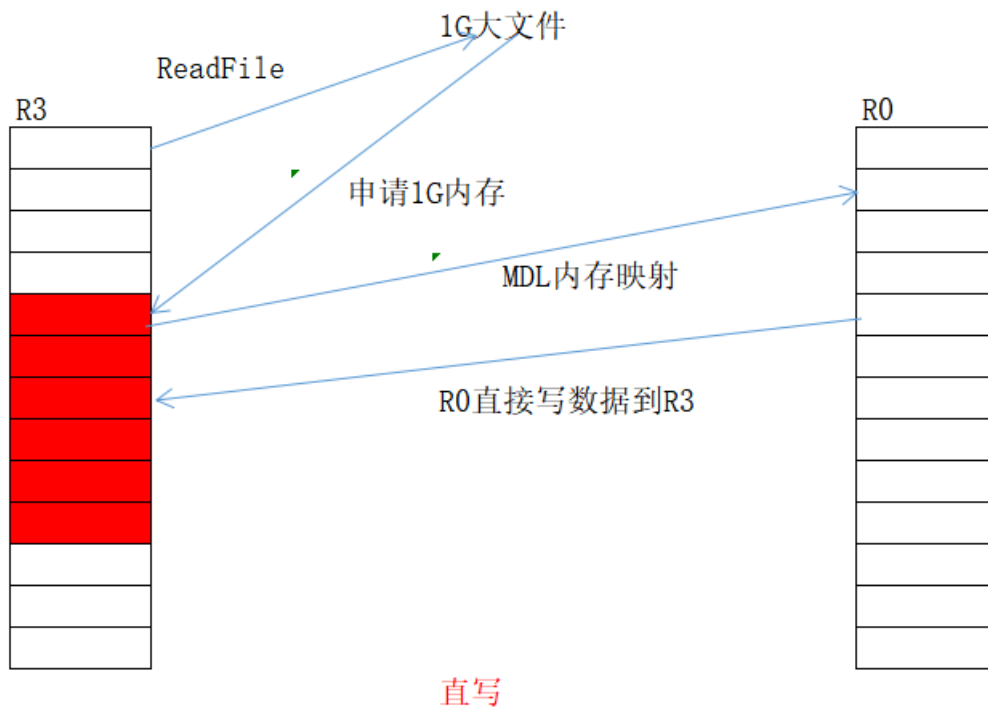
#define CODE_INDEX 0x800
#define CT_TEST
CTL_CODE(FILE_DEVICE_UNKNOWN, CODE_INDEX, METHOD_BUFFERED, FILE_ANY_ACCESS)

#define CTL_CODE( DeviceType, Function, Method, Access ) ( \
    ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method) \
)
//DeviceType 设备类型 FILE_DEVICE_UNKNOWN(需要匹配Device_Object)
//Function 消息号
//Method 写入的方式
//Access 访问权限

```

### 直写和缓冲





## 5.完整代码

### Driver

```
#include<ntifs.h>
VOID DRIVERUNLOAD(_In_ struct _DRIVER_OBJECT* DriverObject)
{
    UNICODE_STRING SymbolLink;
    RtlInitUnicodeString(&SymbolLink, L"\\??\\sybmolLink");
    IoDeleteSymbolicLink(&SymbolLink);//删除符号链接 重要
    IoDeleteDevice(DriverObject->DeviceObject);//删除设备对象 重要
}
NTSTATUS Default_Main(_In_ struct _DEVICE_OBJECT* DeviceObject,_Inout_ struct
_IRP* Irp)//默认处理函数
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    DbgPrint("成功打开/关闭设备");
    IoCompleteRequest(Irp,0);//完成请求，只代表函数完成，不代表消息处理完了。
    return STATUS_SUCCESS;
}

#define CODE_INDEX 0x800
#define CT_TEST
CTL_CODE(FILE_DEVICE_UNKNOWN, CODE_INDEX, METHOD_BUFFERED, FILE_ANY_ACCESS)//初始化控
制码
NTSTATUS Control_Main(
    _In_ struct _DEVICE_OBJECT* DeviceObject,
    _Inout_ struct _IRP* Irp)
{
    PIO_STACK_LOCATION iostack;
    iostack=IoGetCurrentIrpStackLocation(Irp);//取Irp的设备栈
    ULONG IoControlCode=iostack->Parameters.DeviceIoControl.IoControlCode;//取消息
码
    ULONG length_In = iostack->Parameters.DeviceIoControl.InputBufferLength;//取
输入缓冲区的长度
    ULONG length_Out = iostack->Parameters.DeviceIoControl.OutputBufferLength;//
取输出缓冲区的长度
```

```

PVOID inParam = Irp->AssociatedIrp.SystemBuffer;//取R3传过来的缓冲区
switch (IoControlCode)//默认四个字节，所以不对长度进行判断。
{
case CT_TEST:{
    KdPrint((" %x\r\n",*(PULONG)inParam));//拿R3的参数
    *(PULONG)inParam = 0x30;//写回
    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = 4;//返回的消息的长度 一定要返回
}
}
IoCompleteRequest(Irp, 0);
return STATUS_SUCCESS;
}
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pReg)
{
    pDriver->DriverUnload = DRIVERUNLOAD;
    UNICODE_STRING Device_1;
    RtlInitUnicodeString(&Device_1, L"\\Device\\Device_1");
    PDEVICE_OBJECT pDevice = 0;
    NTSTATUS sta=IoCreateDevice(pDriver,0,&Device_1, //创建设备对象
        FILE_DEVICE_UNKNOWN,
        FILE_DEVICE_SECURE_OPEN,
        FALSE,
        &pDevice);
    if (!NT_SUCCESS(sta))
    {
        KdPrint(("Device Create Fail!\n"));
        return STATUS_DEVICE_NOT_READY;
    }

    UNICODE_STRING SymbolLink;
    RtlInitUnicodeString(&SymbolLink, L"\\??\\sybmolLink");
    sta=IoCreateSymbolicLink(&SymbolLink, &Device_1);//挂上符号链接
    if (!NT_SUCCESS(sta))
    {
        IoDeleteDevice(pDevice);//设备对象挂符号链接失败后删除设备对象。
        KdPrint(("SymbolicLink Create Fail!\n"));
        return STATUS_SYMLINK_CLASS_DISABLED;
    }

    pDriver->MajorFunction[IRP_MJ_CREATE] = Default_Main;
    pDriver->MajorFunction[IRP_MJ_CLOSE] = Default_Main;
    pDriver->MajorFunction[IRP_MJ_DEVICE_CONTROL] = Control_Main;
    pDriver->Flags |= DO_BUFFERED_IO;//挂上通信的Method
    return STATUS_SUCCESS;
}

```

## Exe

注意：2019最新版本 debug模式下不能开MT

```

#include<windows.h>
#include<iostream>
using namespace std;
#define CODE_INDEX 0x800
#define CT_TEST
CTL_CODE(FILE_DEVICE_UNKNOWN, CODE_INDEX, METHOD_BUFFERED, FILE_ANY_ACCESS)

```

```

// #define Symbolic_link L"\\??\\SybmolLink"//试用于win7以上版本
#define Symbolic_link L"\\\\.\\SybmolLink"
int main()
{
    HANDLE h_Device=CreateFile(Symbolic_link,
        GENERIC_READ|GENERIC_WRITE,
        0,
        0,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        0);
    if (!h_Device)
    {
        cout << "打开失败! " << endl;
    }
    DWORD inbuf = 0;
    DWORD outbuf = 0;
    DWORD ret_Byte = 0;
    BOOL is=0;
    if (h_Device)
    {
        is = DeviceIoControl(h_Device, CT_TEST, &inbuf, 4, &outbuf, 4,
&ret_Byte, 0);
    }
    if (is)
    {
        cout << outbuf << endl;
    }
    system("pause");
    closehandle(h_Device);
    return 0;
}

```

## 6.IRP派遣函数<windows驱动开发技术详解>

### IRP的处理

大部分IRP来自 文件处理的WIN32 API，如CreateFile，ReadFile。

处理这些IRP最简单的方法是：

- 1.在派遣函数中，将IRP的状态 设置为成功。
- 2.用IoCompleteRequest结束IRP请求。
- 3.派遣函数返回成功。

```

NTSTATUS Default_Main(_In_ struct _DEVICE_OBJECT* DeviceObject, _Inout_ struct
_IRP* Irp)
{
    Irp->IoStatus.status=STATUS_SUCCESS;//返回给发起请求的R3的API，当作返回值。
    //GetLastError 得到的一致。
    Irp->IoStatus.Information=0; //IRP操作了多少字节，如果是ReadFile就是对某设备读了多少
字节。WriteFile就是写了多少字节
    IoCompleteRequest(Irp, 0);//完成请求。
    return STATUS_SUCCESS;
}

```

IRP的处理流程

ReadFile->Ntdll.NtReadFile->系统调用的SSDT中的NtReadFile->创建IRP\_MJ\_WRITE类型的IRP发给某个驱动的派遣函数。----->线程进行等待--->派遣函数CompleteRequest，唤醒线程。

```
NTKERNELAPI
VOID
FASTCALL
IoCompleteRequest(
    _In_ PIRP Irp,
    _In_ CCHAR PriorityBoost
);
```

表 7-2 完成优先级

优先级	说 明
IO_NO_INCREMENT	不增加优先级
IO_CD_ROM_INCREMENT	光驱设备增加的优先级
IO_DISK_INCREMENT	磁盘设备增加的优先级
IO_KEYBOARD_INCREMENT	键盘设备增加的优先级
IO_MOUSE_INCREMENT	鼠标设备增加的优先级
IO_NAMED_PIPE_INCREMENT	命名管道增加的优先级
IO_NETWORK_INCREMENT	网络设备增加的优先级
IO_PARALLEL_INCREMENT	并口设备增加的优先级
IO_SERIAL_INCREMENT	串口设备增加的优先级
IO_SOUND_INCREMENT	声卡设备增加的优先级
IO_VIDEO_INCREMENT	视频设备增加的优先级
SEMAPHORE_INCREMENT	信号灯增加的优先级

设备栈

定义：驱动对象会创建一个个的设备对象，并折叠成一个垂直结构，这就是设备栈。

IRP请求被操作系统发送到设备栈的顶层，如果顶层的设备对象没有CompleteRequest，则发送到下一层。

一个IRP可能被多次转发，为了记录IRP在每层的操作，IRP会有一个IO\_STACK\_LOCATION数组，元素数大于IRP穿过的设备数。

对于本层设备对象的IO\_STACK\_LOCATION，可通过IoGetCurrentIrpStackLocation()获取。

```
PIO_STACK_LOCATION stack=IoGetCurrentIrpStackLocation(pIrp)。
```

DeviceIoControl

```
DeviceIoControl(
    _In_ HANDLE hDevice, //已经打开的设备对象
    _In_ DWORD dwIoControlCode, //控制码, Ioctl
    _In_reads_bytes_opt_(nInBufferSize) LPVOID lpInBuffer, //输入缓冲区
    _In_ DWORD nInBufferSize, //输入缓冲区的大小
    _Out_writes_bytes_to_opt_(nOutBufferSize, *lpBytesReturned) LPVOID
lpOutBuffer, //输出缓冲区
    _In_ DWORD nOutBufferSize, //输出缓冲区大小
    _Out_opt_ LPDWORD lpBytesReturned, //实际返回字节数, pIrp->IoStatus->Information
    _Inout_opt_ LPOVERLAPPED lpOverlapped //是否overlapped
);
```

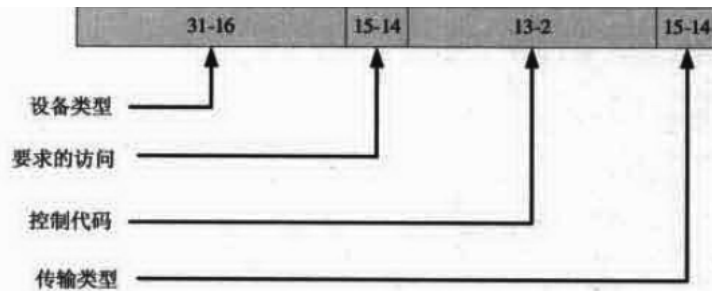


图 7-15 IOCTL 定义

DDK 特意提供了一个宏 CTL\_CODE，其定义如下：

```
CTL_CODE( DeviceType, Function, Method, Access )
```

- DeviceType: 设备对象的类型，这个类型应和创建设备（IoCreateDevice）时的类型相匹配。一般是形如 FILE\_DEVICE\_XX 的宏。
- Function: 这是驱动程序定义的 IOCTL 码。其中：
  - 0X0000 到 0X7FFF: 为微软保留。
  - 0X800 到 0XFFF: 由程序员自己定义。
- Method: 这个是操作模式，可以是下列四种模式之一。
  - METHOD\_BUFFERED: 使用缓冲区方式操作。
  - METHOD\_IN\_DIRECT: 使用直接写方式操作。
  - METHOD\_OUT\_DIRECT: 使用直接读方式操作。
  - METHOD\_NEITHER: 使用其他方式操作。
- Access: 访问权限，如果没有特殊要求，一般使用 FILE\_ANY\_ACCESS。

## 7.驱动通信改

```
#include<ntifs.h>
#define CODE_INDEX 0x800
#define CT_TEST
CTL_CODE(FILE_DEVICE_UNKNOWN, CODE_INDEX, METHOD_BUFFERED, FILE_ANY_ACCESS)
typedef struct
{
    ULONG TYPE;
    ULONG RESULT;
    ULONG64 DATA;
    ULONG64 SIZE;
}DATA, * PDATA;
VOID DRIVERUNLOAD(_In_ struct _DRIVER_OBJECT* DriverObject)
{
    IoDeleteDevice(DriverObject->DeviceObject);
    UNICODE_STRING SYMBOL_NAME = { 0 };
    RtlInitUnicodeString(&SYMBOL_NAME, L"\\??\\FuckSymbol");
    NTSTATUS sta=IoDeleteSymbolicLink(&SYMBOL_NAME);
    if (!NT_SUCCESS(sta))
    {
        DbgPrint("删除符号链接失败");
    }
}

NTSTATUS DEFAULT_DISPATCH(
```

```

_In_ struct _DEVICE_OBJECT* DeviceObject,
_Inout_ struct _IRP* Irp)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    DbgPrint("成功打开/关闭设备");
    IoCompleteRequest(Irp,0);
    return STATUS_SUCCESS;
}

NTSTATUS Main_DISPATCH(
    _In_ struct _DEVICE_OBJECT* DeviceObject,
    _Inout_ struct _IRP* Irp)
{
    PIO_STACK_LOCATION pIrp=IoGetCurrentIrpStackLocation(Irp);
    ULONG Control_Code=pIrp->Parameters.DeviceIoControl.IoControlCode;
    ULONG len_In=pIrp->Parameters.DeviceIoControl.InputBufferLength;
    ULONG len_Out=pIrp->Parameters.DeviceIoControl.OutputBufferLength;
    PDATA pdata=(PDATA)Irp->AssociatedIrp.SystemBuffer;
    switch (Control_Code)
    {
    case CT_TEST:
    {
        DbgPrint("链接成功\n");
        DbgPrint("PDATA.DATA=%d\n",pdata->DATA);
        pdata->DATA = 2;
        Irp->IoStatus.Status = STATUS_SUCCESS;
        Irp->IoStatus.Information = sizeof(DATA);//不填不会覆盖缓冲区
    }
    }
    IoCompleteRequest(Irp, 0);
    return STATUS_SUCCESS;
}

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING preg)
{
    pDriver->DriverUnload = DRIVERUNLOAD;

    PDEVICE_OBJECT Device_Object = 0;
    UNICODE_STRING Device_Name = {0};
    RtlInitUnicodeString(&Device_Name, L"\\Device\\FuckDevice");
    NTSTATUS sta=IoCreateDevice(
        pDriver,
        0,
        &Device_Name,
        FILE_DEVICE_UNKNOWN,
        FILE_DEVICE_SECURE_OPEN,
        FALSE,
        &Device_Object
    );
    if (!NT_SUCCESS(sta))
    {
        DbgPrint("创建设备对象失败");
    }

    UNICODE_STRING SYMBOL_NAME = { 0 };
    RtlInitUnicodeString(&SYMBOL_NAME, L"\\??\\FuckSymbol");
    sta=IoCreateSymbolicLink(&SYMBOL_NAME,&Device_Name);
    if (!NT_SUCCESS(sta))
    {

```

```

        DbgPrint("创建符号链接失败");
    }

    pDriver->MajorFunction[IRP_MJ_CREATE] = DEFAULT_DISPATCH;
    pDriver->MajorFunction[IRP_MJ_CLOSE] = DEFAULT_DISPATCH;
    pDriver->MajorFunction[IRP_MJ_DEVICE_CONTROL] = Main_DISPATCH;
    //pDriver->Flags |= DO_BUFFERED_IO;
    Device_Object->Flags |= DO_BUFFERED_IO;
    return STATUS_SUCCESS;
}

```

```

#include<iostream>
#include<windows.h>
#define CODE_INDEX 0x800
#define CT_TEST
CTL_CODE(FILE_DEVICE_UNKNOWN, CODE_INDEX, METHOD_BUFFERED, FILE_ANY_ACCESS)
//#define Symbolic_link L"\\??\\SymbolLink"//试用于win7以上版本
#define Symbolic_link L"\\\\.\\FuckSymblo"
using namespace std;
typedef struct
{
    ULONG TYPE;
    ULONG RESULT;
    ULONG64 DATA;
    ULONG64 SIZE;
}DATA, * PDATA;
int main()
{
    HANDLE h_Device = CreateFile(
        Symbolic_link,
        GENERIC_READ | GENERIC_WRITE,
        0,
        0,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        0);
    if (!h_Device)
    {
        cout << "打开失败! " << endl;
        return 0;
    }
    DATA inbuf = { 0 };
    inbuf.DATA = 1;
    ULONG a = 0;
    DeviceIoControl(h_Device,
        CT_TEST,
        &inbuf,
        sizeof(DATA),
        &inbuf,
        sizeof(DATA),
        &a, //这里不赋值R3会崩
        0);
    cout << inbuf.DATA << endl;
    CloseHandle(h_Device);
    system("pause");
    return 0;
}

```



## 5.IRP封装

### 1.多特征码内核定位

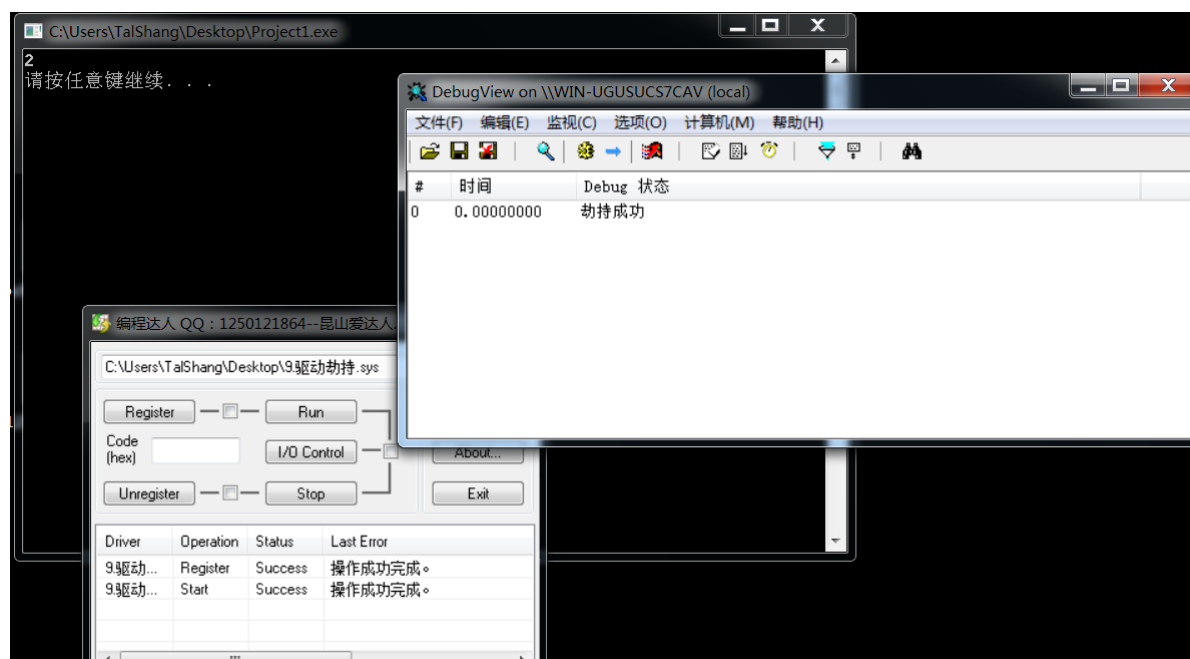
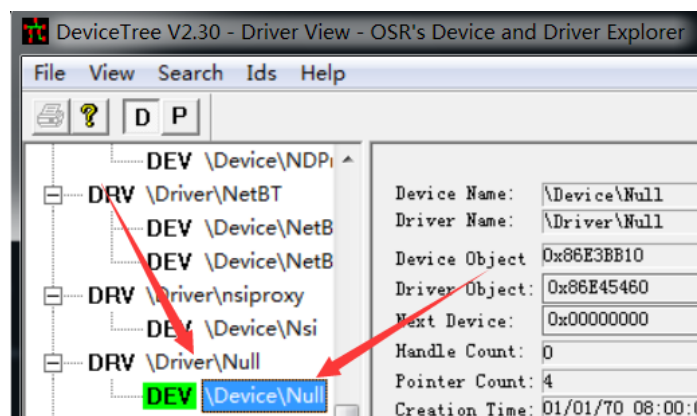
tools\_驱动特征码

### 2.封装驱动

```
//1. 结构体的选择  
//2. 用回调函数来处理IRP  
//全部封装成函数，按照API的简单易用
```

### 3.驱动劫持

Name	Type	SymLink
NDIS	SymbolicLink	\Device\Ndis
NdisWan	SymbolicLink	\Device\NdisWan
NDISWANBH	SymbolicLink	\Device\NDMP5
NDISWANIP	SymbolicLink	\Device\NDMP6
NDISWANIPV6	SymbolicLink	\Device\NDMP7
Nsi	SymbolicLink	\Device\Nsi
NUL	SymbolicLink	\Device\Null



## R0代码

```
#pragma once
#include<ntifs.h>
#define CODE_INDEX 0x800
#define CT_TEST
CTL_CODE(FILE_DEVICE_UNKNOWN, CODE_INDEX, METHOD_BUFFERED, FILE_ANY_ACCESS)
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
);
extern POBJECT_TYPE* IoDriverObjectType;
PDRIVER_OBJECT FindDriverObject(wchar_t* name)
{
    UNICODE_STRING DriverName;
    RtlInitUnicodeString(&DriverName, name);
    PDRIVER_OBJECT pDriverObject = 0;
    NTSTATUS status = ObReferenceObjectByName(&DriverName, OBJ_CASE_INSENSITIVE,
0, 0, *IoDriverObjectType, KernelMode, 0, &pDriverObject);//未文档化函数。
    if (!NT_SUCCESS(status))
    {
        KdPrint(("查找驱动失败"));
        return 0;
    }
    if (pDriverObject)
    {
        ObDereferenceObject(pDriverObject);//引用计数-1，如果想长久持有对象则不加此函数。
    }
    return pDriverObject;
}
```

```
#include"1.h"
typedef struct
{
    ULONG TYPE;
    ULONG RESULT;
    ULONG64 DATA;
    ULONG64 SIZE;
}DATA, * PDATA;

NTSTATUS Main_DISPATCH(
    _In_ struct _DEVICE_OBJECT* DeviceObject,
    _Inout_ struct _IRP* Irp)
{
    PIO_STACK_LOCATION pIrp = IoGetCurrentIrpStackLocation(Irp);
    ULONG Code = pIrp->Parameters.DeviceIoControl.IoControlCode;
    switch (Code)
    {
        case CT_TEST:
```

```

{
    DbgPrint("劫持成功\n");
    PDATA pdata=(PDATA)Irp->AssociatedIrp.SystemBuffer;
    pdata->DATA = 0x2;
    break;
}
}
Irp->IoStatus.Status = STATUS_SUCCESS;
Irp->IoStatus.Information = sizeof(DATA); //坑 没写返回长度 就没覆盖缓冲区
IoCompleteRequest(Irp,0); //坑 它是按长度覆盖缓冲区的 长度不够也拉跨
return STATUS_SUCCESS;
}

typedef NTSTATUS (NTAPI* IRP_DISPATCH)(
    _In_ struct _DEVICE_OBJECT* DeviceObject,
    _Inout_ struct _IRP* Irp);
IRP_DISPATCH pCallback = 0;
VOID DRIVERUNLOAD(_In_ struct _DRIVER_OBJECT* DriverObject)
{
    PDRIVER_OBJECT Driver_1 = 0;
    Driver_1 = FindDriverObject(L"\\Driver\\Null");
    if (Driver_1 == 0)
    {
        DbgPrint("未找到驱动\n");
        return STATUS_SUCCESS;
    }
    if (pCallback != 0)
    {
        Driver_1->MajorFunction[IRP_MJ_DEVICE_CONTROL] = pCallback;
    }
}

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pReg)
{
    pDriver->DriverUnload = DRIVERUNLOAD;
    PDRIVER_OBJECT Driver_1 = 0;
    Driver_1 = FindDriverObject(L"\\Driver\\Null");
    if (Driver_1 == 0)
    {
        DbgPrint("未找到驱动\n");
        return STATUS_SUCCESS;
    }
    pCallback = Driver_1->MajorFunction[IRP_MJ_DEVICE_CONTROL];
    Driver_1->MajorFunction[IRP_MJ_DEVICE_CONTROL] = Main_DISPATCH;
    return STATUS_SUCCESS;
}

```

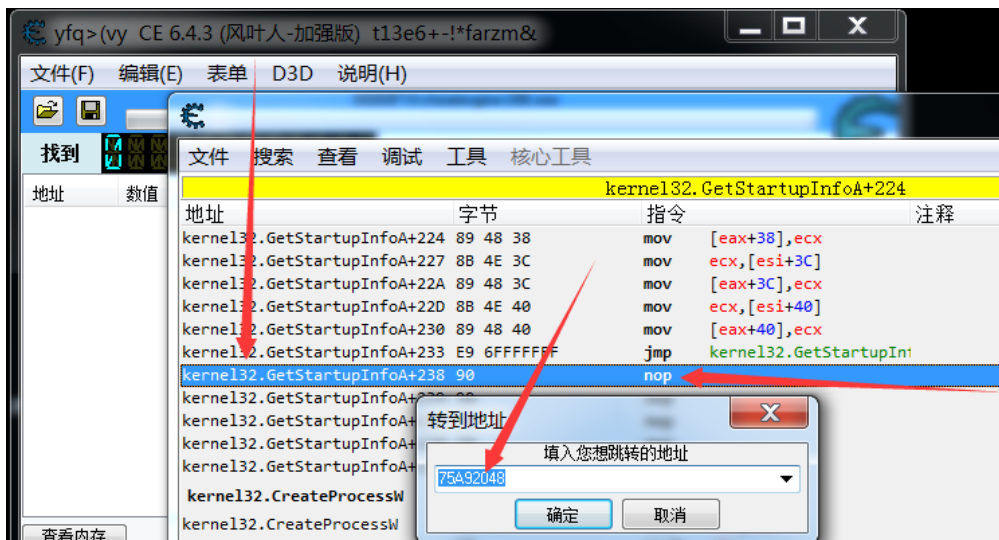
## R3代码

```

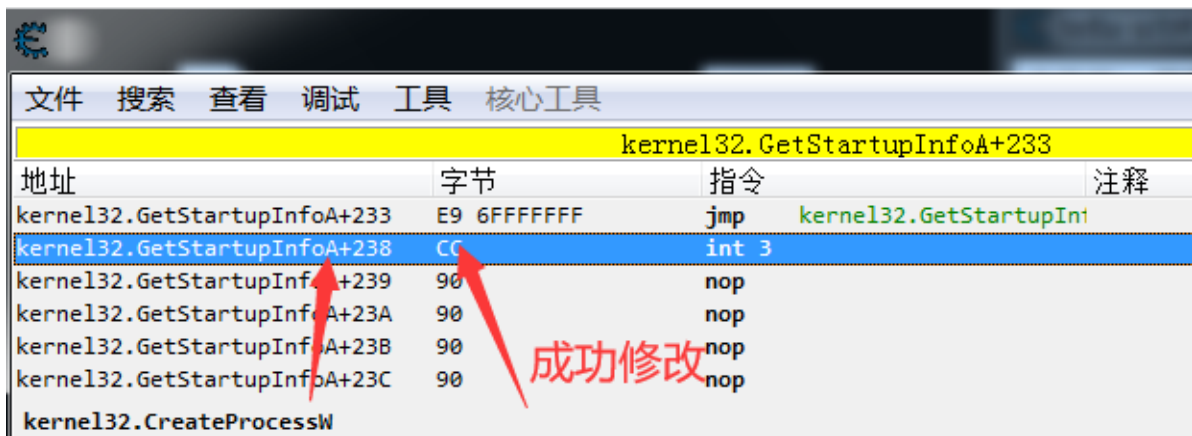
#include<iostream>
#include<windows.h>
#define CODE_INDEX 0x800
#define CT_TEST
CTL_CODE(FILE_DEVICE_UNKNOWN, CODE_INDEX, METHOD_BUFFERED, FILE_ANY_ACCESS)
// #define symbolic_link L"\\??\\sybmolLink" //试用于win7以上版本
#define symbolic_link L"\\\\.\\Nu1" //坑啊啊啊啊啊啊
using namespace std;
typedef struct

```





地址: 0x75A92048



## R0代码

```
#include<ntifs.h>
#include<intrin.h> //R3 R0都有
VOID DRIVERUNLOAD(_In_ struct _DRIVER_OBJECT* DriverObject)
{
}
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pReg)
{
    pDriver->DriverUnload = DRIVERUNLOAD;
    PEPROCESS ep = 0;
    PsLookupProcessByProcessId((HANDLE)392, &ep); //随便挂靠某进程
    if (ep)
    {
        KAPC_STATE kApcState = { 0 };
        KeStackAttachProcess(ep, &kApcState);
        //CR0
        ULONG value = __readcr0();
        __writecr0(value & (~0x10000)); //抹零
        memset(0x75A92048, 0xCC, 1);
        __writecr0(value);
        //CR0
        KeUnstackDetachProcess(&kApcState);
    };
    return STATUS_SUCCESS;
}
```

## 6.驱动IO通信随机化1

原理：寻找某驱动的空白地址，写入shellcode，替换IRP\_MJ\_CONTROL的地址，Shellcode call自己驱动地址。

### 1.查找驱动可利用地址

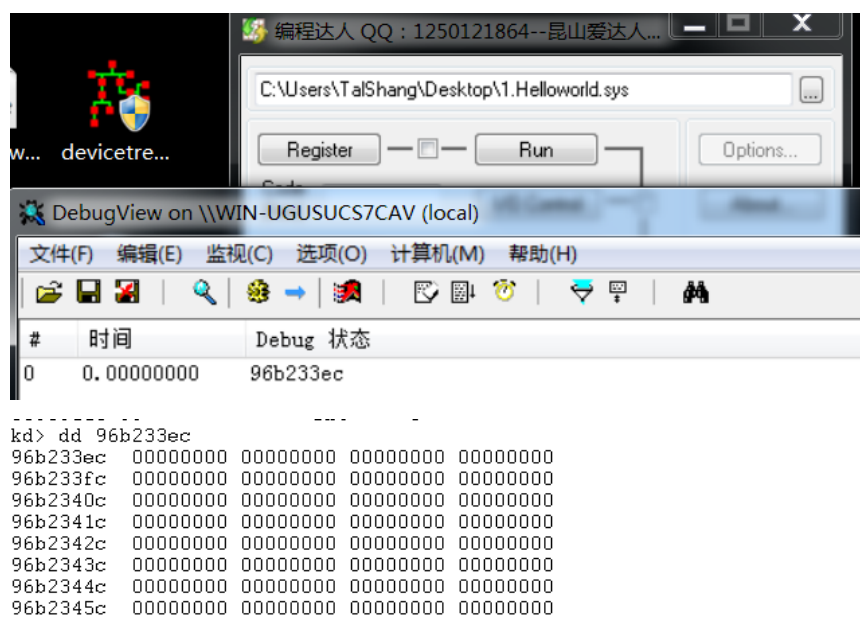
寻找已加载驱动的未使用的节的剩余空间。（可读可执行）

标志(属性块) 常用特征值对照表：

[值:0000020h]	[IMAGE_SCN_CNT_CODE	// Section contains code. (包含可执行代码)]
[值:0000040h]	[IMAGE_SCN_CNT_INITIALIZED_DATA	// Section contains initialized data. (该块包含已初始化的数据)]
[值:0000080h]	[IMAGE_SCN_CNT_UNINITIALIZED_DATA	// Section contains uninitialized data. (该块包含未初始化的数据)]
[值:0000200h]	[IMAGE_SCN_LNK_INFO	// Section contains comments or some other type of information.]
[值:0000800h]	[IMAGE_SCN_LNK_REMOVE	// Section contents will not become part of image.]
[值:0001000h]	[IMAGE_SCN_LNK_COMDAT	// Section contents comdat.]
[值:0004000h]	[IMAGE_SCN_NO_DEFER_SPEC_EXC	// Reset speculative exceptions handling bits in the TLB entries for
[值:0008000h]	[IMAGE_SCN_GPREL	// Section content can be accessed relative to GP.]
[值:0050000h]	[IMAGE_SCN_ALIGN_16BYTES	// Default alignment if no others are specified.]
[值:0100000h]	[IMAGE_SCN_LNK_NRELOC_OVFL	// Section contains extended relocations.]
[值:0200000h]	[IMAGE_SCN_MEM_DISCARDABLE	// Section can be discarded.]
[值:0400000h]	[IMAGE_SCN_MEM_NOT_CACHED	// Section is not cachable.]
[值:0800000h]	[IMAGE_SCN_MEM_NOT_PAGED	// Section is not pageable.]
[值:1000000h]	[IMAGE_SCN_MEM_SHARED	// Section is shareable(该块为共享块).]
[值:2000000h]	[IMAGE_SCN_MEM_EXECUTE	// Section is executable.(该块可执行)]
[值:4000000h]	[IMAGE_SCN_MEM_READ	// Section is readable.(该块可读)]
[值:8000000h]	[IMAGE_SCN_MEM_WRITE	// Section is writeable.(该块可写)]

一下就是0x60000020

查找成功



```
#pragma once
#include<ntifs.h>
#include<ntimage.h> //PE文件库

//ULONG_PTR 在x86编译就是四字节 x64就是8字节
ULONG_PTR Find_DLL_Space(PVOID Module, ULONG size)
{
    if (Module == NULL || size == NULL)
    {
        return 0;
    }

    PIMAGE_DOS_HEADER pDos = (PIMAGE_DOS_HEADER)Module;
```

```

PIMAGE_NT_HEADERS pNT_32 = (PIMAGE_NT_HEADERS)((ULONG_PTR)Module + pDos->e_lfanew);
PIMAGE_SECTION_HEADER pSection = IMAGE_FIRST_SECTION(pNT_32);
//
static char Space_00[100] = { 0 };
static char Space_90[100] = { 0 };
static char Space_CC[100] = { 0 };
if (size >= 100) return 0;
memset(Space_90, 0x90, size);
memset(Space_CC, 0xCC, size);
//
ULONG_PTR isFind = 0;
//
for (int i = 0; i < pNT_32->FileHeader.NumberOfSections; i++)
{
    if ((pSection->Characteristics & 0x60000020) == 0x60000020)
    {
        PCHAR pAddress = (PCHAR)(pSection->VirtualAddress +
(ULONG_PTR)Module);
        ULONG size_Section = pSection->SizeOfRawData;
        PCHAR pEnd_Address = pAddress + size_Section - size;
        for (; pEnd_Address != pAddress; pEnd_Address--)
        {
            //注意x86还需要读一下，x64不需要。
            //确保地址跨页仍然有效
            if (!MmIsAddressValid(pEnd_Address) ||
!MmIsAddressValid(pEnd_Address + size - 1))
            {
                continue;
            }
            if (memcmp(pEnd_Address, Space_00, size) == 0)
            {
                isFind = (ULONG_PTR)pEnd_Address;
                break;
            }
            if (memcmp(pEnd_Address, Space_90, size) == 0)
            {
                isFind = (ULONG_PTR)pEnd_Address;
                break;
            }
            if (memcmp(pEnd_Address, Space_CC, size) == 0)
            {
                isFind = (ULONG_PTR)pEnd_Address;
                break;
            }
        }
    }
    pSection++;
}
if (isFind!=0)
{
    break;
}
return isFind;
}
}
//////////引用//////////
#include "FindPESpace.h"
VOID DRIVERUNLOAD(_In_ struct _DRIVER_OBJECT* DriverObject)

```

```

{

}

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pReg)
{
    pDriver->DriverUnload = DRIVERUNLOAD;
    ULONG_PTR a = Find_DLL_Space(pDriver->DriverStart,20);
    DbgPrint("%x", a);
    return STATUS_SUCCESS;
}

```

## 2.手搓shellcode

```

/*
push ebp
mov ebp,esp
sub esp,0x40    //only 模拟
push [ebp+0xc]
push [ebp+0x8]
push ebx
mov eax,0x1234
mov ebx,0x5678
shl eax,0x10
or eax,ebx
pop ebx
call eax
add esp,0x40    //only 模拟
pop ebp
ret 0x8
*/
776AECC3 | 55                | push ebp                |
776AECC4 | 8BEC              | mov ebp,esp             |
776AECC6 | 83EC 40           | sub esp,40              |
776AECC9 | FF75 0C           | push dword ptr ss:[ebp+C] |
776AECCC | FF75 08           | push dword ptr ss:[ebp+8] |
776AECCF | 53                | push ebx                |
776AECDD | B8 34120000       | mov eax,1234            |
776AECDE | BB 78560000       | mov ebx,5678            |
776AECDA | C1E0 10           | shl eax,10              |
776AECDD | 09D8              | or eax,ebx              |
776AECDF | 5B                | pop ebx                 |
776AECE0 | FFD0              | call eax                |
776AECE2 | 83C4 40           | add esp,40              |
776AECE5 | 5D                | pop ebp                 |
776AECE6 | C2 0800           | ret 8                   |

55
8BEC
83EC 40
FF75 0C
FF75 08
53
B8 34120000
BB 78560000
C1E0 10
09D8
5B

```



```
FFD0
83C4 40
5D
C2 0800
```

## MmCopyVirtualMemory

ReadProcessMemory和WriteProcessMemory的最终调用

```
NTSTATUS
MmCopyVirtualMemory(
    IN PEPROCESS FromProcess, //从哪个进程读 PsGetCurrentProcess() 当前进程
    IN CONST VOID *FromAddress, //读哪个地址
    IN PEPROCESS ToProcess, //读到哪个进程
    OUT PVOID ToAddress, //读到哪个地址
    IN SIZE_T BufferSize, //读多少
    IN KPROCESSOR_MODE PreviousMode, //读的模式 KernelMode UserMode
    OUT PSIZE_T NumberOfBytesCopied // 实际读的大小
)
//在内核中的用处是 挂上物理页
```

```
//读地址的目的是,让挂上的物理页不被释放。
//x64因内存大 不需要读 x86内存小 需要读
ULONG Read_ADDR = 0;
ULONG Read_RET = 0;
MmCopyVirtualMemory(
    PsGetCurrentProcess(),
    pEnd_Address,
    PsGetCurrentProcess(),
    &Read_ADDR,
    0x4,
    KernelMode,
    &Read_RET);
//
```

## 3.完整代码

R3和劫持的一样

R0

```
#pragma once
#include<ntifs.h>
#include<ntimage.h> //PE文件库
NTSTATUS MmCopyVirtualMemory(
    IN PEPROCESS FromProcess,
    IN CONST VOID* FromAddress,
    IN PEPROCESS ToProcess,
    OUT PVOID ToAddress,
    IN SIZE_T BufferSize,
    IN KPROCESSOR_MODE PreviousMode,
    OUT PSIZE_T NumberOfBytesCopied);

//ULONG_PTR 在x86编译就是四字节 x64就是8字节
```

```

ULONG_PTR Find_DLL_Space(PVOID Module, ULONG size)
{
    if (Module == NULL || size == NULL)
    {
        return 0;
    }

    PIMAGE_DOS_HEADER pDos = (PIMAGE_DOS_HEADER)Module;
    PIMAGE_NT_HEADERS pNT_32 = (PIMAGE_NT_HEADERS)((ULONG_PTR)Module + pDos->e_lfanew);
    PIMAGE_SECTION_HEADER pSection = IMAGE_FIRST_SECTION(pNT_32);
    //
    static char Space_00[100] = { 0 };
    static char Space_90[100] = { 0 };
    static char Space_CC[100] = { 0 };
    if (size >= 100) return 0;
    memset(Space_90, 0x90, size);
    memset(Space_CC, 0xCC, size);
    //
    ULONG_PTR isFind = 0;
    //
    for (int i = 0; i < pNT_32->FileHeader.NumberOfSections; i++)
    {
        if ((pSection->Characteristics & 0x60000020) == 0x60000020)
        {
            PCHAR pAddress = (PCHAR)(pSection->VirtualAddress +
(ULONG_PTR)Module);
            ULONG size_Section = pSection->SizeOfRawData;
            PCHAR pEnd_Address = pAddress + size_Section - size;
            for (; pEnd_Address != pAddress; pEnd_Address--)
            {
                //读地址的目的是,让挂上的物理页不被换出。
                //x64因内存大 不需要读 x86内存小 需要读
                ULONG Read_ADDR = 0;
                ULONG Read_RET = 0;
                MmCopyVirtualMemory(
                    PsGetCurrentProcess(),
                    pEnd_Address,
                    PsGetCurrentProcess(),
                    &Read_ADDR,
                    0x4,
                    KernelMode,
                    &Read_RET);
                //
                //确保地址跨页仍然有效
                if (!MmIsAddressValid(pEnd_Address) ||
!MmIsAddressValid(pEnd_Address + size - 1))
                {
                    continue;
                }
                if (memcmp(pEnd_Address, Space_00, size) == 0)
                {
                    isFind = (ULONG_PTR)pEnd_Address;
                    break;
                }
                if (memcmp(pEnd_Address, Space_90, size) == 0)
                {
                    isFind = (ULONG_PTR)pEnd_Address;

```

```

        break;
    }
    if (memcmp(pEnd_Address, Space_CC, size) == 0)
    {
        isFind = (ULONG_PTR)pEnd_Address;
        break;
    }
}
}
if (isFind!=0)
{
    break;
}
pSection++;
}
return isFind;
}

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
);

extern POBJECT_TYPE* IoDriverObjectType;
PDRIVER_OBJECT FindDriverObject(wchar_t* name)
{
    UNICODE_STRING DriverName;
    RtlInitUnicodeString(&DriverName, name);
    PDRIVER_OBJECT pDriverObject = 0;
    NTSTATUS status = ObReferenceObjectByName(&DriverName, OBJ_CASE_INSENSITIVE,
0, 0, *IoDriverObjectType, KernelMode, 0, &pDriverObject);//未文档化函数。
    if (!NT_SUCCESS(status))
    {
        KdPrint(("查找驱动失败"));
        return 0;
    }
    if (pDriverObject)
    {
        ObDereferenceObject(pDriverObject);//引用计数-1, 如果想长久持有对象则不加此函
数。
    }
    return pDriverObject;
}

```

```

#include "FindPESpace.h"
#define CODE_INDEX 0x800
#define CT_TEST
CTL_CODE(FILE_DEVICE_UNKNOWN, CODE_INDEX, METHOD_BUFFERED, FILE_ANY_ACCESS)
typedef struct
{
    ULONG TYPE;
    ULONG RESULT;
}

```

```

        ULONG64 DATA;
        ULONG64 SIZE;
    }DATA, * PDATA;
NTSTATUS Main_DISPATCH(
    _In_ struct _DEVICE_OBJECT* DeviceObject,
    _Inout_ struct _IRP* Irp)
{
    PIO_STACK_LOCATION pIrp = IoGetCurrentIrpStackLocation(Irp);
    ULONG Code = pIrp->Parameters.DeviceIoControl.IoControlCode;
    switch (Code)
    {
    case CT_TEST:
    {
        DbgPrint("劫持成功\n");
        PDATA pdata = (PDATA)Irp->AssociatedIrp.SystemBuffer;
        pdata->DATA = 0x2;
        break;
    }
    }
    Irp->IoStatus.Status = STATUS_SUCCESS;
    Irp->IoStatus.Information = sizeof(DATA); //坑 没写返回长度 就没覆盖缓冲区
    IoCompleteRequest(Irp, 0); //坑 它是按长度覆盖缓冲区的 长度不够也拉
跨
    return STATUS_SUCCESS;
}
VOID DRIVERUNLOAD(_In_ struct _DRIVER_OBJECT* DriverObject)
{
}
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pReg)
{
    DbgBreakPoint();
    pDriver->DriverUnload = DRIVERUNLOAD;
    char shellcode[] = {
        0x55,
        0x8B, 0xEC,
        0x83, 0xEC, 0x40,
        0xFF, 0x75, 0x0C,
        0xFF, 0x75, 0x08,
        0x53,
        0xB8, 0x34, 0x12, 0x00, 0x00, //14
        0xBB, 0x78, 0x56, 0x00, 0x00, //19
        0xC1, 0xE0, 0x10,
        0x09, 0xD8,
        0x5B,
        0xFF, 0xD0,
        0x83, 0xC4, 0x40,
        0x5D,
        0xC2, 0x08, 0x00 };
    PDRIVER_OBJECT Driver_1 = FindDriverObject(L"\\Driver\\Null");
    if (!Driver_1)
    {
        DbgPrint("寻找驱动失败\n");
        return STATUS_SUCCESS;
    }
    ULONG_PTR a = Find_DLL_Space(Driver_1->DriverStart, sizeof(shellcode));
    DbgPrint("%x", a);
    if (a == 0)

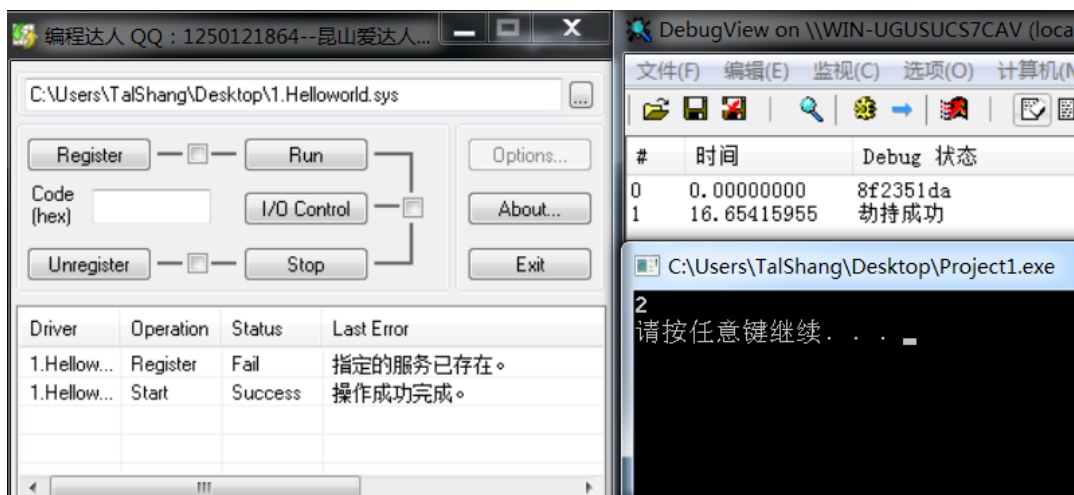
```

```

{
    DbgPrint("查找内存失败\n");
    return STATUS_SUCCESS;
}
else
{
    //修复shellcode
    ULONG_PTR low = (ULONG_PTR)Main_DISPATCH & 0xFFFF;
    ULONG_PTR high = ((ULONG_PTR)Main_DISPATCH >> 0x10) & 0xFFFF;
    *(PULONG)&shellcode[14] = high;
    *(PULONG)&shellcode[19] = low;
    //
    //映射
    PHYSICAL_ADDRESS physical_Add=MmGetPhysicalAddress(a);//不好用,拿到的物理页
    可能被换出
    PVOID mem=MmMapIoSpace(physical_Add,sizeof(shellcode),MmCached);/*此函数将
    指定的物理地址映射到不可分页的
    系统地址空间的一部分。 注意映射的物理页的属性 默认为可读可写可执行,因为函数内部调用了
    MiMapWithLargePages传入的参数是可读可写可执行*/
    if (mem)
    {
        memcpy(mem, shellcode,sizeof(shellcode));
        MmUnmapIoSpace(mem, sizeof(shellcode));
    }
    Driver_1->MajorFunction[IRP_MJ_DEVICE_CONTROL] = a;
}
return STATUS_SUCCESS;
}
}

```

## 成功图



## 4.文件\服务检测

通过驱动对象的DriverExtension 找到ServiceKeyName服务名, 通过注册表找到服务名对应的文件路径。

通过DriverSection。

对比文件和内存。

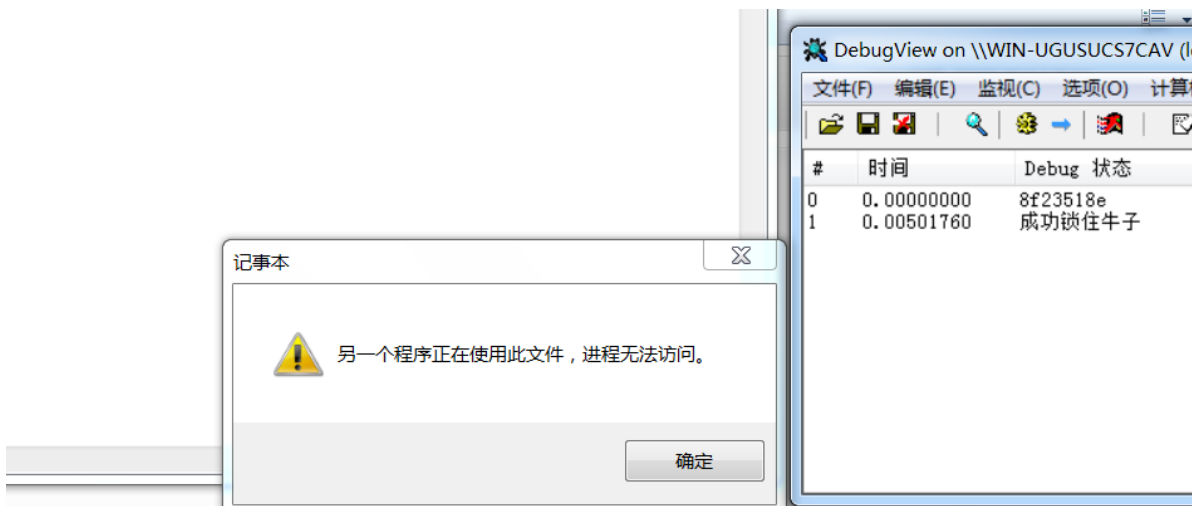
```

kd> dt pDriver
Local var @ 0x83ae29e0 Type _DRIVER_OBJECT*
0x8812ee10
+0x000 Type : 0n4
+0x002 Size : 0n168
+0x004 DeviceObject : (null)
+0x008 Flags : 2
+0x00c DriverStart : 0x959f6000 Void
+0x010 DriverSize : 0x0000
+0x014 DriverSection : 0x888b5850 Void
+0x018 DriverExtension : 0x8812eeb8 _DRIVER_EXTENSION
+0x01c DriverName : _UNICODE_STRING "\Driver\1.Helloworld"
+0x024 HardwareDatabase : 0x841b0250 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPT
+0x028 FastIoDispatch : (null)
+0x02c DriverInit : 0x959f6000 long 1_Helloworld!GsDriverEntry+0
+0x030 DriverStartIo : (null)
+0x034 DriverUnload : (null)
+0x038 MajorFunction : [28] 0x83ef9da3 long nt!IopInvalidDeviceRequest+0
kd> dt 0x8812eeb8 _DRIVER_EXTENSION
nt!_DRIVER_EXTENSION
+0x000 DriverObject : 0x8812ee10 _DRIVER_OBJECT
+0x004 AddDevice : (null)
+0x008 Count : 0
+0x00c ServiceKeyName : _UNICODE_STRING "1.Helloworld"
+0x014 ClientDriverExtension : (null)
+0x018 FsFilterCallbacks : (null)

```

反检测:

- 1.清空服务名和Basedllname FullDllname (buffer和长度)。
- 2.锁住文件。//除非清空文件标志又打开(针对



```
LockFile(L"\\??\\C:\\windows\\system32\\drivers\\NULL.SYS")
```

```

HANDLE LockFile(wchar_t* path) //注意关闭文件句柄
{
    HANDLE hFile = NULL;
    OBJECT_ATTRIBUTES OBJ = { 0 };
    UNICODE_STRING unFileName = { 0 };
    RtlInitUnicodeString(&unFileName, path);
    InitializeObjectAttributes(
        &OBJ,
        &unFileName,
        OBJ_CASE_INSENSITIVE|OBJ_KERNEL_HANDLE,
        NULL,
        NULL);
    IO_STATUS_BLOCK IoStack= { 0 };//返回一个文件的状态
    NTSTATUS sta=ZwCreateFile(
        &hFile,
        FILE_ALL_ACCESS,
        &OBJ,
        &IoStack, //指向接受最终完成状态和所请求操作信息的变量的指针。
        NULL, //只有创建 覆盖 替换文件 大小才生效
        FILE_ATTRIBUTE_NORMAL, //除了文件的创建 替换 覆盖以外，一般是这个属性
        0, //0代表独占 不给别人任何访问权限

```

```

        FILE_OPEN,
        FILE_NON_DIRECTORY_FILE,
        0,
        0);
    if (!NT_SUCCESS(sta))
    {
        DbgPrint("打开文件失败\n");
        return 0;
    }
    if (hFile == NULL)
    {
        DbgPrint("文件句柄为0\n");
        return 0;
    }
    return hFile;
}
//清空服务
PKLDR_DATA_TABLE_ENTRY ldr = (PKLDR_DATA_TABLE_ENTRY)pDriver->DriverSection;

memset(pDriver->DriverExtension->ServiceKeyName.Buffer, 0, pDriver->DriverExtension->ServiceKeyName.Length);
pDriver->DriverExtension->ServiceKeyName.Length = 0;
if (ldr)
{
    memset(ldr->FullDllName.Buffer, 0, ldr->FullDllName.Length);
    memset(ldr->BaseDllName.Buffer, 0, ldr->BaseDllName.Length);
    ldr->BaseDllName.Length = 0;
    ldr->FullDllName.Length = 0;
}

```

## 反反文件检测

//从句柄 和文件标识下手

/\* 强制解锁因其他进程占用而无法删除的文件。

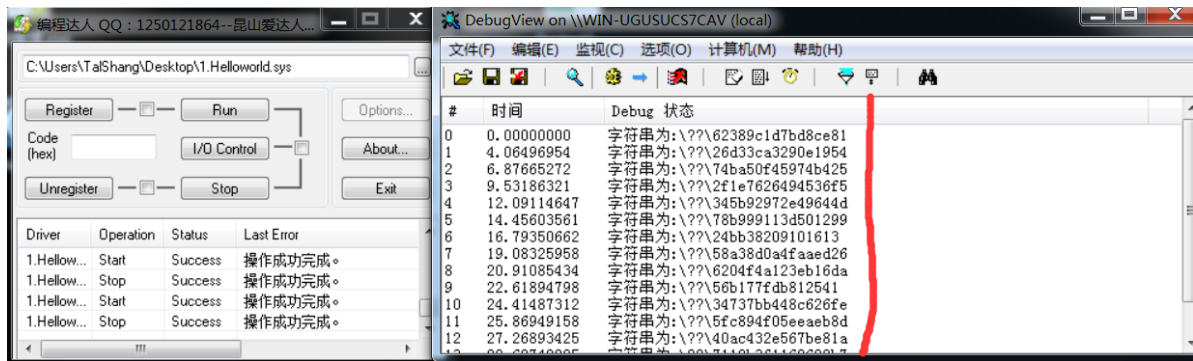
- 1.调用 **ZwQuerySystemInformation** 的 16 功能号来枚举系统里的句柄
- 2.打开拥有此句柄的进程并把此句柄复制到自己的进程
- 3.用 **ZwQueryObject** 查询句柄的类型和名称
- 4.如果 发现此句柄的类型是文件句柄， 名称和被锁定的文件一致，就关闭此句柄
- 5.重复 2、3、4 步，直到遍历完系统里所有的句柄

第4步中因为是要解锁其他进程占用的文件所以有如下细节：

- 1.用 **KeStackAttachProcess**“依附”到目标进程
- 2.用 **ObSetHandleAttributes** 设置句柄为“可以关闭”
- 3.用 **ZwClose** 关闭句柄
- 4.用 **KeUnstackDetachProcess** 脱离“依附”目标进程

## 7.驱动IO通信随机化2

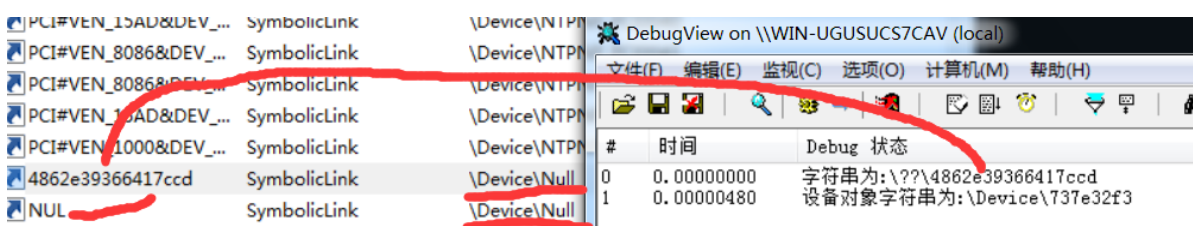
### 1.字符串随机地址随机字符



```
#pragma once
#include<ntifs.h>
#include<ntstrsafe.h> //驱动字符串

PUNICODE_STRING RandomString()
{
    static UNICODE_STRING str_symbolic= { 0 };
    if (str_symbolic.Length != 0)
    {
        return &str_symbolic;
    }
    LARGE_INTEGER time = { 0 };
    KeQuerySystemTime(&time);
    ULONG rand_num1=RtlRandom(&time.LowPart); //四个字节的随机数种子 PS:一个就够了 两个太长了
    //高位时间基本不变
    ULONG rand_num2=RtlRandom(&time.LowPart);
    wchar_t bufstring[100] = { 0 };
    RtlStringCbPrintfW(bufstring,100*sizeof(wchar_t),L"\\??\\
    \\%x%x", rand_num1, rand_num2);
    int len = wcslen(bufstring)*2; //wcslen()取字符个数也就是字符串长度
    str_symbolic.Buffer =(PWSTR)ExAllocatePool(PagedPool, len+2);
    //分页内存一般可读可写 非分页内存一般是可读可写可执行
    str_symbolic.Length = len;
    str_symbolic.MaximumLength = len + 2;
    memset(str_symbolic.Buffer, 0, len);
    memcpy(str_symbolic.Buffer, bufstring, len);
    memset(bufstring, 0, len);
    return &str_symbolic;
}
```

### 2.设备对象挂随机符号



//



```

PUNICODE_STRING pu = NULL;
PUNICODE_STRING Device_NAME = NULL;
//
VOID DRIVERUNLOAD(_In_ struct _DRIVER_OBJECT* DriverObject)
{
    PDRIVER_OBJECT Driver_1 = FindDriverObject(L"\\Driver\\Null");
    if (Driver_1)
    {
        // IoDeleteDevice(Driver_1->DeviceObject); 不能删除NULL驱动的设备 不然直接拉闸
        IoDeleteSymbolicLink(pu);
    }
}

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING pReg)
{
    //DbgBreakPoint();
    pDriver->DriverUnload = DRIVERUNLOAD;
    pu = Random_Symbolic();
    Device_NAME = Random_Device();
    DbgPrint("字符串为:%wZ", pu);
    DbgPrint("设备对象字符串为:%wZ", Device_NAME);
    PDRIVER_OBJECT Driver_1 = FindDriverObject(L"\\Driver\\Null");
    if (!Driver_1)
    {
        DbgPrint("寻找驱动失败\n");
        return STATUS_SUCCESS;
    }
    PDEVICE_OBJECT Device_1 = Driver_1->DeviceObject;
    if (!Device_1)
    {
        DbgPrint("寻找设备失败\n");
        NTSTATUS sta = IoCreateDevice(pDriver, 0, Device_NAME,
            FILE_DEVICE_UNKNOWN,
            FILE_DEVICE_SECURE_OPEN,
            FALSE, //其实设置独占更好
            &Device_1);
        if (!NT_SUCCESS(sta))
        {
            DbgPrint("创建设备失败\n");
            return STATUS_SUCCESS;
        }
        Driver_1->DeviceObject = Device_1;
        IoCreateSymbolicLink(pu, Device_NAME);
        return STATUS_SUCCESS;
    }
    else
    {
        UNICODE_STRING Device_NULL = { 0 };
        RtlInitUnicodeString(&Device_NULL, L"\\Device\\Null");
        NTSTATUS sta = IoCreateSymbolicLink(pu, &Device_NULL);
        if (!NT_SUCCESS(sta))
        {
            DbgPrint("挂符号链接失败");
            return STATUS_SUCCESS;
        }
    }
    return STATUS_SUCCESS;
}

```

### 3.System没有PEB

windbg默认断在System进程

```
Image: userinit.exe
PROCESS 88178030 SessionId: 1 Cid: 0968 Peb: 7ffd5000 ParentCid: 0374
DirBase: 3f332280 ObjectTable: 946c8b98 HandleCount: 106.
Image: dwm.exe
PROCESS 88159ae8 SessionId: 1 Cid: 0980 Peb: 7ffd5000 ParentCid: 0960
DirBase: 3f332420 ObjectTable: 835fd578 HandleCount: 462.
Image: explorer.exe
PROCESS 887ff950 SessionId: 1 Cid: 09e4 Peb: 7ffd8000 ParentCid: 0980
DirBase: 3f332440 ObjectTable: 960f8988 HandleCount: 177.
Image: vmtoolsd.exe
PROCESS 88808410 SessionId: 1 Cid: 0a10 Peb: 7ffda000 ParentCid: 0268
DirBase: 3f332460 ObjectTable: 961963b8 HandleCount: 90.
Image: dllhost.exe
kd> r cr3
cr3=00185000
```

查看System的EPROCESS, 发现没有Peb

```
+0x170 ImageFileName : [15] "System
+0x17b PriorityClass : 0x2 ''
+0x17c JobLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x184 LockedPagesList : (null)
+0x188 ThreadListHead : _LIST_ENTRY [ 0x863d6878 - 0x8883b890 ]
+0x190 SecurityPort : (null)
+0x194 PaeTop : 0x83f66fc0 Void
+0x198 ActiveThreads : 0x5c
+0x19c ImagePathHash : 0
+0x1a0 DefaultHardErrorProcessing : 1
+0x1a4 LastThreadExitStatus : 0
+0x1a8 Peb : (null)
+0x1ac PrefetchTrace : _EX_FAST_REF
+0x1b0 ReadOperationCount : _LARGE_INTEGER 0x99
+0x1b8 WriteOperationCount : _LARGE_INTEGER 0x72
+0x1c0 OtherOperationCount : _LARGE_INTEGER 0x14b5
```

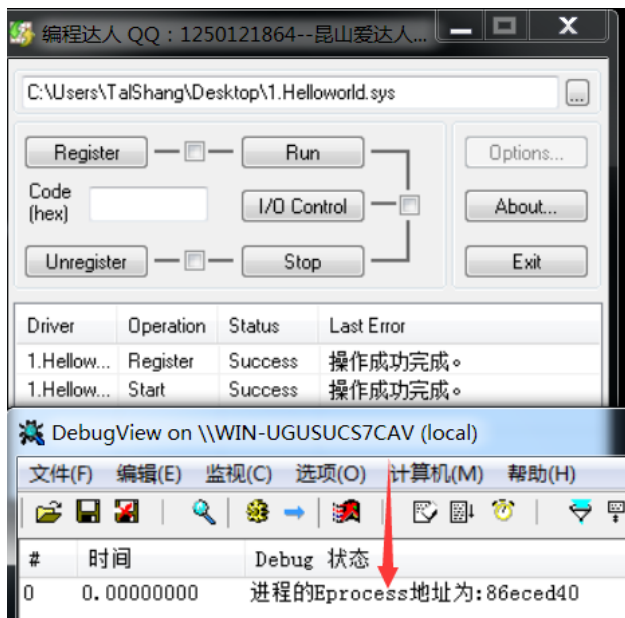
### 4.String传给R3的方法

遍历进程 或者 创建回调

### 5.遍历进程

通过ImageFileName对比遍历找到进程。

```
+0x168 Session : 0x8e322000 Void
+0x16c ImageFileName : [15] "dllhost.exe"
+0x17b PriorityClass : 0x2 ''
+0x17c JobLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x184 LockedPagesList : (null)
+0x188 ThreadListHead : _LIST_ENTRY [ 0x88100740 - 0x87fad9b8 ]
+0x190 SecurityPort : (null)
+0x194 PaeTop : 0x86f323c0 Void
+0x198 ActiveThreads : 0x12
+0x19c ImagePathHash : 0xf3a377f5
```



注意 我没有减少引用计数

```
PEPROCESS GetProcessByname(wchar_t* ProcessName) //没有减少引用计数
{
    PEPROCESS ep = NULL;
    for (int i = 8; i < 70000; i += 4)
    {
        PEPROCESS eprocess = NULL;
        PsLookupProcessByProcessId((HANDLE)i, &ep);
        if (!eprocess) continue;
        PUNICODE_STRING pname = 0;
        SeLocateProcessImageName(&eprocess, &pname); //本质是申请内存，然后拷贝
        ImageFileName结构。
        if (wcsstr(pname->Buffer, ProcessName))
        {
            ep = eprocess;
            ExFreePool(pname);
            break;
        }
        else
        {
            ExFreePool(pname);
        }
    }
    DbgPrint("进程的Eprocess地址为:%x", ep);
    return ep;
}
```

## 1.SeLocateProcessImageName逆向

```

1 int __stdcall SeLocateProcessImageName(int a1, _DWORD *a2)
2 {
3     SIZE_T v3; // esi
4     _DWORD *v4; // eax
5     _DWORD *v5; // edi
6     int v7; // [esp+18h] [ebp+8h]
7
8     *a2 = 0;
9     v7 = SePopulateProcessImageName();
10    if ( v7 >= 0 )
11    {
12        v3 = *(unsigned __int16 *)((_DWORD *)a1 + 492) + 2 + 8;
13        v4 = ExAllocatePoolWithTag(NonPagedPool, v3, 'fIeS');
14        v5 = v4;
15        if ( v4 )
16        {
17            memcpy(v4, *(const void **)(a1 + 492), v3);
18            if ( v5[1] )
19                v5[1] = v5 + 2;
20            *a2 = v5;
21        }
22        else
23        {
24            v7 = -1073741801;
25        }
26    }
27    return v7;
28 }

```

非分页内存

最好用释放一下

## 6.遍历进程的模块

注意：要附加到当前进程，才能看到当前进程的PEB,PEB是三环结构,0X7FF

1.

```

kd> .process /i 8890c4d0
You need to continue execution (press 'g' <enter>) for the context
to be switched. When the debugger breaks in again, you will be in
the new process context.
kd> g
Break instruction exception - code 80000003 (first chance)
nt!RtlpBreakWithStatusInstruction:
83e9a110 cc          int     3

```

2.\_PEB

```

kd> dt 0x7ffd4000 _PEB
nt!_PEB
+0x000 InheritedAddressSpace : 0 ''
+0x001 ReadImageFileExecOptions : 0 ''
+0x002 BeingDebugged : 0 ''
+0x003 BitField : 0 ''
+0x003 ImageUsesLargePages : 0y0
+0x003 IsProtectedProcess : 0y0
+0x003 IsLegacyProcess : 0y0
+0x003 IsImageDynamicallyRelocated : 0y0
+0x003 SkipPatchingUser32Forwarders : 0y0
+0x003 SpareBits : 0y000
+0x004 Mutant : 0xffffffff Void
+0x008 ImageBaseAddress : 0x00400000 Void
+0x00c Ldr : 0x77a57880 _PEB_LDR_DATA

```

进程的模块链表

3.\_PEB\_LDR\_DATA

```
kd> dt 0x77a57880 _PEB_LDR_DATA
```

```
nt!_PEB_LDR_DATA
```

```
+0x000 Length : 0x30
+0x004 Initialized : 0x1 ''
+0x008 SsHandle : (null)
+0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x2c1848 - 0x2cee68 ]
+0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x2c1850 - 0x2cee70 ]
+0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x2c18d8 - 0x2cee78 ]
+0x024 EntryInProgress : (null)
+0x028 ShutdownInProgress : 0 ''
+0x02c ShutdownThreadId : (null)
```

指向同一块内存，只是加载顺序不同

#### 4.\_LDR\_DATA\_TABLE\_ENTRY

```
kd> dt 0x77a57880 _PEB_LDR_DATA
```

```
nt!_PEB_LDR_DATA
```

```
+0x000 Length : 0x30
+0x004 Initialized : 0x1 ''
+0x008 SsHandle : (null)
+0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x2c1848 - 0x2cee68 ]
+0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x2c1850 - 0x2cee70 ]
+0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x2c18d8 - 0x2cee78 ]
+0x024 EntryInProgress : (null)
+0x028 ShutdownInProgress : 0 ''
+0x02c ShutdownThreadId : (null)
```

```
kd> dt _LDR_DATA_TABLE_ENTRY 0x2c1848
```

```
nt!_LDR_DATA_TABLE_ENTRY
```

```
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x2c18c8 - 0x77a5788c ]
+0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x2c18d0 - 0x77a57894 ]
+0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x018 DllBase : 0x00400000 Void
+0x01c EntryPoint : 0x004153b7 Void
+0x020 SizeOfImage : 0x87000
+0x024 FullDllName : _UNICODE_STRING "C:\Users\TaIshang\Desktop\Dbgview.exe"
+0x02c BaseDllName : _UNICODE_STRING "Dbgview.exe"
+0x034 Flags : 0x4000
+0x038 LoadCount : 0xffff
```

## 1.PsGetProcessPeb

```
1 int __stdcall PsGetProcessPeb(int a1)
2 {
3     return *(_DWORD*)(a1 + 424);
4 }
```

```
PVOID NTAPI PsGetProcessPeb(PEPROCESS ep);
```

## 2.坑

```
kd> dt Next
```

```
Local var @ 0x807e09a0 Type _LDR_DATA_TABLE_ENTRY*
```

```
0x774e7880
```

```
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x30 - 0x1 ]
+0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x0 - 0x2c18f0 ]
+0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x32af10 - 0x2c18f0 ]
+0x018 DllBase : 0x0032af18 Void
+0x01c EntryPoint : 0x002c1980 Void
+0x020 SizeOfImage : 0x32ae20
+0x024 FullDllName : _UNICODE_STRING ""
+0x02c BaseDllName : _UNICODE_STRING ""
+0x034 Flags : 0
```

坑

```
kd> dt Next
Local var @ 0x807e09a0 Type _LDR_DATA_TABLE_ENTRY*
0x00000030
+0x000 InLoadOrderLinks : _LIST_ENTRY
+0x008 InMemoryOrderLinks : _LIST_ENTRY
+0x010 InInitializationOrderLinks : _LIST_ENTRY
+0x018 DllBase : ???
+0x01c EntryPoint : ???
+0x020 SizeOfImage : ??
+0x024 FullDllName : _UNICODE_STRING
+0x02c BaseDllName : _UNICODE_STRING
+0x034 Flags : ??
+0x038 LoadCount : ??
+0x03a TlsIndex : ??
+0x03c HashLinks : _LIST_ENTRY
+0x03c SectionPointer : ???
+0x040 CheckSum : ??
+0x044 TimeDateStamp : ??
+0x044 LoadedImports : ???
+0x048 EntryPointActivationContext : ???
+0x04c PatchInformation : ???
```

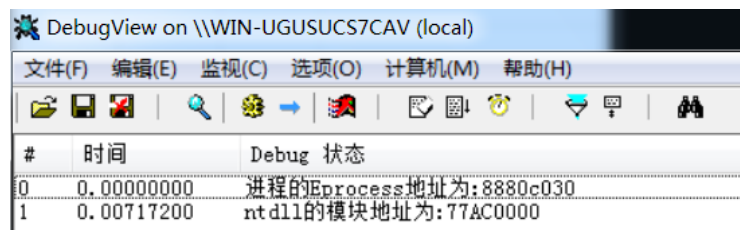
士亢

啥都没有

这里地址不可读 需要直接跳过

预读一下

### 3.遍历模块代码



```
//前提是链表的每个结点的地址有效，每个结点的名字没有清空
//0x30 bytes (sizeof)
struct _PEB_LDR_DATA
{
    ULONG Length;
//0x0
    UCHAR Initialized;
//0x4
    VOID* SsHandle;
//0x8
    LIST_ENTRY InLoadOrderModuleList; //0xc
    LIST_ENTRY InMemoryOrderModuleList; //0x14
    LIST_ENTRY InInitializationOrderModuleList; //0x1c
    VOID* EntryInProgress;
//0x24
    UCHAR ShutdownInProgress;
//0x28
    VOID* ShutdownThreadId;
//0x2c
};
//0x78 bytes (sizeof)
struct _LDR_DATA_TABLE_ENTRY
{
    struct _LIST_ENTRY InLoadOrderLinks;
//0x0
    struct _LIST_ENTRY InMemoryOrderLinks;
//0x8
```

```

        struct _LIST_ENTRY InInitializationOrderLinks;
//0x10
        VOID* DllBase;
//0x18
        VOID* EntryPoint;
//0x1c
        ULONG SizeOfImage;
//0x20
        struct _UNICODE_STRING FullDllName;
//0x24
        struct _UNICODE_STRING BasedDllName;
//0x2c
        ULONG Flags;
//0x34
        USHORT LoadCount;
//0x38
        USHORT TlsIndex;
//0x3a
        union
        {
            struct _LIST_ENTRY HashLinks;
//0x3c
            struct
            {
                VOID* SectionPointer;
//0x3c
                ULONG CheckSum;
//0x40
            };
        };
        union
        {
            ULONG TimeDateStamp;
//0x44
            VOID* LoadedImports;
//0x44
        };
        struct _ACTIVATION_CONTEXT* EntryPointActivationContext;
//0x48
        VOID* PatchInformation;
//0x4c
        struct _LIST_ENTRY ForwarderLinks;
//0x50
        struct _LIST_ENTRY ServiceTagLinks;
//0x58
        struct _LIST_ENTRY StaticLinks;
//0x60
        VOID* ContextInformation;
//0x68
        ULONG OriginalBase;
//0x6c
        union _LARGE_INTEGER LoadTime;
//0x70
    };
    PVOID NTAPI PsGetProcessPeb(PEPROCESS ep);
    PVOID GetDLLBase(PEPROCESS ep, wchar_t* DllName)
    {
        PVOID Hmodule = 0;

```

```

if (ep == NULL || DllName[0] == L'\0')
{
    return 0;
}
PVOID peb = PsGetProcessPeb(ep);
KAPC_STATE apc = { 0 };
KeStackAttachProcess(ep, &apc);
PUCHAR LDR = *(PULONG)((PUCHAR)peb + 0xC);
if (!LDR) return 0;
struct _LDR_DATA_TABLE_ENTRY* List = (struct _LDR_DATA_TABLE_ENTRY*)&((struct
_PEB_LDR_DATA*)LDR)->InLoadOrderModuleList; //这里挂了N次
struct _LDR_DATA_TABLE_ENTRY* Next = List->InLoadOrderLinks.Flink;
while (List != Next)
{
    try//精髓
    {
        ProbeForRead(Next, 1, 1);
        if (Next->BaseDllName.Length && wcsstr(Next->BaseDllName.Buffer,
DllName))
        {
            //需要判断当前节点的Str是否有效 //非常容易异常
            {
                //有些结点没有DLL名字
                Hmodule = Next->DllBase;
                break;
            }
        }
        except(1)
        {
            Next = Next->InLoadOrderLinks.Flink;
            continue;
        }
        Next = Next->InLoadOrderLinks.Flink;
    }
    KeUnstackDetachProcess(&apc);
    return Hmodule;
}

```

## 4.修改模块的DOS头的空白

```

#include<ntimage.h>

VOID EDIT_R3_DLL_DOS(PVOID Dll, PUNICODE_STRING pu, PEPROCESS ep)
{
    //DbgBreakPoint();
    if (Dll == 0 || ep == 0 || pu->Length == 0)
    {
        DbgPrint("修改DOS失败\n");
        return;
    }
    KAPC_STATE apc = { 0 };
    wchar_t bufferName[60] = { 0 };
    RtlStringCbPrintfW(bufferName, 60 * 2, L"\\\\.\\%ws", pu->Buffer + 4);
    //KdPrintEx((77, 0, "字符串:%ws\n", bufferName));

    KeStackAttachProcess(ep, &apc);
    PUCHAR a = (PUCHAR)Dll + sizeof(IMAGE_DOS_HEADER);
    PVOID mem = MmMapIoSpace(MmGetPhysicalAddress(a), 200, MmCached);
    if (!mem)

```

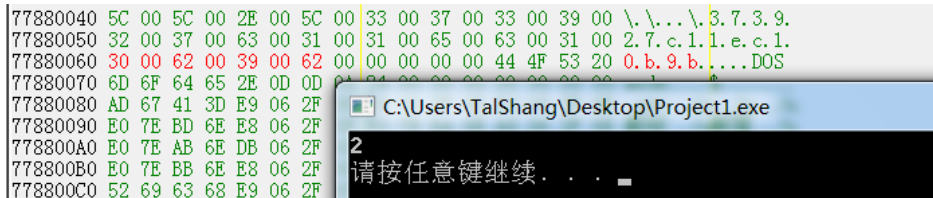


```

{
    DbgPrint("修改DOS映射失败\n");
    return;
}
memset(mem,0, wcslen(bufferName) * 2 + 4); //给R3的字符串读取提供方便
memcpy(mem, bufferName, wcslen(bufferName) * 2 + 2);
MmUnmapIoSpace(mem,200);
KeUnstackDetachProcess(&apc);
}

```

## 5.与R3通信



```

#include<iostream>
#include<windows.h>
#define CODE_INDEX 0x800
#define CT_TEST
CTL_CODE(FILE_DEVICE_UNKNOWN, CODE_INDEX, METHOD_BUFFERED, FILE_ANY_ACCESS)
// #define Symbolic_link L"\\??\\SybmolLink" //试用于win7以上版本
// #define Symbolic_link L"\\\\\\.\Nul" //坑啊啊啊啊啊啊
using namespace std;
typedef struct
{
    ULONG TYPE;
    ULONG RESULT;
    ULONG64 DATA;
    ULONG64 SIZE;
}DATA, * PDATA;
int main()
{
    HMODULE ntdll = GetModuleHandleA("ntdll.dll");
    if (!ntdll)
    {
        cout << "获取ntdll错误" << endl;
        return 0;
    }
    ULONG p_ntdll = (ULONG)ntdll + sizeof(IMAGE_DOS_HEADER);
    wchar_t* pName = (wchar_t*)p_ntdll; //
    HANDLE h_Device = 0;
    h_Device = CreateFile(
        pName,
        GENERIC_READ | GENERIC_WRITE,
        0,
        0,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
        0);
    if (!h_Device)
    {
        cout << "打开失败! " << endl;
        return 0;
    }
}

```

```

DATA inbuf = { 0 };
ULONG a = 0;
DeviceIoControl(h_Device,
    CT_TEST,
    &inbuf,
    sizeof(DATA),
    &inbuf,
    sizeof(DATA),
    &a,
    0);
cout << inbuf.DATA << endl;
CloseHandle(h_Device);
system("pause");
return 0;
}

```

## 8.驱动通信反检测总结

/\*

- 1.自己的驱动有驱动名，设备名，符号链接名，能被检测到，并统计上传。
- 2.采用劫持系统驱动，绕过驱动名，设备名，符号链接名的大数据上传。
- 3.劫持系统驱动，挂上伪装后的**shellcode**，但仍然有自己的驱动在**working**。
- 4.给自己的驱动断链，清除驱动名，防止大数据分析。//效果不好
- 5.锁住自己的驱动和系统驱动，禁止访问上传或者文件 内存对比。//能被针对
- 6.采用了随机符号链接，防止大数据分析。（目的是防止检测到很多R3进程，打开了同一个符号链接。

问题：1.没有隐藏好自己的驱动 2.没有保护好进程。

\*/

## 8.驱动隐藏

/\*

- 1.读PE文件
  - 2.申请内存，拉伸文件按节区复制，//ExAllocatePool被检测到，并且在2004中废弃。并且分配的是进程的堆内存。
  - 3.修复重定位
  - 4.修复IAT //查询模块
  - 5.修复版本差异 //win10能跑，win7蓝屏
  - 6.修复异常 //为了支持易语言
  - 7.call 入口点 //push 0 and 0
  - 8.抹PE头
- 要让驱动支持try except,需要手工支持。
- 内核重载 不需要修复
- wb, 需要pchunter过掉LoadImage系统回调 可能有防火墙。

## SystemLoadGdiDriverInformation

```

NTSTATUS
NTAPI
NtSetSystemInformation (
    in SYSTEM_INFORMATION_CLASS SystemInformationClass,
    in_bcount_opt(SystemInformationLength) PVOID SystemInformation,
    in ULONG SystemInformationLength
)

/*++
Routine Description:
    This function set information about the system.

Arguments:
    SystemInformationClass - The system information class which is to
        be modified.

    SystemInformation - A pointer to a buffer which contains the specified
        information. The format and content of the buffer depend on the
        specified system information class.

    SystemInformationLength - Specifies the length in bytes of the system
        information buffer.

Return Value:
    Returns one of the following status codes:

    STATUS_SUCCESS - Normal, successful completion.

    STATUS_ACCESS_VIOLATION - The specified system information buffer
        is not accessible.

    STATUS_INVALID_INFO_CLASS - The SystemInformationClass parameter

case SystemLoadGdiDriverInformation:
{
    UNICODE_STRING Image;
    PVOID ImageBaseAddress;
    ULONG_PTR EntryPoint;
    PVOID SectionPointer;

    PIMAGE_NT_HEADERS NtHeaders;

    //
    // If the system information buffer is not the correct length,
    // then return an error.
    //
    if (SystemInformationLength != sizeof( SYSTEM_GDI_DRIVER_INFORMATION )) {
        return STATUS_INFO_LENGTH_MISMATCH;
    }

    if (PreviousMode != KernelMode) {
        //
        // The caller's access mode is not kernel so fail.
        // Only GDI from the kernel can call this.
        //
        return STATUS_PRIVILEGE_NOT_HELD;
    }

    Image = ((PSYSTEM_GDI_DRIVER_INFORMATION)SystemInformation)->DriverName;

    Status = MmLoadSystemImage (&Image,
        NULL,
        NULL,
        LoadFlags,
        &SectionPointer,
        (PVOID *) &ImageBaseAddress);

    if ((NT_SUCCESS( Status ))) {
        PSYSTEM_GDI_DRIVER_INFORMATION GdiDriverInfo =
            (PSYSTEM_GDI_DRIVER_INFORMATION) SystemInformation;

        ULONG Size;

```

MmLoadSystemImage 加载了驱动 但没有运行驱动，但是会运行驱动里调用的别的驱动。（maybe work on 杀毒

Zw系列看不到函数原型，就搜Nt系列

## 1.QuerySysModule遍历系统模块

通过ZwQuerySystemInformation遍历所有内核模块。

### ExpQueryModuleInformation

第一次使用ZwQuerySystemInformation，得到返回的所有内核模块所需的结构的总长度。

通过STATUS\_INFO\_LENGTH\_MISMATCH 对比，进行第二次查询。

第二次查询时，需要申请足够的内存，来存放查询结果。



## 1.字符串小写转大写

```

char* CharStrupper(char* str,BOOLEAN IsAllocateMemory)
{
    char* a = str;
    if (IsAllocateMemory)
    {
        a = ExAllocatePool(PagedPool,strlen(str)+2);
        memset(a, 0, strlen(str) + 2);
        memcpy(a, str, strlen(str));
    }
    char * b=_strupr(a);
    return b;
}

```

## 2.遍历几乎所有内核模块

驱动链<ZwQuerySystemInformation(实际也是链表)<驱动对象目录(\Driver)<内核空间的PE头

HTTP.sys	0x83874000	0x00075000	807f4a20	807f4af4	00000000	807f4a3c	83e953e4
browser.sys	0x837c9000	0x000c9000	807f4a30	864014c0	807f4af4	00000000	807f4aac
mpsdrv.sys	0x837c2000	0x000c2000	807f4a40	83e95349	807f4a54	88d470bd	8756f7c8
mrxsmmb.sys	0x838d4000	0x000c23000	807f4a50	8756f7c8	807f4a68	807f4a68	8756f7c8
mrxsmmb10.sys	0x838f7000	0x000c38000					
mrxsmmb20.sys	0x83932000	0x000c18000					
vmemctl.sys	0x83965000	0x000c08000					
peauth.sys	0x96212000	0x000c097000					
secdrv.SYS	0x962A9000	0x000c0A000					
srvmnet.sys	0x962B3000	0x000c021000					
tcpipreg.sys	0x962D4000	0x000c0D000					
srv2.sys	0x962E1000	0x000c04F000					
srv.sys	0x96330000	0x000c051000					

kd> g  
模块地址:83824000, 模块大小:85000

\*BUSY\* Debuggee is running...

//返回模块大小

```

ULONG_PTR QuerySysModule(char* ModuleName, ULONG_PTR *ModuleAddress)
{
    if (ModuleName[0] == '\\0' || ModuleAddress == NULL)
    {
        return 0;
    }
    ULONG_PTR SizeOfImage = 0;

    RTL_PROCESS_MODULES P_M; //第一次出错的目的是为了获取总长度。
    ULONG len = 0;
    NTSTATUS sta=ZwQuerySystemInformation(
        SystemModuleInformation,
        &P_M,
        sizeof(RTL_PROCESS_MODULES),
        &len
    );
    if (sta == STATUS_INFO_LENGTH_MISMATCH)
    {
        PVOID mem = ExAllocatePool(PagedPool,
len+sizeof(RTL_PROCESS_MODULES)); //多给一个模块
        sta = ZwQuerySystemInformation(
            SystemModuleInformation,
            mem,
            len + sizeof(RTL_PROCESS_MODULES),
            &len
        );
        if (!NT_SUCCESS(sta))
        {
            ExFreePool(mem);
            return 0;
        }
        PRTL_PROCESS_MODULES P_R_M = mem;
        PRTL_PROCESS_MODULE_INFORMATION pa = 0;
        char* ModuleName_Upper = CharStrUpper(ModuleName, TRUE);
        char* pFullPathName = NULL;
        for (int i = 0; i < P_R_M->NumberOfModules; i++)
        {
            pa = &P_R_M->Modules[i];
            pFullPathName = CharStrUpper(pa->FullPathName, FALSE);
            if (strstr(pFullPathName, ModuleName_Upper))
            {
                SizeOfImage = pa->ImageSize;
                *ModuleAddress = pa->ImageBase;
                ExFreePool(ModuleName_Upper);
                ExFreePool(mem);
            }
        }
    }
    return SizeOfImage;
}

```

### 3.最全遍历驱动模块方法

RtlPcToFileHeader(线性地址), R3和R0都有的结构, 从线性地址找文件头。

## 2.内存注入

### 1.重定位表x86/X64

X86:

先找重定位表的地址, 不用管整个表的大小, 然后根据重定位块的结构编译整个表, 直到遇到00 00 00 00 00 00

0001 10 00 + 73F = 0001173F

加上ImageBase

Index	Section	RVA	Items
1	2 ("text")	00011000	20h / 32d
2	2 ("text")	00012000	48h / 72d
3	2 ("text")	00013000	58h / 88d
4	2 ("text")	00014000	C0h / 192d

Index	RVA	Offset	Type	Far Address	Data Interpretation
1	0001173F	0000083F	HIGHLOW(3)	0041C000	01 01 01 01 01 01 01
2	00011749	00000849	HIGHLOW(3)	0041A140	00 00 00 00 00 00 00
3	0001179F	0000089F	HIGHLOW(3)	0041C002	01 01 01 01 01 01 01
4	000117C8	000008C8	HIGHLOW(3)	0041B170	IAT Think of "ucrtbased
5	0001182F	0000082F	HIGHLOW(3)	0041C015	01 00 00 00 00 00 00

根据重定位块大小算重定位数量

成功算出

64 ÷ 2 = 32

内

Index	Section	RVA	Items
1	2 ("text")	00011000	20h / 32d
2	2 ("text")	00012000	48h / 72d
3	2 ("text")	00013000	58h / 88d
4	2 ("text")	00014000	C0h / 192d

Index	RVA	Offset	Type	Far Address	Data Interpretation
1	0001173F	0000083F	HIGHLOW(3)	0041C000	01 01 01 01 01 01 01
2	00011749	00000849	HIGHLOW(3)	0041A140	00 00 00 00 00 00 00
3	0001179F	0000089F	HIGHLOW(3)	0041C002	01 01 01 01 01 01 01
4	000117C8	000008C8	HIGHLOW(3)	0041B170	IAT Think of "ucrtbased
5	0001182F	0000082F	HIGHLOW(3)	0041C015	01 00 00 00 00 00 00

X64:

和X86的区别只是 1010(10) 的类型区别, 和ImageBase的区别罢了

基本没变

ImageBase变成8字节

十进制的10

110

Index	Section	RVA	Items
1	3 ("rdata")	00019000	Ch / 12d
2	3 ("rdata")	0001A000	10h / 16d
3	8 ("rdata")	00022000	6h / 6d

Index	RVA	Offset	Type	Far Address	Data Interpretation
1	00019110	00008110	DIR64 (10)	0000000140011FA0	48 83 EC 2
2	00019440	00008440	DIR64 (10)	0000000140011EA0	48 83 EC 2
3	00019550	00008550	DIR64 (10)	0000000140011F80	48 83 EC 2
4	000198C8	000088C8	DIR64 (10)	00000001400198B0	"_ArgList"
5	00019C08	00008C08	DIR64 (10)	0000000140019BC0	"H" (UNIC

## 2.导入表X86/X64

D190h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....	
D1A0h:	40 05 02 00 00 00 00 00 00 00 00 00 6A 08 02 00	.....j...	
D1B0h:	50 01 02 00 E0 05 02 00 00 00 00 00 00 00 00 00	P...à.....	
D1C0h:	1C 0B 02 00 F0 01 02 00 F0 03 02 00 00 00 00 00	...δ...δ.....	
D1D0h:	00 00 00 00 42 0D 02 00 00 00 02 00 00 00 00 00	...B.....	
D1E0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....	
D1F0h:	9C 0C 02 00 00 00 00 00 40 0B 02 00 00 00 00 00	æ...è.....	
D200h:	54 0B 02 00 00 00 00 00 66 0B 02 00 00 00 00 00	T.....	
D210h:	7C 0B 02 00 00 00 00 00 92 0B 02 00 00 00 00 00	.....	
D220h:	A6 0B 02 00 00 00 00 00 C0 0B 02 00 00 00 00 00	;.....A.....	
D230h:	30 0D 02 00 00 00 00 00 22 0D 02 00 00 00 00 00	0....."	
D240h:	12 0D 02 00 00 00 00 00 00 0D 02 00 00 00 00 00	.....	
D650h:	57 00 31 00 5F 5F 76 63 72 74 5F 46 6F 61 64 4C	W.I. VCRT Load	
D660h:	69 62 72 61 72 79 45 78 57 00 56 43 52 55 4E 54	libraryExW.VCRUNT	
D670h:	49 4D 45 31 34 30 44 2E 64 6C 6C 00 B5 00 5F 5F	IME140D.dll.5.	
D680h:	61 63 72 74 5F 69 6F 62 5F 66 75 6E 63 00 5C 00	acrt_iob_func.\.	
D690h:	5F 5F 73 74 64 69 6F 5F 63 6F 6D 6D 6F 6E 5F 76	_stdio common v	
D6A0h:	66 70 72 69 6E 74 66 00 04 00 5F 43 72 74 44 62	fprintf..._CrtDb	
D6B0h:	67 52 65 70 6F 72 74 00 05 00 5F 43 72 74 44 62	gReport..._CrtDb	
D6C0h:	67 52 65 70 6F 72 74 57 00 00 C3 02 5F 73 65 68	gReportW..._A._seh	

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD Characteristics; // 0 for terminating null import descriptor
        DWORD OriginalFirstThunk; // 包含指向IMAGE_THUNK_DATA（输入名称表）结构的数组
    };
    DWORD TimeDateStamp; // 当可执行文件不与被输入的DLL进行绑定时，此字段为0
    DWORD ForwarderChain; // 第一个被转向的API的索引
    DWORD Name; // 指向被输入的DLL的ASCII字符串的RVA
    DWORD FirstThunk; // 指向输入地址表（IAT）的RVA，IAT是一个IMAGE_THUNK_DATA结构的数组
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```

```
/*
第一个 OriginalFirstThunk 是 导入名称表RVA
最后一个 FirstThunk 是 导入地址表RVA
*/
typedef struct _IMAGE_THUNK_DATA64 {
    union {
        ULONGLONG ForwarderString; // PBYTE
        ULONGLONG Function; // PDWORD
        ULONGLONG Ordinal;
        ULONGLONG AddressOfData; // PIMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA64;
typedef IMAGE_THUNK_DATA64 * PIMAGE_THUNK_DATA64;

typedef struct _IMAGE_THUNK_DATA32 {
    union {
        DWORD ForwarderString; // PBYTE
        DWORD Function; // PDWORD
        DWORD Ordinal;
        DWORD AddressOfData; // PIMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA32;

/*
当IMAGE_THUNK_DATA64 的最高位为0,代表序号导入，最高位为1，代表名称导入
*/
指向这样的两个结构。
struct _IMAGE_IMPORT_BY_NAME
2 {
3     0x00 WORD Hint; //1字节
```

```

4     0x02 BYTE Name[1];           //n字节 函数名称字符串
5 };

```

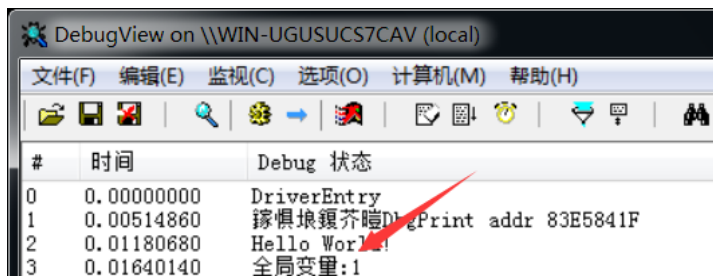
### 3.内存注入隐藏坑点

```

/*
1. 被注入的驱动 不能操作驱动对象 和 驱动路径。
2. 重定位块的移动，不能指针移动，只能大小。
3. 需要遍历所有内核模块的导出表获取函数地址。
4. IAT和重定位修复时，X86和X64有差异
5. X86和X64有NT头的大小差异
*/

```

### 4.x86内存注入代码



#### 1.字符串小转大

```

char* CharToUpper(char* wstr, BOOLEAN isAllocateMemory)
{
    char* ret = NULL;

    if (isAllocateMemory)
    {
        int len = strlen(wstr) + 2;
        ret = ExAllocatePool(PagedPool, len);
        memset(ret, 0, len);
        memcpy(ret, wstr, len - 2);
    }
    else
    {
        ret = wstr;
    }
    _strupr(ret);
    return ret;
}

```

#### 2.遍历内核模块地址和大小

```

typedef struct _RTL_PROCESS_MODULE_INFORMATION {
    HANDLE Section;           // Not filled in
    PVOID MappedBase;
    PVOID ImageBase;
    ULONG ImageSize;
    ULONG Flags;
    USHORT LoadOrderIndex;
    USHORT InitOrderIndex;
    USHORT LoadCount;
    USHORT OffsetToFileName;
}

```



```

    UCHAR    FullPathName[256];
} RTL_PROCESS_MODULE_INFORMATION, * PRTL_PROCESS_MODULE_INFORMATION;

typedef struct _RTL_PROCESS_MODULES {
    ULONG    NumberOfModules;
    RTL_PROCESS_MODULE_INFORMATION Modules[1]; // 柔性数组
} RTL_PROCESS_MODULES, * PRTL_PROCESS_MODULES;

typedef enum _SYSTEM_INFORMATION_CLASS {
    SystemBasicInformation,
    SystemProcessorInformation, // obsolete...delete
    SystemPerformanceInformation,
    SystemTimeOfDayInformation,
    SystemPathInformation,
    SystemProcessInformation,
    SystemCallCountInformation,
    SystemDeviceInformation,
    SystemProcessorPerformanceInformation,
    SystemFlagsInformation,
    SystemCallTimeInformation,
    SystemModuleInformation, // 模块信息
    SystemLocksInformation,
    SystemStackTraceInformation,
    SystemPagedPoolInformation,
    SystemNonPagedPoolInformation,
    SystemHandleInformation,
    SystemObjectInformation,
    SystemPageFileInformation,
    SystemVdmInstemulInformation,
    SystemVdmBopInformation,
    SystemFileCacheInformation,
    SystemPoolTagInformation,
    SystemInterruptInformation,
    SystemDpcBehaviorInformation,
    SystemFullMemoryInformation,
    SystemLoadGdiDriverInformation,
    SystemUnloadGdiDriverInformation,
    SystemTimeAdjustmentInformation,
    SystemSummaryMemoryInformation,
    SystemMirrorMemoryInformation,
    SystemPerformanceTraceInformation,
    SystemObsolete0,
    SystemExceptionInformation,
    SystemCrashDumpStateInformation,
    SystemKernelDebuggerInformation,
    SystemContextSwitchInformation,
    SystemRegistryQuotaInformation,
    SystemExtendServiceTableInformation,
    SystemPrioritySeperation,
    SystemVerifierAddDriverInformation,
    SystemVerifierRemoveDriverInformation,
    SystemProcessorIdleInformation,
    SystemLegacyDriverInformation,
    SystemCurrentTimeZoneInformation,
    SystemLookasideInformation,
    SystemTimeslipNotification,
    SystemSessionCreate,
    SystemSessionDetach,

```

```

SystemSessionInformation,
SystemRangeStartInformation,
SystemVerifierInformation,
SystemVerifierThunkExtend,
SystemSessionProcessInformation,
SystemLoadGdiDriverInSystemSpace,
SystemNumaProcessorMap,
SystemPrefetcherInformation,
SystemExtendedProcessInformation,
SystemRecommendedSharedDataAlignment,
SystemComPlusPackage,
SystemNumaAvailableMemory,
SystemProcessorPowerInformation,
SystemEmulationBasicInformation,
SystemEmulationProcessorInformation,
SystemExtendedHandleInformation,
SystemLostDelayedWriteInformation,
SystemBigPoolInformation,
SystemSessionPoolTagInformation,
SystemSessionMappedViewInformation,
SystemHotpatchInformation,
SystemObjectSecurityMode,
SystemWatchdogTimerHandler,
SystemWatchdogTimerInformation,
SystemLogicalProcessorInformation,
SystemWow64SharedInformation,
SystemRegisterFirmwareTableInformationHandler,
SystemFirmwareTableInformation,
SystemModuleInformationEx,
SystemVerifierTriageInformation,
SystemSuperfetchInformation,
SystemMemoryListInformation,
SystemFileCacheInformationEx,
MaxSystemInfoClass // MaxSystemInfoClass should always be the last enum
} SYSTEM_INFORMATION_CLASS;

//返回值为模块的大小
ULONG_PTR QuerySysModule(char* MoudleName, _Out_opt_ ULONG_PTR* module)
{
    RTL_PROCESS_MODULES info;
    ULONG retPro = NULL;
    ULONG_PTR moduleSize = 0;

    NTSTATUS ststas = ZwQuerySystemInformation(SystemModuleInformation, &info,
sizeof(info), &retPro);
    char* moduleUper = CharToUper(MoudleName, TRUE);

    if (ststas == STATUS_INFO_LENGTH_MISMATCH)
    {
        //申请长度
        ULONG len = retPro + sizeof(RTL_PROCESS_MODULES);
        PRTL_PROCESS_MODULES mem =
(PRTL_PROCESS_MODULES)ExAllocatePool(PagedPool, len);
        memset(mem, 0, len);
        ststas = ZwQuerySystemInformation(SystemModuleInformation, mem, len,
&retPro);

        if (!NT_SUCCESS(ststas))

```

```

    {
        ExFreePool(moduleUpper);
        ExFreePool(mem);
        return 0;
    }
    //开始查询
    if (strstr(MoudleName, "ntkrnlpa.exe") || strstr(MoudleName,
"ntoskrnl.exe"))
    {
        PRTL_PROCESS_MODULE_INFORMATION ModuleInfo = &(mem->Modules[0]);
        *module = ModuleInfo->ImageBase;
        moduleSize = ModuleInfo->ImageSize;
    }
    else
    {
        for (int i = 0; i < mem->NumberOfModules; i++)
        {
            PRTL_PROCESS_MODULE_INFORMATION processModule = &mem-
>Modules[i];
            CharToUpper(processModule->FullPathName, FALSE);
            if (strstr(processModule->FullPathName, moduleUpper))
            {
                if (module)
                {
                    *module = processModule->ImageBase;
                }
                moduleSize = processModule->ImageSize;
                break;
            }
        }
    }
    ExFreePool(mem);
}
ExFreePool(moduleUpper);
return moduleSize;
}

```

### 3.拷贝MZ,NT,节表

```

void CopyImageHeader(PUCHAR fileBuffer, PUCHAR imageBuffer)
{
    PIMAGE_DOS_HEADER pDos = (PIMAGE_DOS_HEADER)fileBuffer;
    PIMAGE_NT_HEADERS32 pNts = (PIMAGE_NT_HEADERS32)(fileBuffer + pDos-
>e_lfanew);
    PIMAGE_SECTION_HEADER pSection = IMAGE_FIRST_SECTION(pNts);

    memcpy(imageBuffer, fileBuffer, pNts->OptionalHeader.SizeOfHeaders);
}

void CopyImageSection(PUCHAR fileBuffer, PUCHAR imageBuffer)
{
    PIMAGE_DOS_HEADER pDos = (PIMAGE_DOS_HEADER)fileBuffer;
    PIMAGE_NT_HEADERS32 pNts = (PIMAGE_NT_HEADERS32)(fileBuffer + pDos-
>e_lfanew);
    PIMAGE_SECTION_HEADER pSection = IMAGE_FIRST_SECTION(pNts);

    for (int i = 0; i < pNts->FileHeader.NumberOfSections; i++)

```

```

{
    memcpy(imageBuffer + pSection->VirtualAddress,
           fileBuffer + pSection->PointerToRawData, pSection->SizeOfRawData);
    pSection++;
}
}

```

#### 4.修复重定位

```

void ReUpdateReloc(PUCHAR image)
{
    PIMAGE_DOS_HEADER pDos = (PIMAGE_DOS_HEADER)image;
    PIMAGE_NT_HEADERS32 pImageNts = (PIMAGE_NT_HEADERS32)(image + pDos->e_lfanew);
    PIMAGE_DATA_DIRECTORY pDir = &pImageNts->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
    if (pDir->Size)
    {
        PIMAGE_BASE_RELOCATION relocation = (PIMAGE_BASE_RELOCATION)(image + pDir->VirtualAddress);
        while (relocation->SizeOfBlock)
        {
            PIMAGE_RELOC pImageReloc = (PIMAGE_RELOC)((ULONG)relocation + sizeof(IMAGE_BASE_RELOCATION));
            //计算有多少个PIMAGE_RELOC结构
            ULONG blockNumber = (relocation->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) / sizeof(USHORT);
            for (int i = 0; i < blockNumber; i++)
            {
                if (pImageReloc[i].Type == IMAGE_REL_BASED_HIGHLOW)
                {
                    //修复的32位
                    PULONG address = (image + pImageReloc[i].Offset + relocation->VirtualAddress);
                    ULONG value = *address - pImageNts->OptionalHeader.ImageBase + (ULONG)image;
                    *address = value;
                }
            }
            //继续下一次
            relocation = (PIMAGE_BASE_RELOCATION)((ULONG)relocation + relocation->SizeOfBlock);
        }
        pImageNts->OptionalHeader.ImageBase = (ULONG)image;
    }
}

```

#### 5.查找内核模块导出表函数地址

PS:借鉴火绒注入

```

ULONG_PTR ExportTableFuncByName(char* pData, char* funcName)
{
    PIMAGE_DOS_HEADER pHead = (PIMAGE_DOS_HEADER)pData;
    PIMAGE_NT_HEADERS pNt = (PIMAGE_NT_HEADERS)(pData + pHead->e_lfanew);
    int numberRvaAndSize = pNt->OptionalHeader.NumberOfRvaAndSizes;
}

```

```

    PIMAGE_DATA_DIRECTORY pDir = (PIMAGE_DATA_DIRECTORY)&pnt-
>OptionalHeader.DataDirectory[0];
    PIMAGE_EXPORT_DIRECTORY pExport = (PIMAGE_EXPORT_DIRECTORY)(pData + pDir-
>VirtualAddress);
    ULONG_PTR funcAddr = 0;
    for (int i = 0; i < pExport->NumberOfNames; i++)
    {
        int* funcAddress = pData + pExport->AddressOfFunctions;
        int* names = pData + pExport->AddressOfNames;
        short* fh = pData + pExport->AddressOfNameOrdinals;
        int index = -1;
        char* name = pData + names[i];
        if (strcmp(name, funcName) == 0)
        {
            index = fh[i];
        }
        if (index != -1)
        {
            funcAddr = pData + funcAddress[index];
            break;
        }
    }
    if (!funcAddr)
    {
        KdPrint(("没有找到函数%s\r\n", funcName));
    }
    else
    {
        KdPrint(("找到函数%s addr %p\r\n", funcName, funcAddr));
    }
    return funcAddr;
}

```

## 6.修复导入表

```

VOID ReIAT(PUCHAR image)
{
    //修复导入表
    PIMAGE_DOS_HEADER pDos = (PIMAGE_DOS_HEADER)image;
    PIMAGE_NT_HEADERS32 pImageNts = (PIMAGE_NT_HEADERS32)(image + pDos-
>e_lfanew);

    PIMAGE_DATA_DIRECTORY pImport = &pImageNts-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];
    PIMAGE_IMPORT_DESCRIPTOR pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)(image +
pImport->VirtualAddress);
    PIMAGE_THUNK_DATA pNames = NULL;
    PIMAGE_THUNK_DATA pFuncP = NULL;

    BOOLEAN isFiled = FALSE;

    for (; pImportDesc->Name; pImportDesc++)
    {
        PCHAR libName = pImportDesc->Name + image;
        ULONG_PTR module = 0, moduleSize = 0;
        if (QuerySysModule(libName, &module))

```

```

    {
        pNames = (PIMAGE_THUNK_DATA)(image + pImportDesc-
>OriginalFirstThunk);
        pFuncP = (PIMAGE_THUNK_DATA)(image + pImportDesc->FirstThunk);

        for (; pNames->u1.ForwarderString; pNames++, pFuncP++)
        {
            PIMAGE_IMPORT_BY_NAME byName = (PIMAGE_IMPORT_BY_NAME)(pNames-
>u1.AddressOfData + image);
            ULONG_PTR func = ExportTableFuncByName((char*)module, byName-
>Name);

            if (func)
            {
                pFuncP->u1.Function = func;
            }
            else
            {
                isFiled = TRUE;
                break;
            }
            //image + pNames->
        }
    }
}
}
}

```

## 7.调用驱动入口，修复配置项

```

NTSTATUS LoadDriver(PUCHAR buffer)
{
    PIMAGE_DOS_HEADER pDos = (PIMAGE_DOS_HEADER)buffer;
    PIMAGE_NT_HEADERS32 pNts = (PIMAGE_NT_HEADERS32)(buffer + pDos->e_lfanew);
    PIMAGE_SECTION_HEADER pSection = IMAGE_FIRST_SECTION(pNts);

    PHYSICAL_ADDRESS phyLow;
    PHYSICAL_ADDRESS phyhei;
    phyLow.QuadPart = 0;
    phyhei.QuadPart = -1;
    PUCHAR image = NULL;

    int count = 3;
    do
    {
        image = MmAllocateContiguousMemorySpecifyCache(pNts-
>OptionalHeader.SizeOfImage
        , phyLow, phyhei, phyLow, MmCached);
        if (image) break;
        --count;
    } while (count);

    if (!image) return STATUS_UNSUCCESSFUL;

    CopyImageHeader(buffer, image);
    CopyImageSection(buffer, image);
}

```

```

ReUpdateReloc(image);
ReIAT(image);

pDos = (PIMAGE_DOS_HEADER)image;
pNts = (PIMAGE_NT_HEADERS32)(image + pDos->e_lfanew);

//获取到入口点
DriverEntryCallback EntryCall = (DriverEntryCallback)(pNts->OptionalHeader.AddressOfEntryPoint + image);
ULONG retSize = 0;

PIMAGE_LOAD_CONFIG_DIRECTORY load = RtlImageDirectoryEntryToData(image,
TRUE, 10, &retSize);
*(PULONG_PTR)load->SecurityCookie += 10;
EntryCall(NULL, NULL);
return STATUS_SUCCESS;
}

```

## 9.修复异常

---

## 10.回调通信

---