# Windows文件透明加密minifilter

## 基础篇知识总结

### 常用函数

Ex系列函数

ExAllocatePool: 内存分配

ExFreePool: 内存释放

ExAcquireFastMutex: 获取一个快速互斥体，用于多线程环境下的同步

ExReleaseFastMutex: 释放一个快速互斥体

ExRaiseStatus: 抛出一个异常，用于从代码很深的地方直接报错

Zw系列函数：

ZwCreateFile: 打开文件实际上也用于打开一个设备

ZwWriteFile: 写文件，用于发写请求

ZwReadFile: 读文件

ZwQueryDirectoryFile: 目录查询

ZwDeviceIoControlFile: 发出设备控制请求

ZwCreateKey: 打开一个注册表键

ZwQueryValueKey: 读取一个注册表的值

Rtl系列函数

RtlInitUnicodeString: 初始化一个字符串

RtlCopyUnicodeString: 拷贝字符串

RtlAppendUnicodeString: 追加字符串

RtlStringCbPrintf: 设置字符串打印格式

RtlCopyMemory: 内存数据块拷贝

RtlMoveMemory: 内存数据块移动

RtlZeroMemory: 内存数据清零

RtlCompareMemory: 比较内存

RtlGetVersion: 获得Windows的版本

Io系列函数

IoCreateFile: 打开文件，比ZwCreateFile要更加底层

IoCreateDevice: 生成一个设备对象

IoCallDriver: 发送请求，实际上这个函数可能是IofCallDriver的一个别名。WindowsIO管理器调用这个函数把不同的IRP发送到不同的设备。

IoCompleteRequest: 完成请求。通知IO管理器这个IRP已经完成了

IoCopyCurrentIrpStackLocationToNext: 将当前IRP栈空间内容拷贝到下一个栈空间

IoSkipCurrentIrpStackLoactionToNext: 跳过当前的IRP栈空间

IoGetCurrentIrpStackLocation: 获得IRP的当前栈空间指针

掌握Windows内核的体系结构，熟悉各种操作系统中常见的基本知识在Windows中的具体体现，掌握系统底层的编程和调试技巧。

Windows驱动的开发模型

从KDM到WDM再到WDF都是一脉相承的，

内核编程的主要调用源，入口函数DriverEntry和卸载函数DriverUnload。

各种分发函数包括普通分发函数和快速IO分发函数。

处理请求时设置的完成函数，请求完成后被系统调用的回调函数。

其他回调函数如各类NDIS驱动程序的特征函数

考虑处理函数的可重入性和运行中断级Dispatch，和Passive，Dispatch级别较高，必须保证Windows的代码都运行在规范的中断级上。需要降低中断级可以生成一个线程专门去执行Passive级代码

#pragma alloc\_text(INIT, DriverEntry)

#pragma alloc\_text这个宏是用来指定某个函数的可执行代码编译出来后在sys文件中的位置

INIT初始化后即被释放不占用内存

PAGE位于可以进行分页交换的内存空间，内存紧张是可以被交换到硬盘上以节省内存。

若未用上述预编译指令处理，则代码默认位于PAGELK节，加载后位于不可分页交换的内存空间中。

注意放在PAGE节的函数不可以再Dispatch级调用，缺页中断不能在Dispatch级完成PAGED\_CODE()进行测试

### 字符串

传统字符串函数以str开头的标准C库中的字符串处理函数，这种字符串不安全有溢出漏洞，仅仅用一个“\0”字符来表明这个字符串的结束，一旦没有空结束的字符串，程序就会陷入崩溃。

Windows内核用Unicode编码.UNICODE\_STRING并不保证Buffer指向字符串是以空结束的，用微软提供的Rtl函数来操作字符串。

字符串初始化

UNICODE\_STRING str = RTL\_CONSTANT\_STRING(L“my first string!”);

或者：

UNICODE\_STRING str;

RtlInitUnicodeString(&str,L”my first string!”);

字符串的拷贝：

UNICODE\_STRING dst; // 目标字符串

WCHAR dst\_buf[256]; // 我们现在还不会分配内存，所以先定义缓冲区

UNICODE\_STRING src = RTL\_CONST\_STRING(L”My source string!”);

// 把目标字符串初始化为拥有缓冲区长度为256的UNICODE\_STRING空串。

RtlInitEmptyString(dst,dst\_buf,256\*sizeof(WCHAR));

RtlCopyUnicodeString(&dst,&src); // 字符串拷贝！

字符串的打印

RtlStringCbPrintfW需要包含头文件ntstrsafe.h，类似于swprintf

// 字符串初始化为空串。缓冲区长度为512\*sizeof(WCHAR)

RtlInitEmptyString(dst,dst\_buf,512\*sizeof(WCHAR));

// 调用RtlStringCbPrintfW来进行打印

status = RtlStringCbPrintfW(

dst->Buffer,L”file path = %wZ file size = %d \r\n”，

&file\_path,file\_size);

// 这里调用wcslen没问题，这是因为RtlStringCbPrintfW打印的

// 字符串是以空结束的。

dst->Length = wcslen(dst->Buffer) \* sizeof(WCHAR);

UNICODE\_STRING类型的指针，用%wZ输出到缓冲区中，因为不能保证字符串是以空位结束避免使用%ws。

DbgPrint函数输出Unicode字符串时，当前的中断级别PASSIVE\_LEVEL,可以通过KeGetCurrentIrql函数获取当前中断级别。

### 内存和链表

内存的分配和释放

dst.Buffer =

(PWCHAR)ExAllocatePoolWithTag(NonpagedPool,src->Length,MEM\_TAG);

ExAllocatePoolWithTag内核分配内存NonpagedPool表明分配的内存是锁定内存。不会被分页交换到硬盘中，第三个是内存分配标识。

内存分配标识用于检测内存泄漏，PagedPool是可分页内存。

ExFreePool释放分配的内存。

LIST\_ENTRY是一个双向链表结构，它总是在使用的时候被插入到已有的数据结构中。将信息串成链表，进行管理，LIST\_ENTR作为成员放在开头是最简单的做法

比如MS的很多结构喜欢开头是结构的长度，所以在通过LIST\_ENTRY结构的地址获取所在节点的地址是有个地址偏移的过程。

典型的遍历过程如下：

for(p = my\_list\_head.Flink; p != &my\_list\_head.Flink; p = p->Flink)

{

PMY\_FILE\_INFOR elem =

CONTAINING\_RECORD(p,MY\_FILE\_INFOR, list\_entry);

// 在这里做需要做的事…

}

}

其中CONTAINING\_RECORD使一个WDK已经存在的宏，通过一个LIST\_ENTRY结构的指针，找到这个结构所在的节点的指针

#define CONTAINING\_RECORD(address, type, field) ((type \*)( \

(PCHAR)(address) - \

(ULONG\_PTR)(&((type \*)0)->field)))

### 自旋锁

链表之类的结构总是设计多线程同步问题，最简单的就是自旋锁。

KSPIN\_LOCK my\_spin\_lock;

KeInitializeSpinLock(&my\_spin\_lock);

对自旋锁的初始化如上

在KeAcquireSpinLock和KeReleaseSpinLock之间的代码只是单线程执行，其他线程会停留在KeAcquireSpinLock等候，同时KeAcquireSpinLock会提高当前中断级别。

KIRQL irql;

KeAcquireSpinLock(&my\_spin\_lock,&irql);

// To do something …

KeReleaseSpinLock(&my\_spin\_lock,irql);

所有线程公用一个锁，锁才有意义。

LIST\_ENTRY加锁操作：

插入一个节点：

ExInterlockedInsertHeadList(

&my\_list\_head,

(PLIST\_ENTRY)& my\_file\_infor,

&my\_list\_lock);

增加KSPIN\_LOCK作为参数。

删除一个节点：

my\_file\_infor = ExInterlockedRemoveHeadList (

&my\_list\_head,

&my\_list\_lock);

### 文件操作

内核不接受一个字符串，接收一个OBJECT\_ATTRIBUTES结构，这个结构总是InitializeObjectAtttibutes初始化，文档中OBJECT\_ATTRIBUTES的结构没有公开。

VOID InitializeObjectAttributes(

OUT POBJECT\_ATTRIBUTES InitializedAttributes,

IN PUNICODE\_STRING ObjectName,

IN ULONG Attributes,

IN HANDLE RootDirectory,

IN PSECURITY\_DESCRIPTOR SecurityDescriptor);

下面的函数用于打开一个文件：

NTSTATUS ZwCreateFile(

OUT PHANDLE FileHandle,

IN ACCESS\_MASK DesiredAccess,

IN POBJECT\_ATTRIBUTES ObjectAttribute,

OUT PIO\_STATUS\_BLOCK IoStatusBlock,

IN PLARGE\_INTEGER AllocationSize OPTIONAL,

IN ULONG FileAttributes,

IN ULONG ShareAccess,

IN ULONG CreateDisposition,

IN ULONG createOptions,

IN PVOID EaBuffer OPTIONAL,

IN ULONG EaLength);

文件读：

NTSTATUS

ZwReadFile(

IN HANDLE FileHandle,

IN HANDLE Event OPTIONAL,

IN PIO\_APC\_ROUTINE ApcRoutine OPTIONAL,

IN PVOID ApcContext OPTIONAL,

OUT PIO\_STATUS\_BLOCK IoStatusBlock,

OUT PVOID Buffer,

IN ULONG Length,

IN PLARGE\_INTEGER ByteOffset OPTIONAL,

IN PULONG Key OPTIONAL);

写函数与读函数形式相同，只是输入和输出方向不同。

### 注册表操作

|  |  |
| --- | --- |
| 应用编程中对应的子键 | 驱动编程中的路径写法 |
| HKEY\_LOCAL\_MACHINE | \Registry\Machine |
| HKEY\_USERS | \Registry\User |
| HKEY\_CLASSES\_ROOT | 没有对应的路径 |
| HKEY\_CURRENT\_USER | 没有简单的对应路径，但是可以求得 |

应用程序和驱动程序不同的是，应用程序总是由某个当前用户启动的，而驱动程序与用户无关。

NTSTATUS

ZwOpenKey(

OUT PHANDLE KeyHandle,

IN ACCESS\_MASK DesiredAccess,

IN POBJECT\_ATTRIBUTES ObjectAttributes

);

接收OBJECT\_ATTRIBUTES结构输出注册建的HANDLE

读注册表：

NTSTATUS ZwQueryValueKey(

IN HANDLE KeyHandle,

IN PUNICODE\_STRING ValueName,

IN KEY\_VALUE\_INFORMATION\_CLASS KeyValueInformationClass,

OUT PVOID KeyValueInformation,

IN ULONG Length,

OUT PULONG ResultLength

);

注册表的写：

NTSTATUS ZwSetValueKey(

IN HANDLE KeyHandle,

IN PUNICODE\_STRING ValueName,

IN ULONG TitleIndex OPTIONAL,

IN ULONG Type,

IN PVOID Data,

IN ULONG DataSize

);

### 时间与定时器

Win32开发有一个GetTickCount()，函数返回自系统启动之后经历的毫秒数，驱动开发中有一个对应的函数KeQueryTickCount()，

VOID

KeQueryTickCount(

OUT PLARGE\_INTEGER TickCount

);

TickCount不是简单的毫秒数，需要下面一个函数计算时间：

ULONG

KeQueryTimeIncrement(

);

获取一个间隔，可以根据下面的例子，获得毫秒数

void MyGetTickCount (PULONG msec)

{

LARGE\_INTEGER tick\_count;

ULONG myinc = KeQueryTimeIncrement();

KeQueryTickCount(&tick\_count);

tick\_count.QuadPart \*= myinc;

tick\_count.QuadPart /= 10000;

\*msec = tick\_count.LowPart;

}

如何获取系统时间？

KeQuerySystemTime()得到一个格林威治时间，之后可以用ExSystemTimeToLocalTime()转换成当地时间，

VOID

KeQuerySystemTime(

OUT PLARGE\_INTEGER CurrentTime

);

VOID

ExSystemTimeToLocalTime(

IN PLARGE\_INTEGER SystemTime,

OUT PLARGE\_INTEGER LocalTime

);

再通过RtlTimeToTimeFields()转换为TIME\_FIELDS。

VOID

RtlTimeToTimeFields(

IN PLARGE\_INTEGER Time,

IN PTIME\_FIELDS TimeFields

);

定时器

KeSetTimer()

BOOLEAN

KeSetTimer(

IN PKTIMER Timer, // 定时器

IN LARGE\_INTEGER DueTime, // 延后执行的时间

IN PKDPC Dpc OPTIONAL // 要执行的回调函数结构

);

其中定时器Timer和执行回调函数结构Dpc都必须初始化。

KTIMER my\_timer;

KeInitializeTimer(&my\_timer);

Dpc的初始化比较麻烦：

VOID

KeInitializeDpc(

IN PRKDPC Dpc,

IN PKDEFERRED\_ROUTINE DeferredRoutine,

IN PVOID DeferredContext

);

回调例程格式如下：

VOID

CustomDpc(

IN struct \_KDPC \*Dpc,

IN PVOID DeferredContext,

IN PVOID SystemArgument1,

IN PVOID SystemArgument2

);

要想定时反复执行，就必须在CustomDpc函数被调用时再次调用KeSetTimer来保证运行。

### 线程

系统线程所在的进程名为“System”，内核API原型如下：

NTSTATUS

PsCreateSystemThread(

OUT PHANDLE ThreadHandle,

IN ULONG DesiredAccess,

IN POBJECT\_ATTRIBUTES ObjectAttributes OPTIONAL,

IN HANDLE ProcessHandle OPTIONAL,

OUT PCLIENT\_ID ClientId OPTIONAL,

IN PKSTART\_ROUTINE StartRoutine,

IN PVOID StartContext);

启动函数原型：

VOID CustomThreadProc(IN PVOID context)

其中关闭句柄并不结束线程

驱动中睡眠函数：

NTSTATUS

KeDelayExecutionThread(

IN KPROCESSOR\_MODE WaitMode,

IN BOOLEAN Alertable,

IN PLARGE\_INTEGER Interval);

事件通知，内核中的事件是一个数据结构，这个结构的指针被当做一个参数传入一个等待函数中，如果事件不被设置，则等待函数不会返回，这个线程被阻塞。如果事件被设置，线程继续，常用于多线程之间的同步，一个线程等待另一个线程完成某事后才能做某事，可以用事件等待，另一个线程完成后设置事件。

OID

KeInitializeEvent(

IN PRKEVENT Event,

IN EVENT\_TYPE Type,

IN BOOLEAN State

);

// 定义一个事件

KEVENT event;

// 事件初始化

KeInitializeEvent(&event,SynchronizationEvent,TRUE);

……

// 事件初始化之后就可以使用了。在一个函数中，你可以等待某

// 个事件。如果这个事件没有被人设置，那就会阻塞在这里继续

// 等待。

KeWaitForSingleObject(&event,Executive,KernelMode,0,0);

……

// 这是另一个地方，有人设置这个事件。只要一设置这个事件，

// 前面等待的地方，将继续执行。

KeSetEvent(&event);

### 应用与内核通信

控制设备：配置，开启或关闭某些功能。CDO

NTSTATUS

IoCreateDevice(

IN PDRIVER\_OBJECT DriverObject,

IN ULONG DeviceExtensionSize,

IN PUNICODE\_STRING DeviceName OPTIONAL,

IN DEVICE\_TYPE DeviceType,

IN ULONG DeviceCharacteristics,

IN BOOLEAN Exclusive,

OUT PDEVICE\_OBJECT \*DeviceObject

);

生成设备对象

符号链接：

设备对象可以没有名字，但是控制设备需要有一个名字，这样它才会被暴露出来，供其他程序打开与之通信。设备的名字可以在调用IoCreateDevice时指定。应用层是无法直接通过设备的名字来打开对象的，为此要建立一个暴露给应用层的符号链接。

IoCreateSymbolicLink(&symbl\_name, &device\_name);

符号链接总是以[\\Device\\??](file:///\\Device\\%3f%3f)的方式存在。

驱动入口与驱动对象。

NTSTATUS

DriverEntry (

IN PDRIVER\_OBJECT DriverObject,

IN PUNICODE\_STRING RegistryPath

)

{

NTSTATUS status = STATUS\_UNSUCCESSFUL;

return status;

}

DriverEntry()函数是驱动的入口，DRIVER\_OBJECT中有分发函数处理各种发到这个驱动的请求，所以编写一个驱动，实际上就是编写这些处理请求的分发函数。

分发函数的格式如下：

NTSTATUS MyDispatchFunction(PDEVICE\_OBJECT device,PIRP irp)

{

……

}

64位和32位内核开发差异。

WOW64子系统，兼容32位程序，当一个32位程序发起系统调用时，WOW64子系统拦截后先把指针长度换成合适的长度，再把系统请求，提交给内核。通常称为“thunking”。

PatchGuard可以避免系统内核损坏，一旦发现关键位置的数据或代码被篡改，系统就会被触发蓝屏。

## 开发篇

## 需求分析

文件加密的需求如下：

合法用户在合法环境下对文件是否加解密是无法感知的，及文件的加解密对用户是透明的，加解密的过程在内核中，由操作系统进行调用。

密钥由服务器管理，作为唯一加解密的标准。传递给合法用户使用。非法用户无法获得密钥。

对特定文件进行加解密操作，这个可在服务器进行文件类型设定。

文件打开时解密，保存时加密，文件内容由对称加密，采用快速加密算法，优化效率

文件保存时断电如何处理？初步设想：定时自动保存机制，这样断电后恢复到上一个保存点，此时对加密算法效率要求高。

文件已经加密，文件头部有加密标识，读取数据时要解密，修改数据保存要重新加密。

文件拷出无法读取。

文件带出指定网络环境无法读取。

文件上传至服务器也是密文。

文件在介质中的形式始终为密文

1. 机密文件自动透明加解密：

系统核心功能，在系统使用中，终端的工作组用户，只需要安装客户端程序（包括驱动），填写个人账号信息，点击开启保护，客户端会向服务器请求身份认证，并下载该工作组的加解密策略，然后启动驱动的拦截保护功能。最终的效果是，用户开启保护模式后（自动开启保护模式），使用机密进程访问机密文档，显示正常的明文，非机密进程访问机密文档，显示密文，未加密的文档被机密进程访问打开后会被加密，对于用户都是透明的，加解密的过程，用户不可感知。

1. 服务器加解密配置信息管理

加解密策略表和秘钥都是存放到服务器上的，服务器安装运行服务端程序，就可以管理企业的加解密配置信息，开启服务后，就可以向局域网的客户机提供身份验证，加解密策略和秘钥下载，客户机离线授权，特殊文件解密授权服务，提供部门分类服务，企业内部有多少个部门就可以在服务器建立多少个部门资料，每个部门都有自己的加解密策略表，和加解密秘钥，部门资料中存储该部门人员资料，只有该部门的人员才能获取该部门的加解密策略和秘钥，同一个部门的机器可以正常解密该部门的机密文档。管理者可以做到，在企业内部各部门之间的机密文档不能共享查看，独立化。

1. 终端用户离线授权

企业某些用户可能用笔记本办公，当他们离开工作网络后，系统客户端无法连接服务器，保护模式会关闭，这时候机密文档无法解密，但是员工出差，也是需要正常读取机密文档的，为此设计终端用户离线授权服务，需要离线工作的员工可以向管理部门申请离线授权密码，然后在在线保护模式下申请离线授权证书，证书上存储加解密策略和授权的离线时间，员工在客户端打开离线保护模式，这样就能在出差的时候正常读取机密文档。考虑到安全问题，离线证书使用安全的加密算法加密，并设置散列码校验，任何修改证书的行为都会导致证书失效。证书存储有客户授权时间，授权时间到达后，证书自动失效。证书存储有用户身份信息，只由申请的用户使用，拷贝到其他机器上会使校验失败，不能开启离线保护。

1. 文件加解密通道：

正常情况下，工作网络中的所有机密文档都是密文存储在磁盘上，只有机密进程能够读取，即使拷贝出去，文档也是无法解密的。有些时候，一些文档虽然是机密文档，但是由于正常业务原因，需要将其拷贝到外面的机器上并且能正常读取。考虑到这种情况，添加文件加解密通道功能，需要解密机密文档的客户需要向管理员请求文件加密授权密码，被解密的文档以明文的方式存储在Disk上，可以自由拷贝文件到外网机器，如果用户在工作中需要对某些机密文件也进行加密的话，可以使用加密通道，此时不涉及安全问题，不用授权。文件加解密通道设计的目的是为了让系统应对特殊情况，平常使用中不应该使用该功能，管理员应该保护好授权密码。

## 环境配置

### 准备工作

系统环境：win10 version 1607

开发工具：VS2015

驱动开发工具：[WDK10](https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit)（安装时，VS必须处于关闭状态）

WindowSDK: SDK10 必须和驱动版本，主机系统一样

VMware WorkStation 12 PRO

虚拟机系统：WIN10 version 1607

### 测试机内核调试

在测试机上以管理员打开CMD命令窗口，输入以下两个命令：

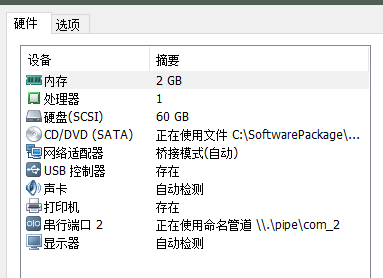
bcdedit /set {default} DEBUG YES（设置虚拟机环境可调试）

bcdedit /set TESTSIGNING ON（设置windbg可调试）

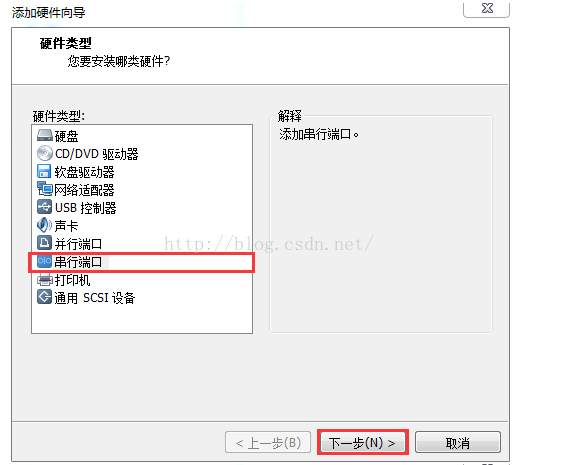
设置完重启测试机。

然后在VMware中添加一个串口设备：

这是我已经配置好的。



虚拟机中测试机必须要关机，按步骤如下：







后面的com\_1是可以自定义的名称。完成配置后就会多一个串行设备，因为打印机使用了端口1，所以我使用的是串行端口2.

接下来在测试机上设置串口调试，管理员打开CMD命令：

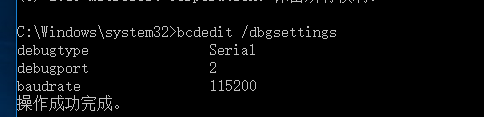
bcdedit /debug on（设置调试开启）

bcdedit /dbgsettings serial debugport:n baudrate:115200（设置调试连接一些参数，最后一个参数，比特率不可更改）

其中n是串口端口号，我使用的是端口2，所以将其换成2。重启测试机，在测试机的CMD中输入以下命令：

bcdedit /dbgsettings

出现以下信息，则说明配置成功：



接下来，记住在主机中先打开WINDBG，用以下命令：

cd C:\Program Files (x86)\Windows Kits\10\Debuggers\x64

**windbg -k com:pipe,port=\\.\pipe\com\_2,resets=0,reconnect**

在主机上**先打开windbg，后重启测试机，测试机在黑屏状态**，在windbg界面按组合键**CTRL+BREAK（可能要多次输入组合键）**，这样才会出现如下连接成功：



说明windbg内核测试环境搭建成功。

接下来搭建VS2015开发驱动环境。

第一步：

在主机中找到位于

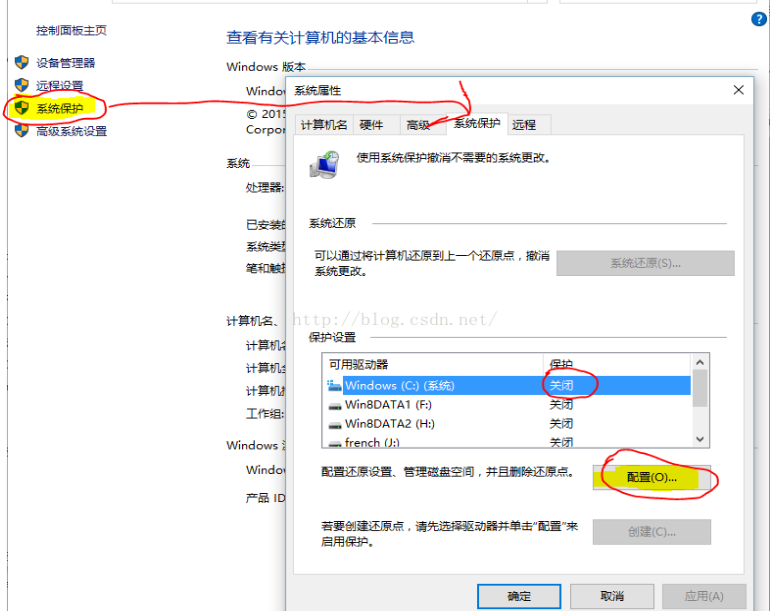
C:\Program Files (x86)\WindowsKits\10\Remote\x64\WDK Test Target Setup x64-x64\_en-us.msi

和C:\Program Files (x86)\WindowsKits\10\Remote\x86\WDK Test Target Setup x86-x86\_en-us.msi （直接拷贝到虚拟机中安装）

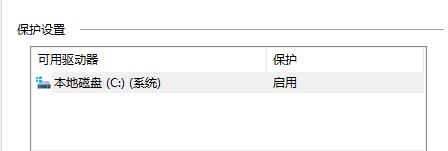
安装到测试机中。

第二步：

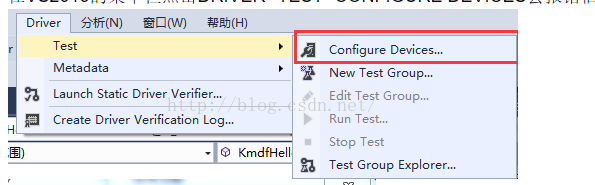
在测试机中开启系统保护如下图：

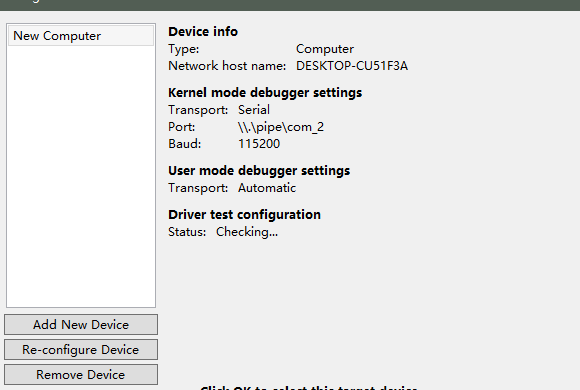


开启后，如下图：



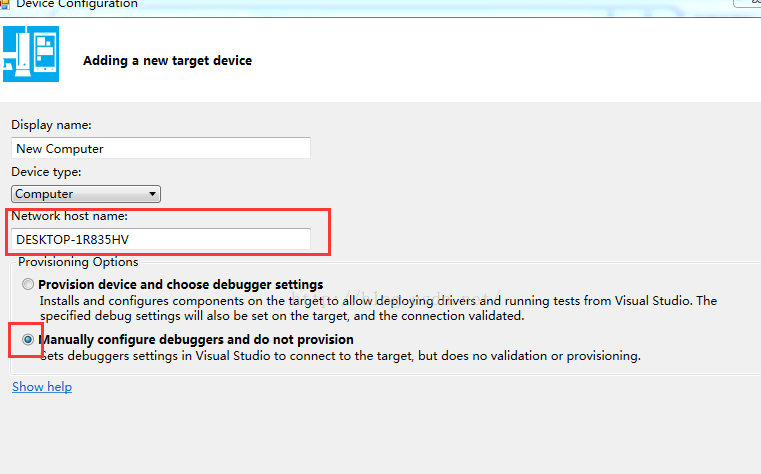
在主机VS2015菜单中选择 Driver->Test->Configure device点击Add New Device，





以上是我配置好的VS2015环境

在配置界面如下：

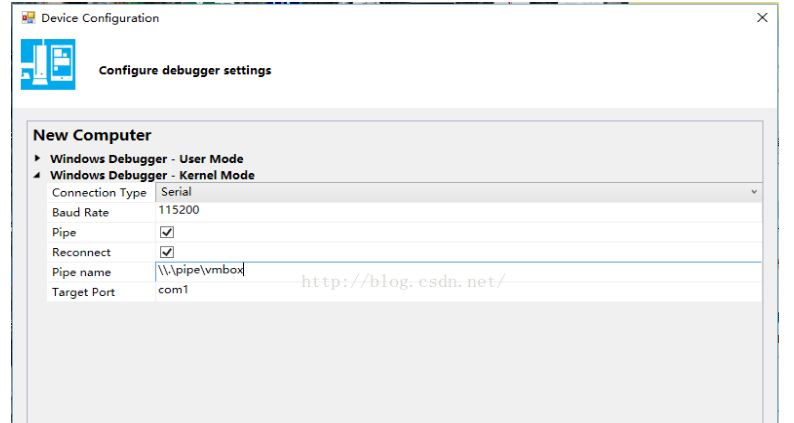


Network host name是测试机的名称，可以在测试机此电脑属性中找到。

Provison…是自动部署方式，个人建议选择这个。

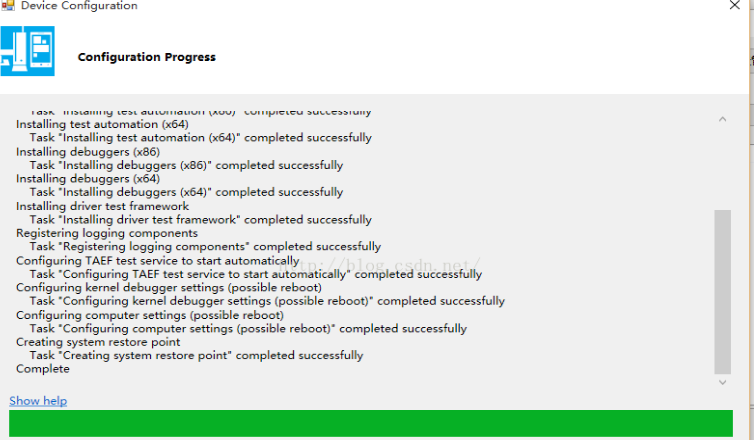
Manually…手动方式。

点击下一步出现连接类型，如下。

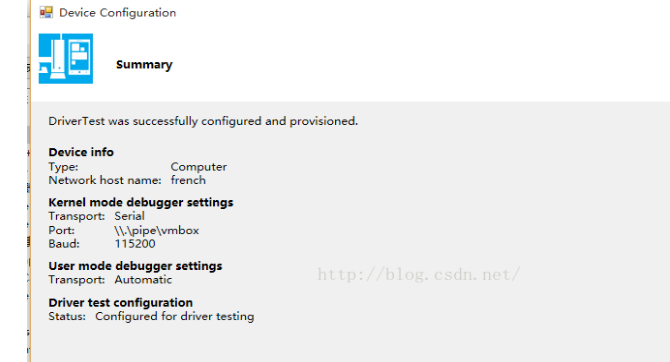


在Pipe name中我填写的是\\.\pipe\com\_2，Target Port中填写com2，点击下一步。

会在测试机进行相应的环境变量配置，期间测试目标主机会有多次重启。期间虚拟机，**不要有任何改动，虚拟机需要处于开启状态，如果动了虚拟机，可能后续操作无法进行，只能重新安装虚拟机系统**



设置完成后提示配置成功。



以上就是内核开发环境的配置。

## 具体设计

### HelloWorld字符串传入内核

在用户层程序下发一个字符串信息到内核中，内核中捕捉到这个字符串显示出来，如下图：

helloworld

Application

在内核中用DEBUGVIEW显示字符串

Kernel

### 信息拦截

在内核中对特定的程序打开的文件，监控这部分内存，显示内容。如下图：

Application

在此处捕获数据，对特定的应用监控

Kernel

disk

### 加密解密

在前面的基础上，加入加密和解密模块，对数据进行加解密，监控特定程序的读写操作。如下图所示:

Server

加密使用的KEY

Application

在应用层显示前和写入内存前，连接加解密模块

Kernel

Disk

### 缓冲备份管理

主要解决Back-up技术点，读取和写入异常时的恢复策略。如图所示

Application

异常中断时，Back-up文件起作用

Kernel

Back-up

Real-File

## 开发进度详细报告

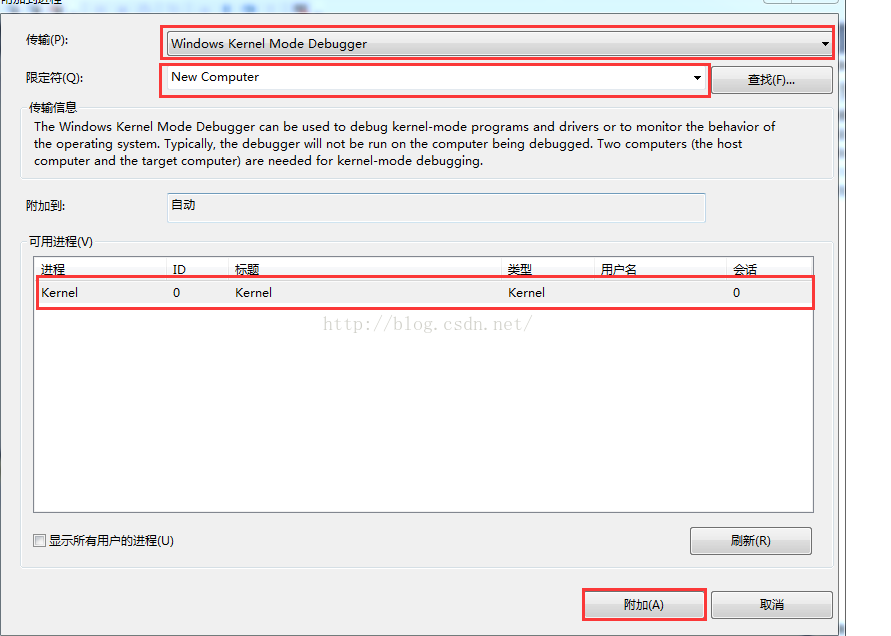
### 工具配置环境及使用

在VS2015中开发驱动和调试驱动的方式。

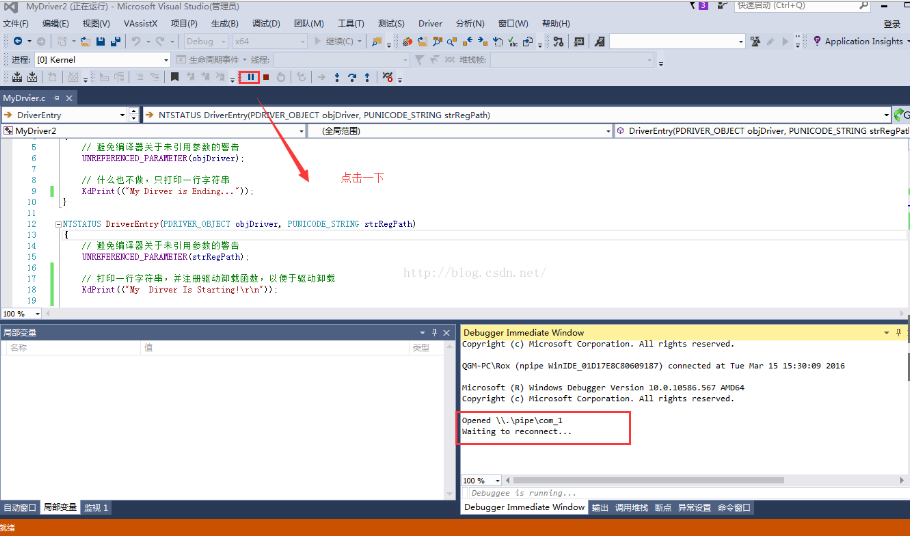
首先把虚拟机的内核附加到VS2015



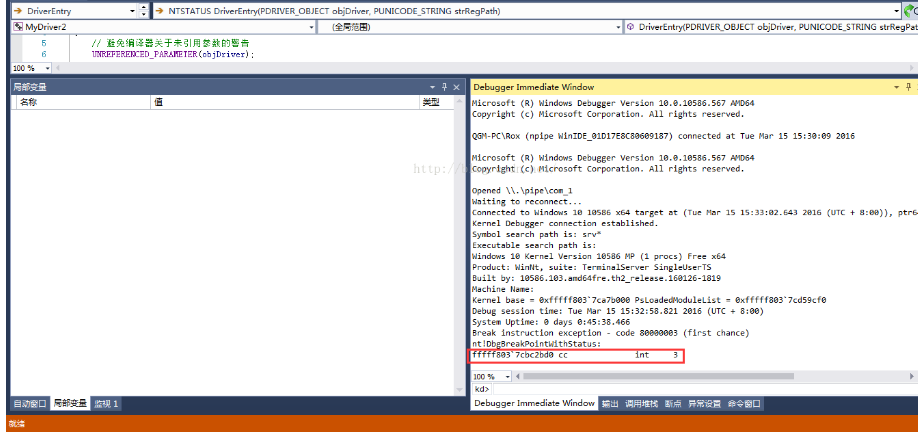
选中内核调试模式，选自己配置的那个电脑，选中下面的内核，



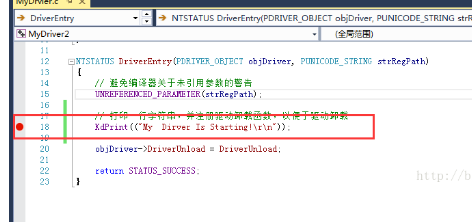
点击附加后，出现以下界面，点击一下，全部中断。



系统就断下了，虚拟机中鼠标已经动不了，系统处于挂起状态，见下图：



然后在代码中下断点，



然后在Debugger Immediate Window中输入g，系统运行起来，鼠标可以点击了。

最后一步，将刚刚编译的SYS文件拷入虚拟机，用驱动加载工具安装驱动，启动驱动时，触发VS2015中下的断点，此时就可以像平时调试程序一样单步调试了。

拷贝Debug版本，64位（和OS对应）的驱动程序

注意在WIN7以后的X64的驱动需要签名，由微软授权，可以通过以下简易操作

bcdedit /set loadoptions DISABLE\_INTEGRITY\_CHECKS 加载无签名的驱动

出现的问题：

1. 虚拟机中的目标计算机，必须处于可调试状态
2. DebugView和InstDrv必须以管理员的身份运行；普通用户，内核态的信息无法捕获
3. 虚拟机目标计算机处于挂起状态后，下断点，然后让目标计算机运行，安装驱动文件，启动服务，此时可以单步调试，顺序不能颠倒。
4. 如果虚拟机出现异常，重启解决，由于DebugView会锁定一些资源，导致系统不可运行。

### 字符串传入内核

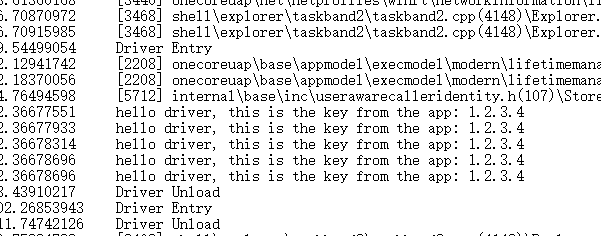
字符串传入内核，应用程序向驱动写信息，驱动截获信息在debugview显示信息，内核层源码如下：



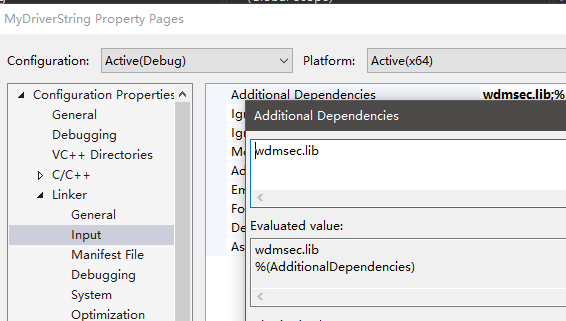
在应用层通信程序，打开设备后，传递IO控制码，携带字符串信息传递到内核中，由内核驱动处理。源码如下：



运行结果如下图：



在VS中编译需要连接库wdmsec.lib



### 加解密模块

加解密主要采用RC4算法进行加解密，对数据流操作后，长度大小不变，这对内核文件信息处理较为方便

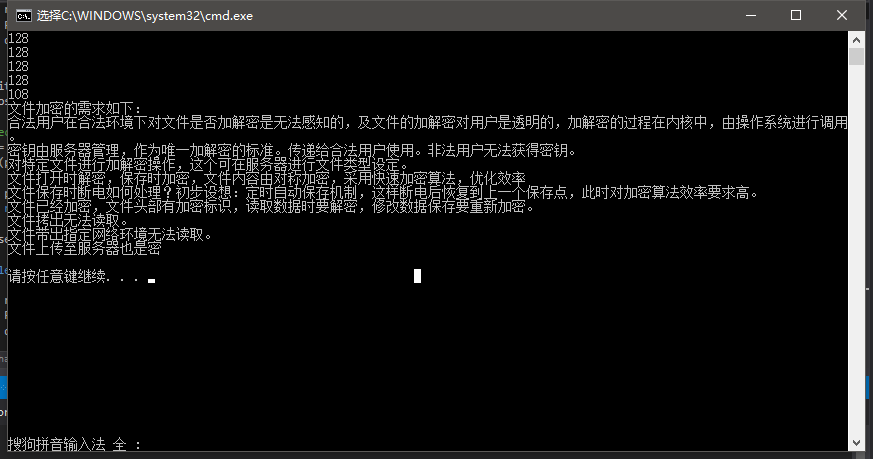
测试中英文结果如下：



测试程序如下：



控制台输出界面如下：



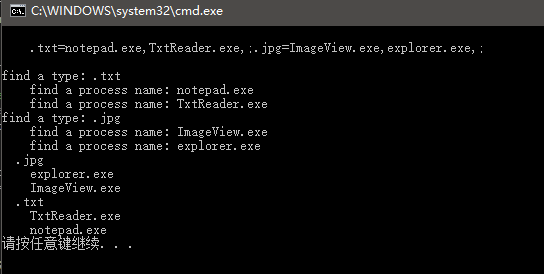
关于ppt讲述稿必须着手制作了

### 策略表解析模块

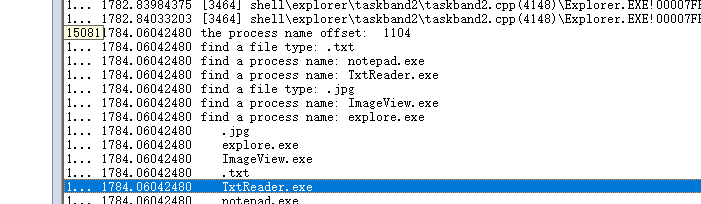
对策略表解析模块进行测试，测试代码如下：



测试结果如下:



在内核中结果如下：



### 获取当前访问文件名和进程名

获取当前访问特定文件的进程名，以及正在处理的文件名称，源码如下：



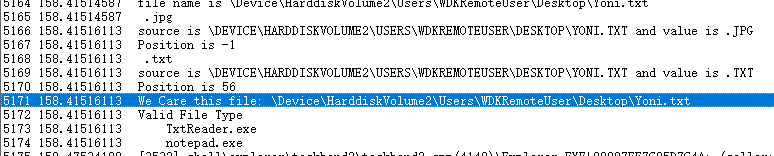
在虚拟机中用DebugView获得的结果如下：



可以看到在create 延后操作中notepad.exe在处理Eula.txt文件

### 验证查询策略表

新建一个.txt文件，在POSTCREAT操作中捕获到文件名，对比策略表，然后显示与其对应的机密进程如下图所示：

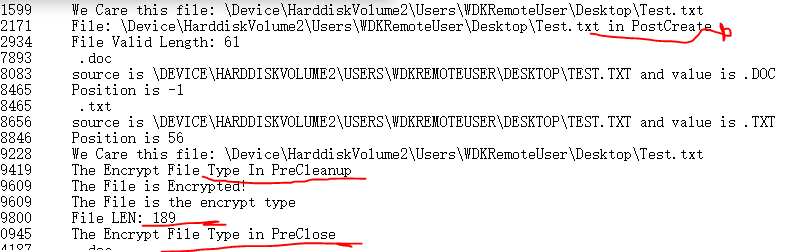


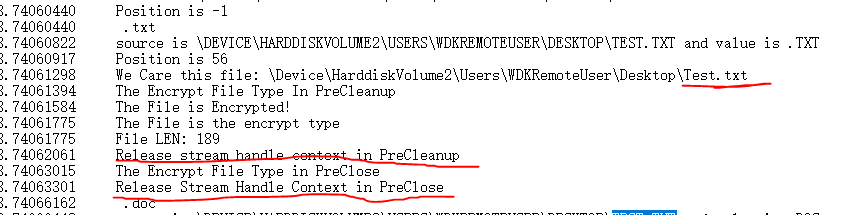
源码如下：



### 为驱动添加上下文

这个实在POSTCREATE中获取文件名和进程名，同时比较策略表，只有通过这些检查的，才去设置上下文



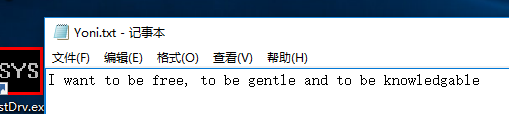


流句柄上下文传递状态，并且适时释放。

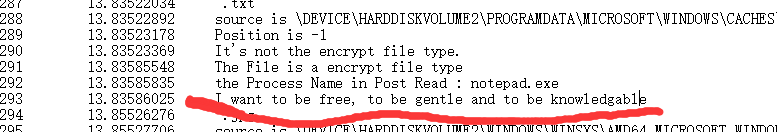
### 读写内容显示

对文件的读，在内核中取出文件的内容，打印出来。

文件Yoni.txt内容如下：



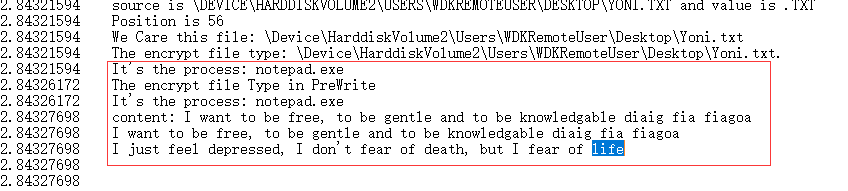
内核截取的内容如下：



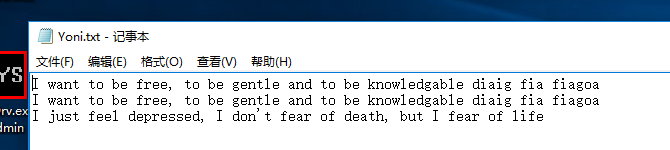
关于读的源码如下：



我在记事本中写入字符，然后CTRL+S保存，在内核中截取的数据如下图：



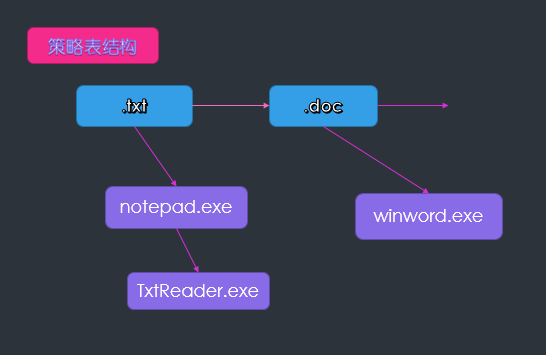
Yoni.txt文件内容如下：



关于写的源码如下：

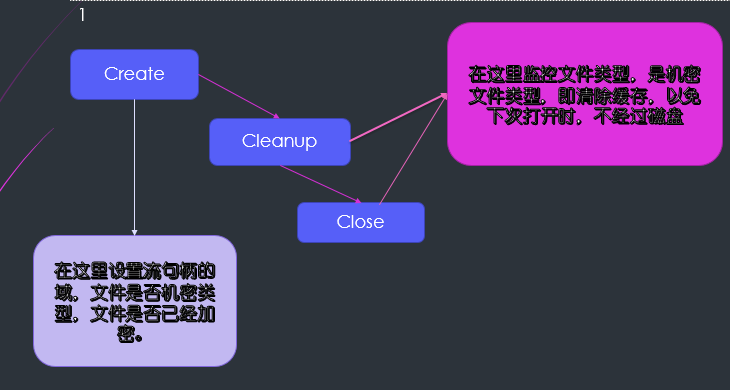


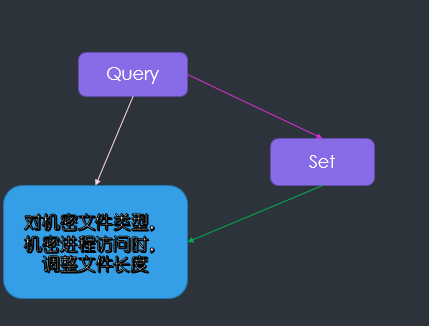
### 关于开发的一些细节



CREATE, CLEANUP, CLOSE, QUERY, SET, READ,WRITE.

IO请求的开发顺序依照以下步骤：





最后是读写，对数据进行加密

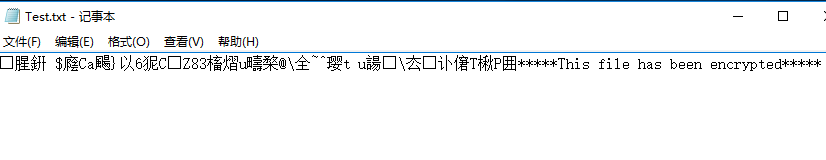
驱动开发经常遇到的错误如下图



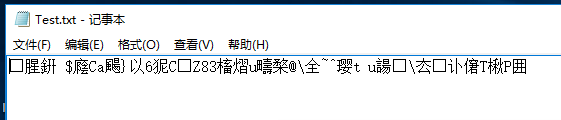
### Query and set information

当机密进程访问机密文件时，需要隐藏加密标识，

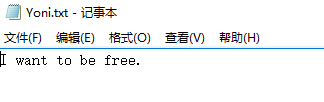
驱动启动前如下图：对于加密文件



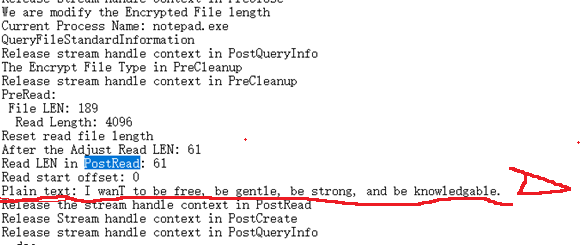
驱动启动后，隐藏尾部加密标识。

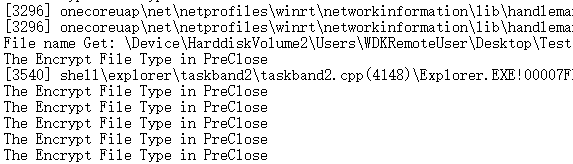


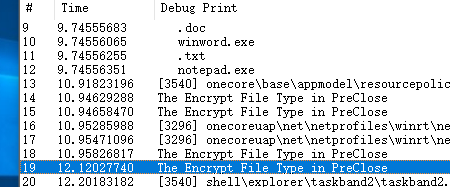
而未加密文件，内容长度均不发生改变，如下图



### Read and write 加解密

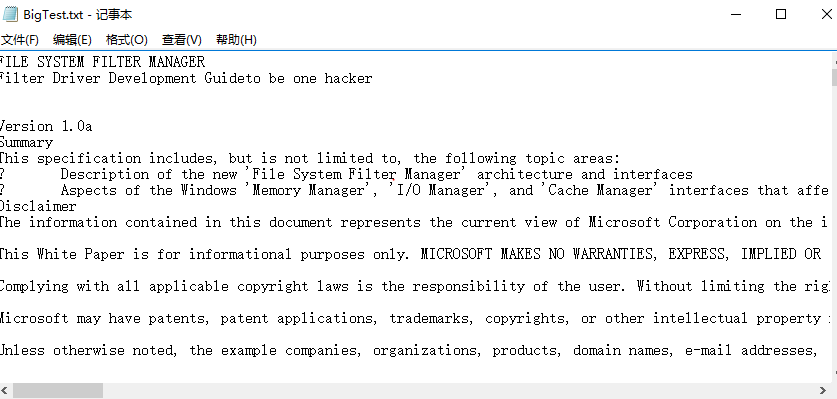


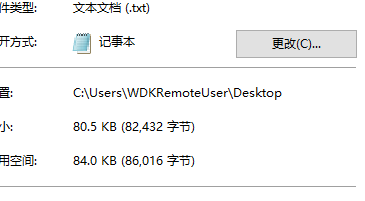




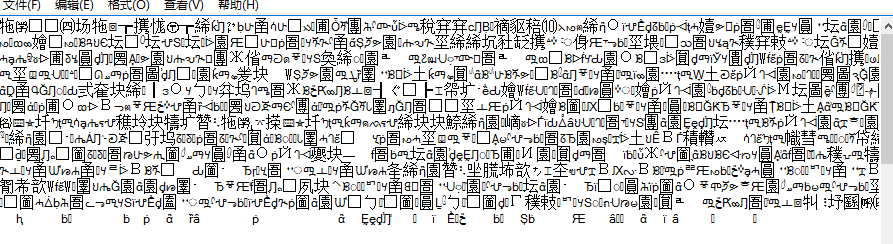
通过对文件的加解密可以看到以下结果：

原文本文件内容如下：





实际存储在磁盘中的内容如下：

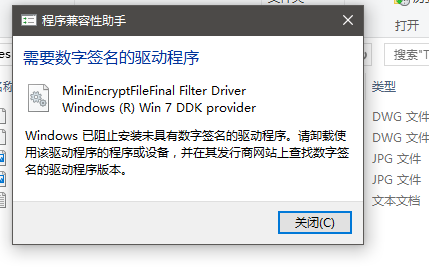


是密文。

### 12 相关问题

1. 内核中文件标识写入文件失败，
2. 文件标识设计，
3. 内核操作同步，多线程访问时变量控制
4. 当前主要在处理，文件标识的问题，
5. FltWriteFile
6. **文件引用次数的问题，close IO请求写入文件加密标识的时机**

### 数字签名



### Minifilter通信

利用Minifilter提供的API，构建通信程序

