

FOKS-TROT 开发文档

引言：

Foxtrot 是一个实验性项目，且作者对于文件系统等理解难免会存在偏差，因此可能会产生误导，望读者辩证的学习，并且请读者遵循相关的开源协议。且本项目 5 月 12 号以前的版本已作为毕设，如有雷同，纯属雷同`((/- -)/)`

经过五个月的维护，**Foxtrot** 迎来了第一个稳定版 1.0.0.2265 版，项目现在是可以稳定运行在 Windows 10 x64 各版本上的（应该可以），建议大家重新 clone 一下，不过本版本不支持之前驱动加密过的文件（见第 11 条）。

P.S. 本来不想更新了，但又想想，还是决定实现一个能稳定运行的版本，要不有 bug 显得我有点菜|·'-'·)ゞ

简介：

Foxtrot 是一个使用 minifilter 框架的双缓冲透明加密解密过滤驱动，当进程有写入特定的文件扩展名（比如 txt, docx）文件的倾向时自动加密文件。授权进程想要读取密文文件时自动解密，非授权进程不解密，显示密文，且不允许修改密文。

桌面端可以发送特权加密和特权解密命令，实现单独加密或解密文件；或者配置进程权限，机密文件夹，需加密的文件类型。

1. 本项目使用双缓冲，授权进程和非授权进程分别使用明文缓冲和密文缓冲；

2. 使用 StreamContext 存放驱动运行时的文件信息，在文件的尾部使用 4KB 文件标识尾储存解密所需信息；

3. 使用 AES-128 ECB 模式，16 个字节以内分别在 SetInfo->EOF 和 WriteCachedIo 时扩展文件大小到 16 字节，大于 16 个字节，使用密文挪用（Ciphertext stealing）的方法，避免明文必须分块对齐的问题；

4. Write 和 Read 使用 SwapBuffers 的方式进行透明加密解密；

5. 特权加密和特权解密使用重入（Reentry）的方式，使驱动加密解密文件；

6. 在 FileRenameInformationEx 和 FileRenameInformation 重命名操作时做处理，可以自动加密解密 docx, doc, pptx, ppt, xlsx, xls 等使用 tmp 文件重命名方式读写的文件；

7. 注册进程相关回调，使用链表统一管理授权与非授权进程；注册进程与线程对象回调，保护进程 EPROCESS, ETHREAD 对象；对授权进程的代码段进行完整性校验。

8. 设置机密文件夹，文件处于该文件夹下才会透明加密，并可以从桌面 PocUser 配置机密文件夹与需管控的文件扩展名 @wangzhankun

9. PostOperation 统一使用函数 FltDoCompletionProcessingWhenSafe（PostRead 除外），InstanceSetup 时使用 Dpc+WorkItem 回调（封装为 PocDoCompletionProcessingWhenSafe），避免在 DISPATCH_LEVEL 时出现 IRQL_NOT_LESS_OR_EQUAL 之类的蓝屏；

10. PostClose 时使用单独的线程，等待操作该文件的所有授权进程结束以后，再重入加密或写入文件标识尾，解决了 docx 类文件的死锁问题。

11. 将 ULONG 改成 LONGLONG，原则上可以支持 4GB 以上文件（目前特权加密和特权解密暂不支持 4GB 以上文件，而且在内存有限的情况下，特权解密有可能会因非分页内存的缺少而失败，不想写了，这里可以放循环里读写大文件）

目录

1. 双缓冲部分的设计:	4
1) 密文缓冲的初始化:	4
2) 密文缓冲和明文缓冲的同步问题:	6
2. StreamContext 和文件标识尾	7
1) StreamContext:	7
2) 文件标识尾:	8
3) 文件标识尾读取和写入的时机:	8
4) 文件标识尾以何种方式写入:	9
5) 文件标识尾的隐藏:	10
3. 密文挪用与扩展文件大小	10
1) 密文挪用的原理:	11
2) 特殊情况解决方法:	11
3) 特殊情况的测试方法:	12
4) 文件小于 16 字节时, 扩展文件大小到 16 字节:	12
4. SwapBufferrs	14
1) Write:	14
2) Read:	15
5. 重入 (Reentry) 和特权加密、特权解密	16
1) 重入与非重入	16
2) 特权加密	17
6. FileRenameInformation/Ex:	20
7. 授权进程的控制与保护:	21
(1) 进程的控制	21
(2) 进程的保护	22
8. Dpc+WorkItem 回调:	22

设计部分：

1. 双缓冲部分的设计：

双缓冲即使用原来的缓冲作为明文缓冲，新建的缓冲作为密文缓冲。主要涉及到密文缓冲的创建与销毁，以及密文缓冲与明文缓冲的同步问题。密文缓冲不允许下发非缓冲写请求到文件系统驱动，它的非缓冲读请求不解密，因而应用程序会显示密文。

1) 密文缓冲的初始化：

FileObject.c->PocInitShadowSectionObjectPointers

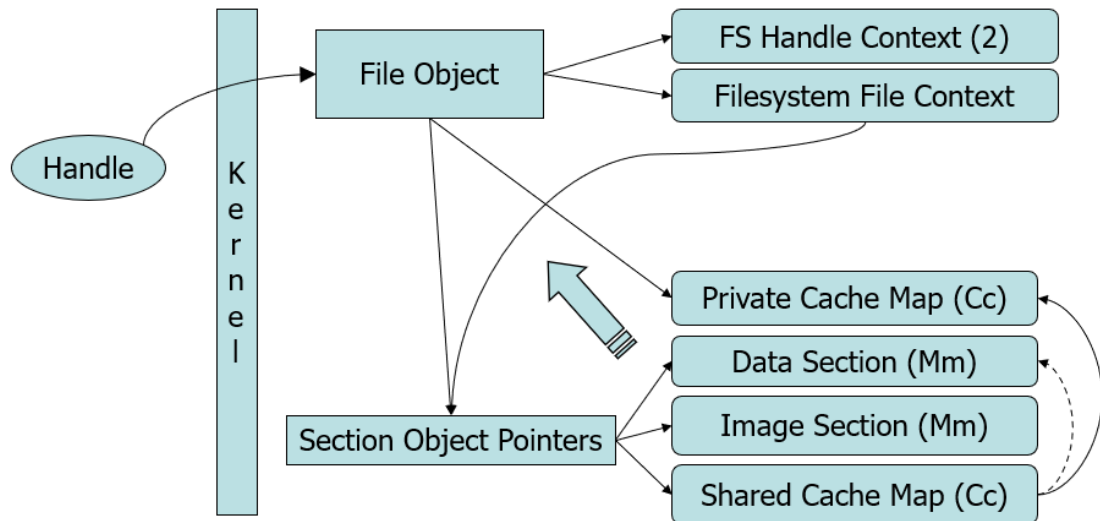
```
1.     PFILE_OBJECT FileObject = FltObjects->FileObject; //非授权进程的 FO
2.
3.     // ShadowSectionObjectPointers 是在 StreamContext 分配的 NonPagedPool 内存
4.     if (NULL == StreamContext->ShadowSectionObjectPointers)
5.     {
6.         DbgPrint("PocInitShadowSectionObjectPointers->ShadowSectionObjectPointers is NULL\n");
7.         Status = STATUS_INVALID_PARAMETER;
8.         goto EXIT;
9.     }
10.
11.     ExEnterCriticalRegionAndAcquireResourceExclusive(StreamContext->Resource);
12.     //在 StreamContext 中记录原始的 FCB 中 SOP 指针位置
13.     StreamContext->OriginSectionObjectPointers = FileObject->SectionObjectPointers;
14.
15.     ExReleaseResourceAndLeaveCriticalRegion(StreamContext->Resource);
16.     //把 FO 的 SOP 指向我们 StreamContext 的 Shadow SOP
17.     FileObject->SectionObjectPointer = StreamContext->ShadowSectionObjectPointers;
18.     //这里直接从上层调接口初始化该 FO 的 Shadow SOP，而不需要自己调用 Cc 函数
19.     Status = FltReadFileEx(FltObjects->Instance, FileObject, &ByteOffset,
20.         sizeof(Buffer), &Buffer, 0, NULL, NULL, NULL, NULL, NULL);
21.
22.     if (!NT_SUCCESS(Status) && STATUS_END_OF_FILE != Status)
23.     {
24.         DbgPrint("PocInitShadowSectionObjectPointers->FltReadFileEx init ciphertext cache failed. Status = 0x%x\n", Status);
25.         goto EXIT;
26.     }
27.     //判断一下缓冲建立好了没有
28.     if (!CcIsFileCached(FileObject))
29.     {
```

```

30.         DbgPrint("PocInitShadowSectionObjectPointers->after FltReadFileEx file doesn't have cache.\n");
31.         Status = STATUS_UNSUCCESSFUL;
32.         goto EXIT;
33.     }

```

如果读者看过 FastFat 的源码，就会发现，缓冲并不是在 Create 中建立的，而是在 Read 或者 Write 有 CachedIo 的需求下建立的，



F0->PrivateCacheMap 和 F0->SOP 是 F0 中和 Cache 相关的部分。

```

1. FileObject->SectionObjectPointer = &Fcb->NonPaged->SectionObjectPointers;

```

F0->SOP 是指向 Fcb 中一块旁视链表分配的 NonPagedPool 内存的指针，这块内存是所有 F0 共享的，也意味着同一个文件的所有 F0 都使用一个缓冲。SOP 的 DataSectionObject 和 SharedCacheMap 是文件缓冲使用的两个指针，DataSectionObject 即 ControlArea，是 Memory Manager 管理缓冲的部分，SharedCacheMap 是 Cache Manager 管理缓冲的部分，ImageSectionObject 是进程用的，这里和文件没有关系。

F0->PrivateCacheMap 是每个 F0 私有的，存放着该 F0 读操作的历史记录，给预读（Read Ahead）算法提供预读位置和预读长度，它不存放实际的缓冲。基本上，我们不需要考虑它，因为它本身指向的内存或者是 F0->SOP->SharedCacheMap 中一块提前预备的内存，或者是 Cache Manager 新分配的内存，然后插入到对应 SharedCacheMap 的 PrivateList 链表中，就是说 PrivateCacheMap 和 SharedCacheMap 有着多对一的对应关系。因为我们是在 PostCreate 中开始处理 F0，所以此时 PrivateCacheMap 并没有建立，只要我们在缓冲还没有建好就替换 SOP，那么 PrivateCacheMap 就是与我们之后建立的 Shadow SOP 相对应。

而我们把 F0->SOP 指向了 StreamContext 中分配的 NonPagedPool，然后调用 FltReadFileEx，使用 CachedIo 向 FSD（File System Driver）下发读请求，FSD 会帮我们调用 CcInitializeCacheMap 建立缓冲，然后它会调用 CcCopyReadEx 尝试从缓冲中读数据。（这里提一个验证缓冲是否建立成功的测试方法：可以在 PreRead 中对使用我们 Shadow SOP 的 F0 进行阻塞

(FLT_PREOP_COMPLETE)，并返回 STATUS_SUCCESS，此时应用程序里就会显示缓冲中原有的脏数据而不是文本内容)

如此一来，密文缓冲就建立好了，存放在 StreamContext->SOP 中，当非授权进程的其他 FO 也想要读写密文缓冲时，我们就把该 FO->SOP 指向已经建立好的密文缓冲。通过密文缓冲读写的 FO，Cache Manager 都会使用 SharedCacheMap->FileObject 进行读写，即缓冲建立时的原始 FO，并增加该 FO 的引用数，保证该 FO 不会被提前 Close。密文缓冲的销毁是在 Cleanup 中调用 CcUnitializeCacheMap 实现的，当 SharedCacheMap->OpenCount==0 等条件满足时就会被删除。

2) 密文缓冲和明文缓冲的同步问题:

FileFuncs, c->PocFlushOriginalCache

```
3)    RtlInitUnicodeString(&uFileName, FileName);
4)
5)    InitializeObjectAttributes(&ObjectAttributes, &uFileName, OBJ_KERNEL_HANDLE
    | OBJ_CASE_INSENSITIVE, NULL, NULL);
6)    //创建一个带有原始缓冲(明文缓冲)的 FO
7)    Status = FltCreateFileEx(
8)        gFilterHandle,
9)        FltObjects->Instance,
10)       &hFile,
11)       &FileObject,
12)       0, //DesiredAccess 为 0, 避免 SharedAccess 冲突
13)       &ObjectAttributes,
14)       &IoStatusBlock,
15)       NULL,
16)       FILE_ATTRIBUTE_NORMAL,
17)       FILE_SHARE_READ | FILE_SHARE_WRITE,
18)       FILE_OPEN,
19)       FILE_NON_DIRECTORY_FILE,
20)       NULL,
21)       0,
22)       IO_IGNORE_SHARE_ACCESS_CHECK); //告诉 FSD 不检查 SharedAccess
23)
24)    if (STATUS_SUCCESS != Status)
25)    {
26)        DbgPrint("PocFlushOriginalCache->FltCreateFileEx failed. Status = 0x%x\n", Status);
27)        goto EXIT;
28)    }
29)
30)    if (CcIsFileCached(FileObject))
31)    { //从上层调用刷新缓存的接口, 刷新明文缓存
```

```

32)         Status = FltFlushBuffers(FltObjects->Instance, FileObject);
33)
34)         if (STATUS_SUCCESS != Status)
35)         {
36)             DbgPrint("PocFlushOriginalCache->FltFlushBuffers failed. Status = 0
xxx\n", Status);
37)             goto EXIT;
38)         }
39)     }

```

当非授权进程打开的 FO 有读的倾向时，我们使明文缓存（如果有的话）尽快下刷到磁盘中，这样等到密文缓冲建立时，从磁盘中读到的数据是最新的。刷新缓存尽量要从上层去调接口，让 FSD 去判断条件，取相关的锁，然后刷新，尽可能地避免自己调用 CcFlushCache。

2. StreamContext 和文件标识尾

StreamContext 是 minifilter 驱动运行时为每个文件维护的一块内存，用于存放运行时所需的文件信息，比如文件大小，是否已加密等。文件标识尾是写入文件结尾的一块数据，用于在 minifilter 驱动卸载时静态存储文件的信息。

1) StreamContext:

StreamContext 是我们的 minifilter 驱动为每个文件维护的一块数据，这方面它类似于 Fcb 一样，不过 Fcb 是 FSD 维护的，我们可以在其中定义需要记录的数据，minifilter 卸载时它就被释放了。当然也可以为文件维护一张链表，但是，面对大量的文件时 StreamContext 比链表快，至少它省去了每次遍历查找链表的时间。

```

1. typedef struct _POC_STREAM_CONTEXT
2. {
3.
4.     ULONG Flag;
5.     PWCHAR FileName;
6.
7.     /*
8.      * FileSize 中存着明文==密文大小，因为写进去的尾是 NonCachedIo，所以有扇区对齐，不是紧接着密文写的
9.      * FileSize 主要是用于隐藏尾部，在 PostQueryInformation 和 PreRead, PostRead 中使用
10.     * FileSize 会在 PostWrite 中更新，并在 PostClose 中写进尾部，以便驱动启动后第一次打开文件时，从尾部中取出
11.     */
12.     LONGLONG FileSize;
13.
14.     PSECTION_OBJECT_POINTERS OriginSectionObjectPointers;
15.     PSECTION_OBJECT_POINTERS ShadowSectionObjectPointers;

```

```

16.
17.     BOOLEAN IsCipherText;
18.
19.     // 将倒数第二个扇区大小的块存在 StreamContext->PageNextToLastForWrite 中
20.     POC_PAGE_TEMP_BUFFER PageNextToLastForWrite;
21.
22.     // 等待操作的进程结束以后，写入文件标识尾用
23.     PPOC_CREATED_PROCESS_INFO ProcessInfo[POC_MAX_AUTHORIZED_PROCESS_COUNT];
24.     HANDLE ProcessId[POC_MAX_AUTHORIZED_PROCESS_COUNT];
25.     BOOLEAN AppendTailerThreadStart;
26.     PFLT_VOLUME Volume;
27.     PFLT_INSTANCE Instance;
28.
29.     PERESOURCE Resource;
30.
31. } POC_STREAM_CONTEXT, * PPOC_STREAM_CONTEXT;

```

2) 文件标识尾:

之前的 minifilter 项目使用的是加密标识头，下面简单的列举标识头和标识尾的优缺点。

加密标识头优点是，只要写入标识头文件就是密文，而且头部不会被修改、覆盖；缺点在于既要在读和写时加上偏移跳过头，而且还要修改 FileSize，ValidDataLength 等去隐藏头，对于 FSD 来说，各方面的影响比较多一点。

加密标识尾则不需要在读写修改偏移，但是，写操作以后，需要重新添加标识尾，因为原始的标识尾被破坏了，所以，不能根据是否有尾去判断文件是否被加密，所以使用了标识 IsCipherText，而且也需要修改 FileSize 去隐藏标识尾。

```

1. typedef struct _POC_ENCRYPTION_TAILER
2. {
3.     CHAR Flag[32];
4.     WCHAR FileName[POC_MAX_NAME_LENGTH];
5.     LONGLONG FileSize;
6.     BOOLEAN IsCipherText;
7.     CHAR EncryptionAlgorithmType[32]; //记录使用的加密算法
8.     CHAR KeyAndCiphertextHash[32]; //密钥和一块密文的哈希，用来以后判断密钥是否
    解密成功
9.
10. }POC_ENCRYPTION_TAILER, * PPOC_ENCRYPTION_TAILER;

```

加密标识头和标识尾主要是为了适配 FAT32，而在 NTFS 下有着更多的可操作空间，具体留给读者去研究。

3) 文件标识尾读取和写入的时机:

标识尾是在 PostCreate 中读取并判断的，可以看 FileFuncs.c->PocCreateFileForEncTailer，因为是 NonCachedIo，所以读偏移和读的大小需要和扇区对齐，然后从标识尾中读取数据，写入 StreamContext 中。

```
1.  /*
2.  * 驱动加载后，文件如果有缓冲或者被内存映射读写过，清一下缓冲，防止出现密文
3.  * 但对于非授权进程不需要，因为它不使用该缓冲
4.  * IsCipherText == 0，说明是在驱动加载以后，第一次被打开
5.  */
6.  Status = PocIsUnauthorizedProcess(ProcessName);
7.
8.  if (0 == StreamContext->IsCipherText &&
9.      POC_IS_AUTHORIZED_PROCESS == Status)
10. {
11.     if (FltObjects->FileObject->SectionObjectPointer->DataSectionObject != NULL)
12.     { //这里通过自己构造 IRP，实现 FltFlushBuffers2 功能，刷新并清除之前的缓冲
13.         Status = PocNtfsFlushAndPurgeCache(FltObjects->Instance, FltObjects->FileObject);
14.         //这个函数的清除缓冲部分只有 ntfs 有实现，fastfat 不行
15.         if (STATUS_SUCCESS != Status)
16.         {
17.             DbgPrint("PocCreateFileForEncTailer->PocNtfsFlushAndPurgeCache failed. Status = 0x%x\n", Status);
18.         }
19.         else
20.         {
21.             DbgPrint("PocCreateFileForEncTailer->File has been opened. Flush and purge cache.\n");
22.         }
23.     }
24. }
```

在 NonCachedIo 的 Write 加密以后，我们设置 StreamContext->Flag 为 POC_TO_APPEND_ENCRYPTION_TAILER，PostClose 时，如果 Flag 有相应的标识，就写入标识尾（FileFuncs.c->PocAppendEncTailerToFile）。

由 Write 或 SetInformation 或 PostCreate 设置 StreamContext->Flag，由 PostClose 创建线程，等待操作该文件的所有授权进程结束以后，再写入文件标识尾。

4) 文件标识尾以何种方式写入：

最好以 NonCachedIo 写入，在文件 EOF 以扇区大小对齐以后的偏移位置写入。如果以 CachedIo 写入，有重入的风险（详见下文的 5.1）.a）。

5) 文件标识尾的隐藏:

在 PostQueryInformation 把和 EOF 相关的值都修改成原始大小 (notepad.exe)

```
1. case FileEndOfFileInformation:
2. {
3.     PFILE_END_OF_FILE_INFORMATION Info = (PFILE_END_OF_FILE_INFORMATION)InfoBuffer;
4.     Info->EndOfFile.LowPart = StreamContext->FileSize;
5.     break;
6. }
```

然后在 PostRead 中修改读取成功的结果值: (wordpad.exe notepad++.exe)

```
1. if (STATUS_SUCCESS == Data->IoStatus.Status)
2. {
3.     if (StartingVbo + Data->IoStatus.Information > FileSize)
4.     {
5.         Data->IoStatus.Information = FileSize - StartingVbo;
6.     }
7. }
8. else if (!NT_SUCCESS(Data->IoStatus.Status) || (Data->IoStatus.Information == 0))
9. {
10.     Status = FLT_POSTOP_FINISHED_PROCESSING;
11.     goto EXIT;
12. }
```

这里是因为 wordpad.exe notepad++.exe 每次会读固定的大小, 例如 EEF 等, 直到 EOF, 然后根据 Read 返回的结果值分配内存。下面再 PreRead 中设置返回 EOF。

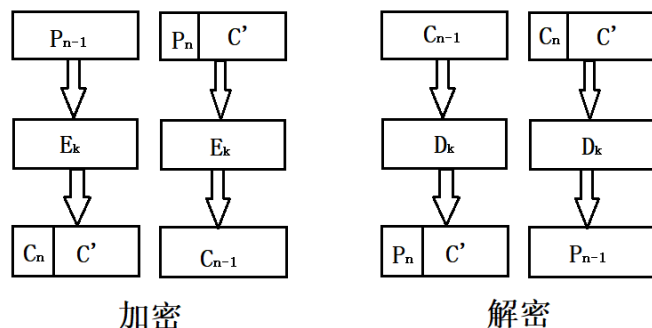
```
1. if (StartingVbo >= StreamContext->FileSize)
2. {
3.     DbgPrint("PocPreReadOperation->%s read end of file.\n", ProcessName);
4.     Data->IoStatus.Status = STATUS_END_OF_FILE;
5.     Data->IoStatus.Information = 0;
6.
7.     Status = FLT_PREOP_COMPLETE;
8.     goto ERROR;
9. }
```

3. 密文挪用与扩展文件大小

对于分块的加密算法, 如何在 PagingIo 下扩展文件大小是做透明加密比较突出的问题。对于文件大小本身就与块大小对齐的文件, 我们直接加密解密。对于不对齐的文件, 这里我们采用密文挪用的方式, 这个方式有两个问题需要解决: 文件大小小于一个块, 不能用这种方式加密; NonCachedIo 以扇区为单位读写数据, 如果最后一次读或写操作的实际数据小于一个块, 首先整个文件是需要做密文挪用加密解密的, 其次需要倒数第二个块, 才能正确加密或解密最后一个块。

密文挪用理论上可以用在其他块加密的算法中，这里使用的是 AES-ECB 模式，AES-CBC 模式同样可以。

1) 密文挪用的原理：



P_{n-1} , P_n 是明文，前者是块对齐的，后者是需要 padding 的
 C_n , C_{n-1} 是密文， C' 是 padding 大小的密文，
 P_{n-1} 经过加密后的密文分为两部分，一部分后移成为 C_n ，另一部分和 P_n 补齐，加密后成为密文 C_{n-1} ， C_{n-1} 前移
最后成为 $C_{n-1}C_n$ 的一块数据，由以上可知，这块数据的长度与原始明文长度一致。
解密流程与加密一致，最终可以解出原始明文。

以上源码在 Cipher.c-> PocAesECBEncrypt_CiphertextStealing 和 PocAesECBDecrypt_CiphertextStealing

2) 特殊情况解决方法：

由于密文挪用的加密解密都是需要倒数第二个块和最后一个块一起处理的，所以，对于整体文件小于一个块的文件，不适合。

NonCachedIo PagingIo Write:

因为 Cache Manager 的 Lazy Writer 刷脏数据时，是用 PagingIo 的，而 PagingIo 是不允许拓展 FileSize 即 EOF 的，EOF 是由应用程序 CachedIo 写操作时扩展的。大致就是先由 CachedIo 修改 EOF，之后某个时间点 Lazy Writer 一块一块的刷下缓冲。

```
1. //LengthReturned 是本次 Write 真正需要写的数据
2. if (!PagingIo || FileSize >= StartingVbo + ByteCount)
3. {
4.     LengthReturned = ByteCount;
5. }
6. else
7. {
8.     LengthReturned = FileSize - StartingVbo;
9. }
```

设想一种特殊情况，如果 Lazy Writer 把一个和块大小不对齐的文件分两次刷下，一次是 0x1000，另一次是 LengthReturned==0xA（实际写入的大小是 0xA，但 Lazy Writer 还是按照 ByteCount==0x1000 刷的），那么这种情况，最后两个块不在一起，就没办法使用密文挪用了。

0x1000	0x200		0xA	
	0x200			
	0x200			
	0x200			
	0x200			
	0x200			
	0x200			
	0x200			
0xE00	0x200		0x200	0x1200
	0x200		0xA	
	0x200			
	0x200			
	0x200			
	0x200			
	0x200			

这里的解决方法是，0x1000 这一次，只刷 0x1000-0x200（扇区大小）= 0xE00 的大小，然后在 StreamContext-> PageNextToLastForWrite 这块之前分配好的内存中，写入最后一个扇区的明文（之所以要按扇区是因为 NonCachedIo 无论偏移还是大小都要和扇区对齐）；然后在 0xA 这一次，取出 0x200 的明文，把 0x200+0xA=0x20A 大小的明文，密文挪用加密后，修改这次写操作的偏移（减一个扇区）和大小，把这 0x200+0x1000=0x1200（实际上的大小）整体下发到 FSD。

注意在每次的 PostWrite 把实际写入的大小改回它应该写入的大小。

a) NonCachedIo Read:

相比 Write 而言, Read 就简单一点了, 情况是一样的, 只是 Read 是读操作。Read 不需要修改两次下发到 FSD 的偏移和大小, 在 0x1000 这一次, 单独以 NonCachedIo 或者非重入的方式, 读出最后一个块的密文, 把 0x1000+0xA 一起解密, 把解密后的 0x1000 写到 OriginalBuffer 中; 0xA 同理。

以上源码分别在 Write.c 和 Read.c 中

3) 特殊情况的测试方法:

Write 的测试需要先用 Nt 或 FltSetInformationFile 设置 EOF，这一步就是模仿上文提到的 CachedIo 扩展 EOF，然后以**重入**且是 **PagingIo**（PagingIo 不修改 EOF，但单独的 NonCachedIo 会扩展 EOF）的方式分别写入 0x1000，0xA（实际上是 0x200，但真正的数据只有 0xA）。

Read 的测试则分别以 NonCachedIo 且重入的方式读取 0x1000, 0xA。

4) 文件小于 16 字节时, 扩展文件大小到 16 字节:

因为，AES 加密需要填充到和 AES 的块大小对齐，而 16 个字节以内的数据无法使用密文挪用，所以，这里使用传统的方法，将文件的 EOF 扩展到 16 个字节。

分别在 Write-CachedIo 和 SetInformation 时扩展。

Write-CachedIo:

```
1. if (!NonCachedIo)
2. {
3.     if (FileSize < AES_BLOCK_SIZE)
4.     {
5.
6.         ExEnterCriticalRegionAndAcquireResourceExclusive(StreamContext->Resource);
7.
8.         StreamContext->FileSize = Data->Iopb->Parameters.Write.Length;
9.
10.        ExReleaseResourceAndLeaveCriticalRegion(StreamContext->Resource);
11.
12.        Status = PocSetEndOfFileInfo(
13.            FltObjects->Instance,
14.            FltObjects->FileObject,
15.            AES_BLOCK_SIZE);
16.
17.        if (STATUS_SUCCESS != Status)
18.        {
19.            PT_DBG_PRINT(PTDBG_TRACE_ROUTINES, ("%s->PocSetEndOfFileInfo failed.\n",
20.                __FUNCTION__));
21.            Data->IoStatus.Status = STATUS_UNSUCCESSFUL;
22.            Data->IoStatus.Information = 0;
23.            Status = FLT_PREOP_COMPLETE;
24.            goto ERROR;
25.        }
26.    }
```

PostSetInformation:

```
1. case FileEndOfFileInformation:
2. {
3.     PFILE_END_OF_FILE_INFORMATION Info = (PFILE_END_OF_FILE_INFORMATION)InfoBuffer;
4.
5.     if (Info->EndOfFile.QuadPart < AES_BLOCK_SIZE && Info->EndOfFile.QuadPart >
6.         0)
7.     {
8.         ExEnterCriticalRegionAndAcquireResourceExclusive(StreamContext->Resource);
9.
10.        StreamContext->FileSize = Info->EndOfFile.QuadPart;
```

```

10.
11.         ExReleaseResourceAndLeaveCriticalRegion(StreamContext->Resource);
12.
13.         Info->EndOfFile.QuadPart = (Info->EndOfFile.QuadPart / AES_BLOCK_SIZE +
14.             1) * AES_BLOCK_SIZE;
15.
16.         Status = PocGetProcessName(Data, ProcessName);
17.
18.         PT_DBG_PRINT(PTDBG_TRACE_ROUTINES, ("%s->EndOfFile filename = %ws origin
19.             filesize = %I64d new filesize = %I64d process = %ws.\n",
20.             __FUNCTION__,
21.             StreamContext->FileName,
22.             StreamContext->FileSize,
23.             Info->EndOfFile.QuadPart,
24.             ProcessName));
25.
26.         FltSetCallbackDataDirty(Data);
27.     }
28.     break;
29. }

```

但是，这种情况，就不要再 PostWrite 中更新 StreamContext->FileSize 的值了，因为已经在上面两处更新过了。

```

1. ExEnterCriticalRegionAndAcquireResourceExclusive(StreamContext->Resource);
2.
3.     if (BooleanFlagOn(Data->Iopb->IrpFlags, IRP_NOCACHE) &&
4.         StreamContext->FileSize >= AES_BLOCK_SIZE ||
5.         0 == StreamContext->FileSize)
6.     {
7.         StreamContext->FileSize = ((PFSRTL_ADVANCED_FCB_HEADER)FltObjects->FileOb
8.             ject->FsContext)->FileSize.QuadPart;
9.     }
10.    ExReleaseResourceAndLeaveCriticalRegion(StreamContext->Resource);

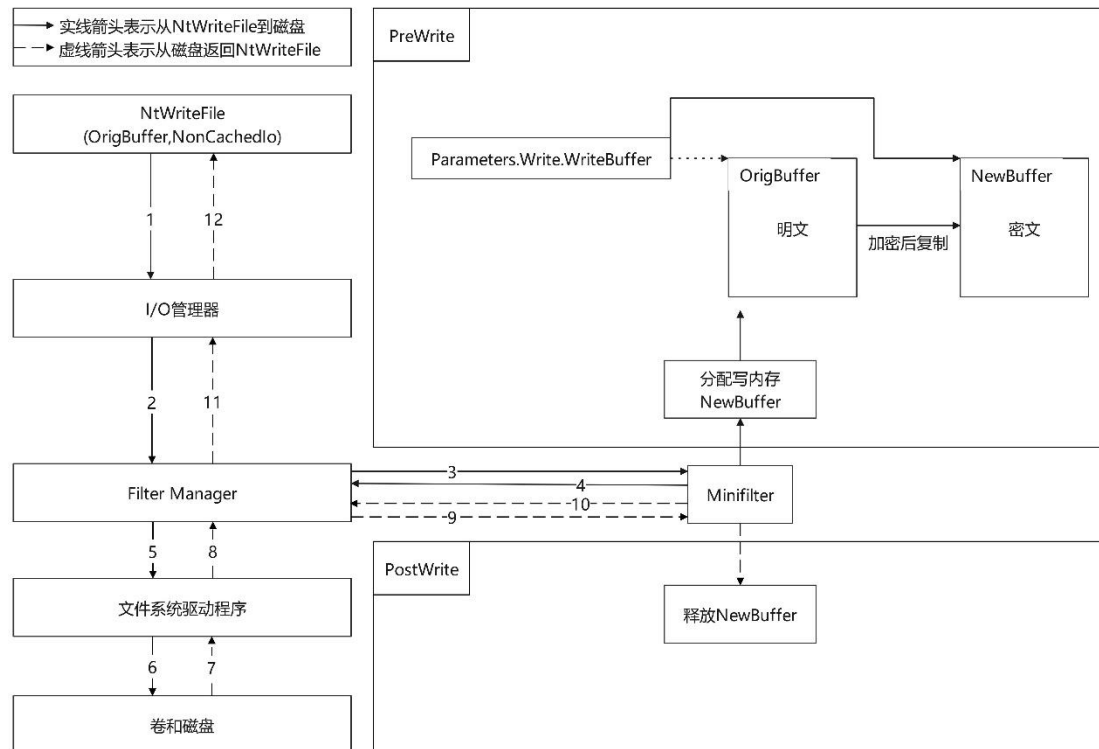
```

4. SwapBuffers

1) Write:

在 Write 从 I/O Manager 把 IRP 发到我们的 minifilter 时，即 PreWrite，我们重新分配一块内存 NewBuffer，把原始内存 OrifBuffer 的数据加密后写入 NewBuffer，把 IRP 的 Buffer 等指针重新指向我们新分配的 NewBuffer，这样发下去的就是密文。然后再 PostWrite 时，把新分配的内存 NewBuffer 释放

掉，并且更新 StreamContext->FileSize。具体并不复杂，读者可以看微软的 swapBuffers sample。



```

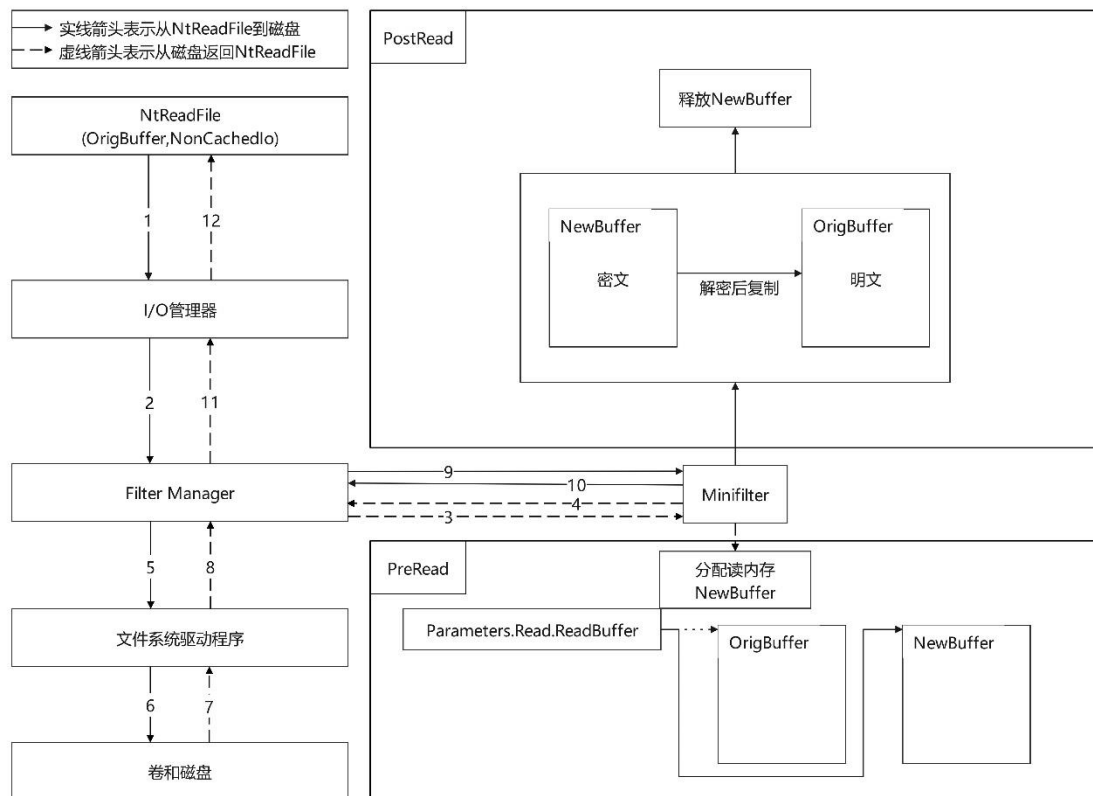
1. if (FltObjects->FileObject->SectionObjectPointer == StreamContext->ShadowSection
    ObjectPointers && NonCachedIo)
2. {
3.     DbgPrint("PocPreWriteOperation->Block StartingVbo = %d ProcessName = %s File
        = %ws.\n",
4.         Data->Iopb->Parameters.Write.ByteOffset.LowPart, ProcessName, StreamCont
        ext->FileName);
5.
6.     Data->IoStatus.Status = STATUS_SUCCESS;
7.     Data->IoStatus.Information = Data->Iopb->Parameters.Write.Length;
8.
9.     Status = FLT_PREOP_COMPLETE;
10.    goto ERROR;
11. }

```

按照之前的设计，密文缓冲是不允许下发的，这里直接结束 IRP，返回成功。

2) Read:

与 Write 稍有不同的是，Read 是在 PostRead 做的 SwapBuffers，因为 PostRead 时，数据才从 FSD 读出，我们也是在 PreRead 分配内存，替换 IRP 指针，然后在 PostRead 中解密密文。



```

1. if (FltObjects->FileObject->SectionObjectPointer
2.    == StreamContext->ShadowSectionObjectPointers)
3. {
4.     SwapBufferContext->StreamContext = StreamContext;
5.     *CompletionContext = SwapBufferContext;
6.     Status = FLT_PREOP_SUCCESS_WITH_CALLBACK;
7.     goto EXIT;
8. }

```

非授权进程的密文缓冲在 PostRead 中不使用 NewBuffer 替换 OrigBuffer。

```

1. if (FltObjects->FileObject->SectionObjectPointer
2.    == StreamContext->ShadowSectionObjectPointers)
3. {
4.     DbgPrint("PocPostReadOperation->Don't decrypt ciphertext cache map.\n");
5.     Status = FLT_POSTOP_FINISHED_PROCESSING;
6.     goto EXIT;
7. }

```

非授权进程的密文缓冲在 PostRead 中不解密。

5. 重入 (Reentry) 和特权加密、特权解密

1) 重入与非重入

关于 minifilter Create, Read, Write 的重入与非重入，读者可以去看下面这篇文章：

Issuing IO in minifilters: Part 2 - Flt vs. Zw

这里简单的解释一下：

a) FltCreateFile 创建的 F0，之后无论是 Flt 还是 Zw 读写操作，都会是非重入的方式。这是因为 I/O Manager 会调用 IoGetRelatedDeviceObject，解析出 F0 相关的设备对象，然后把 IRP 发给 Filter Manager，由 Filter Manager 决定下发的卷实例。

b) 使用 ZwCreateFile 的 F0，必须使用 FltRead/WriteFile 才能非重入。

c) 当然也可以使用 FltPerformSynchronousIo 直接下发 IRP，不经过 I/O Manager 来非重入。

d) 除了文章中的情况以外，如果在写之前已经有别的进程创建了缓冲，那么该缓冲下发时使用的 F0 是该进程创建的 F0，而不是我们写操作的非重入 F0，这样的写操作是重入的。也就是说，即便用了 FltWriteFile/Ex，如果是 **CachedIo**，还是有可能是重入的，或者是其他最终调用 MmFlushCache 的函数，比如 FltSetInformationFile->EOF，也会导致重入的写操作。

因为此时 minifilter 的 Create, Read, Write 相关操作都已建立，所以特权加密，特权解密直接使读写操作重入，minifilter 调用自己来实现加密或解密。

2) 特权加密

FileFuncs.c->PocReentryToEncrypt

```
3) Status = PocReentryToGetStreamContext( //调用 ZwCreateFile 重入到 Create 中
4)     Instance, //判断是否有标识尾
5)     FileName, //并且建立 StreamContext
6)     &StreamContext);
7)
8) .....
9)
10) if (TRUE == StreamContext->IsCipherText) //判断 StreamContext 中的标识
11) { //判断是否已经是密文了
12)     Status = POC_FILE_IS_CIPHertext;
13)     DbgPrint("PocDirectEncrypt->%ws is ciphertext. Encrypt failed.\n", File
        Name);
14)     goto EXIT;
15) }
16)
17)
18) RtlInitUnicodeString(&uFileName, FileName);
19)
20) InitializeObjectAttributes(
```

```

21)         &ObjectAttributes,
22)         &uFileName,
23)         OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE,
24)         NULL,
25)         NULL);
26)
27)     Status = ZwCreateFile (
28)         gFilterHandle,
29)         Instance,
30)         &hFile,
31)         &FileObject,
32)         GENERIC_READ,
33)         &ObjectAttributes,
34)         &IoStatusBlock,
35)         NULL,
36)         FILE_ATTRIBUTE_NORMAL,
37)         FILE_SHARE_READ | FILE_SHARE_WRITE,
38)         FILE_OPEN,
39)         FILE_NON_DIRECTORY_FILE |
40)         FILE_SYNCHRONOUS_IO_NONALERT,
41)         NULL,
42)         0,
43)         0);
44)
45)     if (STATUS_SUCCESS != Status)
46)     {
47)         DbgPrint("PocDirectEncrypt->FltCreateFileEx failed. Status = 0x%x\n", S
tatus);
48)         goto EXIT;
49)     }
50)
51)     FileSize = PocQueryEndOfFileInfo(Instance, FileObject);
52)
53)     .....
54)
55)     Status = PocReadFileFromCache(    //读出原文，此时是明文
56)         Instance,
57)         FileObject,
58)         ByteOffset,
59)         ReadBuffer,
60)         FileSize);
61)
62)     if (NULL != hFile)
63)     {

```

```

64)     FltClose(hFile);
65)     hFile = NULL;
66) }
67)
68) if (NULL != FileObject)
69) {
70)     ObDereferenceObject(FileObject);
71)     FileObject = NULL;
72) }
73)
74)
75)
76)
77)     RtlZeroMemory(&ObjectAttributes, sizeof(ObjectAttributes));
78)
79)     InitializeObjectAttributes(
80)         &ObjectAttributes,
81)         &uFileName,
82)         OBJ_KERNEL_HANDLE | OBJ_CASE_INSENSITIVE,
83)         NULL,
84)         NULL);
85)
86)
87)     Status = ZwCreateFile(           //以重入的方式写入，使 minifilter 加密明文
88)         &hFile,
89)         GENERIC_WRITE,
90)         &ObjectAttributes,
91)         &IoStatusBlock,
92)         NULL,
93)         FILE_ATTRIBUTE_NORMAL,
94)         FILE_SHARE_READ,
95)         FILE_OPEN,
96)         FILE_NON_DIRECTORY_FILE |
97)         FILE_SYNCHRONOUS_IO_NONALERT |
98)         FILE_WRITE_THROUGH,
99)         NULL,
100)        0);
101)
102).....
103)
104)     Status = PocWriteFileIntoCache(
105)         Instance,
106)         FileObject,
107)         ByteOffset,

```

```

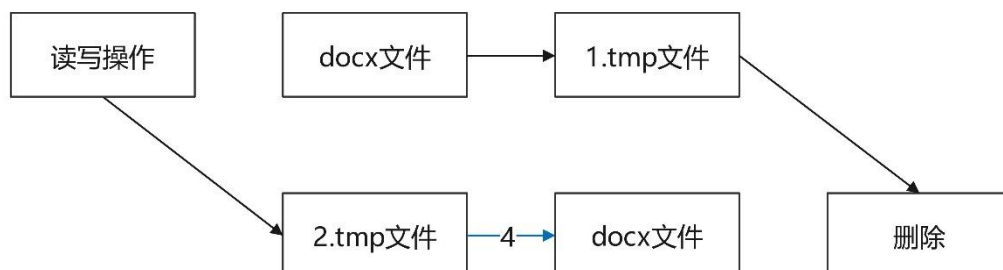
108)     ReadBuffer,
109)     FileSize);
110)
111)     DbgPrint("\nPocDirectEncrypt->success. FileName = %ws FileSize = %d.\n",
112)     FileName,
113)     ((PFSRTL_ADVANCED_FCB_HEADER)(FileObject->FsContext))->FileSize.LowPart);

```

代码并不复杂，特权解密也是一样的，只是特权解密最后的明文需要用 NonCachedIo 非重入写入（原因见 d）。

6. FileRenameInformation/Ex:

有时用户会有重命名文件的需求，有时还会修改扩展名；像 WPS 这种软件，处理 docx，doc 等文件时，是通过两次重命名实现的：将原来的 docx 文件，重命名为 tmp 扩展文件，将另一个打开过程真正写入数据的 tmp 文件重命名为 docx 文件，这样就实现了替换。



而且还会有已加密文件重命名后，加密标识尾中的 FileName 需要更新的问题。

所以，作者为 Rename 制定了几条实现的规则：

- (1) 已加密、未加密目标扩展名文件重命名为非目标扩展名文件：

删除 StreamContext 中的数据。

- (2) 已加密目标扩展名文件重命名为目标扩展名文件：

删除 StreamContext 中的数据，之后下一次的 Create 时，会发现实际文件名与标识尾的文件名不同，设置 StreamContext->Flag，然后 PostClose 会重新写入标识尾。

- (3) 未加密目标扩展名文件重命名为目标扩展名文件：

删除 StreamContext 中的数据。

- (4) 非目标扩展名文件(且不是由加密文件重命名的)重命名为目标扩展名文件：

在 PostSetInformation 中重入 Create 中判断是否有标识尾，没有则设置 StreamContext->Flag，在 PostClose 中用重入的方式加密文件。

- (5) 非目标扩展名文件且本身是密文重命名为目标扩展名文件：

删除 StreamContext 中的数据，之后下一次的 Create 时，会发现实际文件名与标识尾的文件名不同，设置 StreamContext->Flag，然后 PostClose 会重新写入标识尾

五条规则中，删除 StreamContext 中的数据就意味着，除非改成了目标扩展名文件，否则就不会再进入 PostCreate 中了，我们的 minifilter 就不会再处理该文件了。

关于 4) 还有一点需要补充, 在 PostSetInformation 我们捕捉到重命名, 并重入 PostCreate 判断文件并不是密文、建立 StreamContext, 返回 PostSetInformation 设置 StreamContext->Flag 之后, 到 PostClose 真正加密, 这之间有一段时间, 有可能 WPS 会写入数据, 我们的 Write 不会对这部分数据加密, 会直接放过。

之前 4) 这种情况, 是不单独重入加密的, 因为 WPS 会在重命名以后, 再写入数据, 就会自动进入 Write 中进行加密。但是因为 WPS 对 docx, doc, pptx, ppt 文件的处理方式不同, 有一些会在重命名后再写下数据, 有一些则在重命名之前就写入了数据(这种情况就没法加密了), 有一些甚至重命名后会加一段偏移以后, 写下数据(损坏)。所以, 为了统一, 按照 4) 规则处理。

```
1. if (POC_RENAME_TO_ENCRYPT == StreamContext->Flag) //Write 中放过这一段时间的写入
2. {
3.     DbgPrint("PocPreWriteOperation->leave PostClose will encrypt the file. StartingVbo = %d ProcessName = %s File = %ws.\n",
4.         Data->Iopb->Parameters.Write.ByteOffset.LowPart,
5.         ProcessName,
6.         StreamContext->FileName);
7.     Status = FLT_PREOP_SUCCESS_NO_CALLBACK;
8.     goto ERROR;
9. }
```

这里同样由 PostClose 创建线程, 等待操作该文件的所有授权进程结束后, 再重入加密文件, 解决了写入此类文件的死锁问题。

源码在 FileInfo.c 中, 相信经过上述规则的解释, 读者应该能看懂。

7. 授权进程的控制与保护:

对于进程的鉴别与保护也是本文一个比较重要的问题, 如果一个非授权进程可以冒充授权进程而得到明文, 那么就会导致数据的泄露。

(1) 进程的控制

使用两张链表 ProcessRules 和 ProcessInfo 分别保存进程规则和新创建的进程信息, 在每一个 ProcessRules 中指向对应的 ProcessInfo 链表头 PocCreatedProcessListHead, 这里 ProcessInfo 是一条动态的链表, 而 ProcessRules 是在驱动开始之前配置的静态链表。两张链表 ProcessRules 和 ProcessInfo 中定义各成员如下:

```
typedef struct _POC_PROCESS_RULES
{
    LIST_ENTRY ListEntry;
    PWCHAR ProcessName;
    ULONG Access;
```

```

LIST_ENTRY PocCreatedProcessListHead;
KSPIN_LOCK PocCreatedProcessListSpinLock;

}POC_PROCESS_RULES, * PPOC_PROCESS_RULES;

typedef struct _POC_CREATED_PROCESS_INFO
{
    LIST_ENTRY ListEntry;
    HANDLE ProcessId;
    BOOLEAN ThreadStartUp;

    PPOC_PROCESS_RULES OwnedProcessRule;

}POC_CREATED_PROCESS_INFO, * PPOC_CREATED_PROCESS_INFO;

```

使用函数 `PsSetCreateProcessNotifyRoutineEx` 注册进程创建回调，参数 `CreateInfo` 不为空表示新进程的创建，如果该进程在 `ProcessRules` 链表中，则将其加入到 `ProcessInfo` 中，而当 `CreateInfo` 为空表示该进程结束，将其从 `ProcessInfo` 链表摘除。并且调用函数 `NtQuerySystemInformation` 遍历系统内的 `EPROCESS`，将在回调注册之前就已经创建的相关进程加入到链表中。这样一来就可以按照链表中进程的权限（`Access`）项，判断当前进程的权限^[4]，从而决定对于文件是解密还是不解密。

(2) 进程的保护

`Minifilter` 中创建线程，间隔一段时间对授权进程的 `.text`（代码段）进行完整性校验。首先读取进程的文本文件，将进程的各节映射到内存中，对代码按照重定位表进行重定位，然后计算进程文本文件 `.text`（代码段）的 `SHA-256` 哈希值。

然后使用函数 `KeStackAttachProcess` 附加到目标进程的内存空间中，修改进程内存的保护属性为可读可执行，读取 `.text` 代码段计算哈希。将两个哈希值进行比较即可判断进程 `.text` 的完整性。通过这种方式可以一定程度上保护授权进程本身不被修改。

8. Dpc+WorkItem 回调：

在 `PostOperation` 中，在 `DISPATCH_LEVEL` 时可以直接调用 `FltDoCompletionProcessingWhenSafe` 函数，而在其他地方，我这里使用 `DPC+ WorkItem` 回调的方式，是函数能运行在一个比较安全的 `IRQL` 中。（函数在 `Dpc.c` 中）

就是传统的实现方式，不过我这里封装成了

```

NTSTATUS PocDoCompletionProcessingWhenSafe(
    IN PVOID SafePostCallback,
    IN PVOID Context,

```

```

    IN PKEVENT Event)
,
在 WorkItem 的回调线程中，直接用函数指针调用 SafePostCallback。
if (NULL != WorkItem->WorkItemParam)
{
    ((PocSafePostCallback) (WorkItem->WorkItemParam->WorkerRoutine)) (
        DeviceObject,
        WorkItem->WorkItemParam->Context);

    if (NULL != WorkItem->WorkItemParam->Event)
        KeSetEvent (WorkItem->WorkItemParam->Event, IO_NO_INCREMENT, TRUE);
}

```

结束：

项目中肯定还有 bug，而且因为测试的不充分，有可能之前的功能被后添加的功能破坏（尽管我努力的测试过）。欢迎读者在 github 上提 issue，如果能修复，也欢迎提 PR，如果你发现并修复了一个 bug，意味着我们的应用又少了一个缺陷。