Windows驱动_WFP之三Callout驱动的操作

原创

Z18_28_19 于 2013-10-20 21:26:46 发布 16924 收藏 3
分类专栏： Windows驱动_WFP

本文链接：https://blog.csdn.net/Z18_28_19/article/details/12888433

还是没最终下那个决定，也许自己真的没有做大事的魄力，或者根本就做不成大事。做大事者，最忌，犹豫不决，婆婆妈妈，拖泥带水。但是，我有那么多的牵绊，我不得不考虑。其实，我心里已经做了决定，现在只是时间的问题。现在，心里的决定还是没有动摇的，这个我确信不疑的。所以，接下来的时间，我要好好准备。WFP差不多了，要开始到文件系统的minifilter，有时间的话，还要开始网络IO，还有javascript。重视基础。

这篇博客，应该是我最最长的一篇了，因为它差不多花了我一个星期左右的时间。我们来看看WFP中的callout驱动的操作。本来 是想分割成几篇的。但是，不太好分，索性就不分了吧。

Callout Driver Operations

Initializing a Callout Driver

callout驱动在DriverEntry函数中初始化自己，主要的初始化任务包含：

指定卸载函数例程。

基于WDM的Callout驱动。

```
VOID
Unload(
  IN PDRIVER_OBJECT DriverObject
);


NTSTATUS
DriverEntry(
  IN PDRIVER_OBJECT DriverObject,
  IN PUNICODE_STRING RegistryPath
)
{
```

```
    ...

    // Specify the callout driver's Unload function
    DriverObject->DriverUnload = Unload;


    ...
}
```

基于WDF的callout驱动的卸载例程。

```
VOID
Unload(
    IN WDFDRIVER Driver
);


NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
)
{
    NTSTATUS status;
    WDF_DRIVER_CONFIG config;
    WDFDRIVER driver;


      ...


    // Initialize the driver config structure
    WDF_DRIVER_CONFIG_INIT(&config, NULL);


    // Indicate that this is a non-PNP driver
    config.DriverInitFlags = WdfDriverInitNonPnpDriver;


    // Specify the callout driver's Unload function
    config.EvtDriverUnload = Unload;


    // Create a WDFDRIVER object
    status =
    WdfDriverCreate(
        DriverObject,
        RegistryPath,
        NULL
```

```
            NULL,
            &config,
            &driver
        );


            ...


            return status;
}
```

创建设备对象。

基于WDM的Callout驱动：

```
PDEVICE_OBJECT deviceObject;


NTSTATUS
DriverEntry(
  IN PDRIVER_OBJECT DriverObject,
  IN PUNICODE_STRING RegistryPath
)
{
  NTSTATUS status;


   ...


   // Create a device object
  status =
  IoCreateDevice(
        DriverObject,
        0,
        NULL,
        FILE_DEVICE_UNKNOWN,
        FILE_DEVICE_SECURE_OPEN,
        FALSE,
        &deviceObject
    );


       ...


      return status;
```

```
}
```

基于WDF的Callout驱动

因为用过滤引擎注册callout，基于WDF的callout驱动需要得到一个指向WDM设备对象的指针，它和框架的设备对象关联。我们可以使用 WdfDeviceWdmGetDeviceObject 这个函数。

```
WDFDEVICE wdfDevice;


NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
)
{
        WDFDRIVER driver;
        PWDFDEVICE_INIT deviceInit;
        PDEVICE_OBJECT deviceObject;
        NTSTATUS status;


         ...


        // Allocate a device initialization structure
        deviceInit =
        WdfControlDeviceInitAllocate(
                driver;
                &SDDL_DEVOBJ_KERNEL_ONLY
        );


        // Set the device characteristics
         WdfDeviceInitSetCharacteristics(
                deviceInit,
                FILE_DEVICE_SECURE_OPEN,
                FALSE
        );


        // Create a framework device object
        status =
         WdfDeviceCreate(
                &deviceInit,
                WDF_NO_OBJECT_ATTRIBUTES,
                &wdfDevice
        );
```

```
        );


          // Check status
        if (status == STATUS_SUCCESS) {


        // Initialization of the framework device object is complete
        WdfControlFinishInitializing(
                wdfDevice
        );


        // Get the associated WDM device object
              deviceObject = WdfDeviceWdmGetDeviceObject(wdfDevice);
        }


              ...


      return status;
  }
```

使用过滤引擎注册Callouts.

Callout驱动创建设备对象后，它就可以使用过滤引擎注册callout,callout驱动可以在任何时候使用过滤引擎注册callout,甚至过滤引擎不在工作状态。注册callout调用FwpsCalloutRegister0.

```
// Prototypes for the callout's callout functions
VOID NTAPI
ClassifyFn(
 IN const FWPS_INCOMING_VALUES0  *inFixedValues,
 IN const FWPS_INCOMING_METADATA_VALUES0  *inMetaValues,
 IN OUT VOID  *layerData,
 IN const FWPS_FILTER0  *filter,
 IN UINT64  flowContext,
 OUT FWPS_CLASSIFY_OUT0  *classifyOut
);


NTSTATUS NTAPI
NotifyFn(
 IN FWPS_CALLOUT_NOTIFY_TYPE notifyType,
 IN const GUID  *filterKey,
 IN const FWPS_FILTER0  *filter
);
```

```
 VOID NTAPI
 FlowDeleteFn(
  IN UINT16  layerId,
  IN UINT32  calloutId,
  IN UINT64  flowContext
);


 // Callout registration structure
 const FWPS_CALLOUT0 Callout =
{
{ ... }, // GUID key identifying the callout
0,       // Callout-specific flags (none set here)
ClassifyFn,
NotifyFn,
FlowDeleteFn
};


// Variable for the run-time callout identifier
UINT32 CalloutId;


NTSTATUS
DriverEntry(
IN PDRIVER_OBJECT DriverObject,
IN PUNICODE_STRING RegistryPath
)
{
    PDEVICE_OBJECT deviceObject;
    NTSTATUS status;


    ...


    status =
    FwpsCalloutRegister0(
          deviceObject,
          &Callout,
          &CalloutId
    );


     ...
```

```
    return status;
}
```

如果调用FwpsCalloutRegister0函数成功，函数的最后一个参数包含一个实时标识callout标识符的指针。实时标识符是一个为callout键的GUID.

一个callout驱动可以执行多个callout,如果callout驱动执行多个callout，它必须为每一个callout调用函数FwpsCalloutRegister0函数。

Processing Notify Callouts

过滤引擎调用callout的notifyFn callout函数通知callout 驱动和callout相关联的事件发生。

Filter Addition

当过滤器为过滤动作指定的callout已经加入到过滤引擎中，过滤引擎调用callout的notifyFn callout函数，notifyType指定为FWPS_CALLOUT_NOTIFY_ADD_FILTER.

Callout驱动在过滤器为过滤的动作指定Callout加入到过滤引擎以后，为过滤引擎注册callout.在这种情况下，过滤引擎不会调用callout的notifyFn callout函数，通知关于任何过滤器的callout.

过滤引擎只当新的为过滤动作指定的过滤器callout被加入到过滤引擎中的时候，调用notifyFn callout函数通知相关的callout.在这种条件下，callout的notifyFn callout函数可能不会为过滤动作指定的callout已经加入到过滤引擎中的每个过滤器获得调用。

如果，在过滤引起已经启动以后，callout驱动注册了一个callout，这个callout必须接收到每一个为过滤动作增加的Callout的每一个已经加入到过滤引擎中的过滤器的信息。callout驱动必须调用合适的管理函数来枚举过滤引擎中的所有过滤器。callout驱动必须通过过滤器类表进行分类，从而找到为过滤动作指定callout的过滤器。

Filter Deletion

当为了过滤动作指定的callout的过滤器从过滤引擎中被删除，过滤引擎调用callout的notifyFn callout函数，在参数notifyType参数中传递FWPS_CALLOUT_NOTIFY_DELETE_FILTER,参数filterKey传递NULL.过滤引擎会为每一个为了过滤动作指定callout的过滤器从过滤引擎中被删除的时候调用notifyFn callout函数。这些过滤器也包含那些callout驱动注册callout前已经被加入到过滤引擎的过滤器。因此，callout可能收到过滤器删除通知，但是收到一些过滤器的加入通知。

如果callout的notifyFn函数没有在其notifyType参数中认出通知的类别。驱动应该省略通知，并返回STATUS_SUCCESS.

callout驱动可以在过滤器被加入到过滤引擎的时候，指定和过滤器相关联的一个上下文结构。这个上下文结构对过滤引擎不透明。callout的classifyFn callout函数可以使用这个上下文结构保存一些信息状态给callout的函数下次给过滤引擎调用时使用。当过滤器从过滤引擎中被删除，callout驱动执行一些必须的清除上下文的动作。

```
// Context structure to be associated with the filters
   typedef struct FILTER_CONTEXT_ {
    .
```

```
  .  // Driver-specific content
  .
} FILTER_CONTEXT, *PFILTER_CONTEXT;


// Memory pool tag for filter context structures
#define FILTER_CONTEXT_POOL_TAG 'fcpt'


// notifyFn callout function
NTSTATUS NTAPI
NotifyFn(
   IN FWPS_CALLOUT_NOTIFY_TYPE  notifyType,
   IN const GUID  *filterKey,
   IN const FWPS_FILTER0  *filter
)
{
   PFILTER_CONTEXT context;


   ASSERT(filter != NULL);


   // Switch on the type of notification
   switch(notifyType) {


   // A filter is being added to the filter engine
        case FWPS_CALLOUT_NOTIFY_ADD_FILTER:


   // Allocate the filter context structure
   context =
      (PFILTER_CONTEXT)ExAllocatePoolWithTag(
                        NonPagedPool,
                        sizeof(FILTER_CONTEXT),
                        FILTER_CONTEXT_POOL_TAG
      );


   // Check the result of the memory allocation
   if (context == NULL) {


     // Return error
     return STATUS_INSUFFICIENT_RESOURCES;
   }
```

```
    // Initialize the filter context structure
    ...

    // Associate the filter context structure with the filter
     filter->context = (UINT64)context;

     break;

    // A filter is being removed from the filter engine
    case FWPS_CALLOUT_NOTIFY_DELETE_FILTER:

        // Get the filter context structure from the filter
         context = (PFILTER_CONTEXT)filter->context;

        // Check whether the filter has a context
        if (context) {

          // Cleanup the filter context structure
           ...

          // Free the memory for the filter context structure
          ExFreePoolWithTag(
                context,
                FILTER_CONTEXT_POOL_TAG
           );

   }
    break;

        // Unknown notification
        default:

        // Do nothing
         break;
}

      return STATUS_SUCCESS;
```

```
}
```

Processing Classify Callouts

当网络数据将被callout进行处理的时候，过滤引擎就会调相应分类callout的函数。这仅仅只发生在指定过滤动作的callout的过滤条件都满足的时候才发生。如果过滤没有过滤条件，过滤引擎就会一直调用分类的callout函数。过滤引擎会传递不同种类的数据给分类callout的函数。这些数据类型包括固定的数据值，原始网络数据，过滤信息，流上下文。过滤引擎传递的部分数据类型要取决于指定的过滤层和分类函数调用的条件。分类的函数也可以使用这些数据元素的复合类型做过滤决定。

WFP支持异步处理分类的callout函数。

Using a Callout for Deep Inspection

当callout正在执行一些深度检查，它的分类callout函数可以检查任何传递的固定数据，媒体数据，任何包原始数据组成的数据和存储在和过滤相关联的上下文或数据流的数据。

```
// classifyFn callout function
VOID NTAPI
ClassifyFn(
IN const FWPS_INCOMING_VALUES0  *inFixedValues,
IN const FWPS_INCOMING_METADATA_VALUES0  *inMetaValues,
IN OUT VOID  *layerData,
IN const FWPS_FILTER0  *filter,
IN UINT64  flowContext,
OUT FWPS_CLASSIFY_OUT  *classifyOut
)
{
 PNET_BUFFER_LIST rawData;
 ...


 // Test for the FWPS_RIGHT_ACTION_WRITE flag to check the rights
 // for this callout to return an action. If this flag is not set,
 // a callout can still return a BLOCK action in order to VETO a
 // PERMIT action that was returned by a previous filter. In this
 // example the function just exits if the flag is not set.
 if (!(classifyOut->rights & FWPS_RIGHT_ACTION_WRITE))
 {
   // Return without specifying an action
   return;
 }


 // Get the data fields from inFixedValues
```

```
 ...


// Get any metadata fields from inMetaValues
 ...


// Get the pointer to the raw data
rawData = (PNET_BUFFER_LIST)layerData;


// Get any filter context data from filter->context
...


// Get any flow context data from flowContext
...


// Inspect the various data sources to determine
// the action to be taken on the data
 ...


// If the data should be permitted...
if (...) {


  // Set the action to permit the data
  classifyOut->actionType = FWP_ACTION_PERMIT;


  // Check whether the FWPS_RIGHT_ACTION_WRITE flag should be cleared
if (filter->flags & FWPS_FILTER_FLAG_CLEAR_ACTION_RIGHT)
{
  // Clear the FWPS_RIGHT_ACTION_WRITE flag
  classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
}


  return;
}


    ...


// If the data should be blocked...
```

```
if (...) {

// Set the action to block the data
classifyOut->actionType = FWP_ACTION_BLOCK;

// Clear the FWPS_RIGHT_ACTION_WRITE flag
classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;

 return;
}

   ...

// If the decision to permit or block should be passed
// to the next filter in the filter engine...
if (...) {

// Set the action to continue with the next filter
classifyOut->actionType = FWP_ACTION_CONTINUE;

return;
}

 ...
}
```

分类callout函数可以查看filter->action.type类型从而决定classifyOut参数的actionType应该返回什么值。我们来看一下这个FWPS_ACTION0的结构体。

```
typedef struct FWPS_ACTION0_ {
        FWP_ACTION_TYPE type;
        UINT32          calloutId;
} FWPS_ACTION0;
```

如果callout需要在决定对数据进行许可还是禁止前要在分类的callout函数外执行额外的对数据包进行检查，它必须先暂停包数据直到数据处理完毕。

在一些过滤层上，传递给分类的callout函数的layerData的值为NULL.

Using a Callout for Deep Inspection of Stream Data

```
// classifyFn callout function
VOID NTAPI
ClassifyFn(
  IN const FWPS_INCOMING_VALUES0  *inFixedValues,
  IN const FWPS_INCOMING_METADATA_VALUES0  *inMetaValues,
  IN OUT VOID  *layerData,
  IN const FWPS_FILTER0  *filter,
  IN UINT64  flowContext,
  OUT FWPS_CLASSIFY_OUT  *classifyOut
)
{
FWPS_STREAM_CALLOUT_IO_PACKET0 *ioPacket;
FWPS_STREAM_BUFFER0 *dataStream;
UINT32 bytesRequired;
SIZE_T bytesToPermit;
SIZE_T bytesToBlock;
...


// Get a pointer to the stream callout I/O packet
ioPacket = (FWPS_STREAM_CALLOUT_IO_PACKET0 *)layerData;


// Get the data fields from inFixedValues
 ...


// Get any metadata fields from inMetaValues
 ...


// Get the pointer to the data stream
dataStream = ioPacket->dataStream;


// Get any filter context data from filter->context
...


// Get any flow context data from flowContext
...


// Inspect the various data sources to determine
// the action to be taken on the data
...
```

```
// If more stream data is required to make a determination...
if (...) {


// Let the filter engine know how many more bytes are needed
ioPacket->streamAction = FWPS_STREAM_ACTION_NEED_MORE_DATA;
ioPacket->countBytesRequired = bytesRequired;
ioPacket->countBytesEnforced = 0;


// Set the action to continue to the next filter
classifyOut->actionType = FWP_ACTION_CONTINUE;


return;
}
...


 // If some or all of the data should be permitted...
if (...) {


    // No stream-specific action is required
    ioPacket->streamAction = FWPS_STREAM_ACTION_NONE;


    // Let the filter engine know how many of the leading bytes
    // in the stream should be permitted
    ioPacket->countBytesRequired = 0;
    ioPacket->countBytesEnforced = bytesToPermit;


    // Set the action to permit the data
    classifyOut->actionType = FWP_ACTION_PERMIT;


    return;
}


      ...


  // If some or all of the data should be blocked...
if (...) {
```

```
    // No stream-specific action is required
    ioPacket->streamAction = FWPS_STREAM_ACTION_NONE;


    // Let the filter engine know how many of the leading bytes
    // in the stream should be blocked
    ioPacket->countBytesRequired = 0;
    ioPacket->countBytesEnforced = bytesToBlock;


    // Set the action to block the data
    classifyOut->actionType = FWP_ACTION_BLOCK;


    return;
  }


      ...


  // If the decision to permit or block should be passed
  // to the next filter in the filter engine...
  if (...) {


  // No stream-specific action is required
  ioPacket->streamAction = FWPS_STREAM_ACTION_NONE;


  // No bytes are affected by this callout
  ioPacket->countBytesRequired = 0;
  ioPacket->countBytesEnforced = 0;


   return;
  }


    ...

}
```

Inspecting Packet and Stream Data

Packet Inspection Points

传入的数据包的地址，分配给要接收的计算机按如下的顺序遍历WFP层。

IP Packer(Network Layer)

所有的IP的包，包含IP包的段，都在这一层进行检查。但是，当包被IPSec进行保护，内容检查或修改在这一层上都不被许可因为包还没有做身份验证或解密。

Transport Layer

在这一层上，单机或者重新组合包都可以被检查。IPSec保护包已经被身份验证或者解密。

Application Layer Enforcement (ALE) Receive or Accept

在这一层，第一个包显示到达本地计算机端点。举例来说，正在到达的TCP同步段(SYN)，UDP流的第一个UDP消息，都将被显示声明。这些包为了连接需要重新进行身份验证，举例来说，防火墙政策改变，都将在这一层声明，ALE重新身份验证的标记也会被设置。

Datagram Data or Stream

UDP的消息和不是ICMP错误消息都在数据报的数据层进行显示声明。TCP数据流(数据流)在数据流层被检查。
发出的数据包的地址，分配给要接收的计算机按如下的顺序遍历WFP层

ALE Connect

TCP连接请求(在SYN段产生之前)和第一个UDP消息，被发送到远程地址都是在这一层完成。

Datagram Data or Stream

Transport and ICMP Error

TCP连接请求(在SYN段产生之前)和第一个UDP消息，被发送到远程地址都是在这一层完成。

IP Packet

IP包段没有被声明，检查发出的IP段是不可以的。

对于那些不源于，没有注定的，分配到本机计算机的地址的IP包或者段可以在转发层进行数据检查。举例来说，如果一个包注定到一个本地客户端被修改至非本地目的地址，然后注入到接收路径，这些都需要在转发层进行注入。同样的，如果一个源于本地资源地址的包需要被修改到一个非本地资源地址，被注入到发送路径中以后，被分发到转发层。

WFP Layer Requirements and Restrictions

Forwarding Layer

如果IP转发被使能对于起源的包或者目的包分配给计算机的地址和被发送或接收到本地计算机不同接口之间的包。默认情况下IP转发是被关闭的，对于IPV4，我们可以使用netsh interface ipv4 set interface 命令打开，对于IPV6使用netsh interface ipv6 set interface命令打开。

转发层可以转发每一个到达的接收的段或者等到接收到所有的段然后在转发。这个叫做段组合。当段组合被关闭(默认是关闭)，只是一次转发IP包到WFP。当段组合被使能，段到WFP，有两次声明，第一次是段的本身，后面段组合包含一个NET_BUFFER_LIST链。WFP当声明段组合到转发层的callout的时候设置FWP_CONDITION_FLAG_IS_FRAGMENT_GROUP标志。我们可以使用netsh interface {ipv4|ipv6} set global groupforwardedfragments=enabled使能段组合。段组合更重新组装是不一样的，它是重新构造原始的IP包到目的地。

转发层的NET_BUFFER_LIST结构体可以描述全部的IP包，IP包的段，或IP包段的组。当IP包的段穿越转发层的时候，对于callout来说将有两次声明，第一次是段，第二次是段的组。当段的组被声明，FWP_CONDITION_FLAG_IS_FRAGMENT_GROUP标志会传递到分类callout函数的参数中。NetBufferList参数是NET_BUFFER_LIST链的第一个结点。

转发注入包在任何的WFP层上都是不可以的。注入的包对callout驱动来说可能再次被声明，为了避免无限循环，驱动应该在被继续调其classifyFn之前首先调用FwpsQueryPacketInjectionState0.驱动应该许可那些注入状态FWPS_PACKET_INJECTION_STATE被设置 FWPS_PACKET_INJECTED_BY_SELF or FWPS_PACKET_PREVIOUSLY_INJECTED_BY_SELF原封不动的传递。

可以使用如下的命令查看当前的"Group Forwarded Fragments"netsh interface {ipv4|ipv6} show global.

Network Layer

IP包的段，被用来对接收的路径声明，在这一层上被声明三次，第一次作为一个IP包，第一次，作为IP段，第三次作为重新组装IP包的一部分。WFP当对网络层的callouts声明段的时候设FWP_CONDITION_FLAG_IS_FRAGMENT.

当加上额外的条件，FWP_MATCH_FLAGS_NONE_SET 和 FWP_CONDITION_FLAG_IS_FRAGMENT标志结合使用可以避免第二次声明。如果callout必须仅仅只检查包(不是段和重新组装)，它必须分析IP的头避免处理装为IP包声明的段。另外的，其实也可以在传输层来检查包。

Transport Layer and ALE

为了能够和IPSec处理共存，在传输层上面分析包的callout也必须在ALE接收和接受层上也注册callout.这样的callout在传输层上检查或者修改大多数的传输，但是必须也许可包分发到ALE 接收接受层。这样的callout也可以在ALE层上检查或修改包。WFP设置FWPS_METADATA_FIELD_ALE_CLASSIFY_REQUIRED 标志指示传输的metadata数据包也需要ALE的检查。IPSec处理被延后，直到那些包创建了初始化连接，那些已经到达ALE层的包需要进行重新身份验证。

传输层和ALE层callout必须注册他们作为轻量级的子层而不是通用型的子层。IPSec/ALE强制内部的callouts驻留在通用型的子层。

如下的表格显示了可以在ALE层声明的包的类型。通过这个可以明确在一些ALE层，不会一直有和其关联的包。

ALE layer	TCP packets	UDP packets
Bind (resource assignment)	not applicable	not applicable
Connect	no packet	first UDP packet (outgoing)
Receive/Accept	SYN (incoming)	first UDP packet (incoming)
Flow Established	final ACK (incoming & outgoing)	first UDP packet (incoming & outgoing)

Packet Indication Format

网络数据在WFP上声明显示是以NDIS的NET_BUFFER_LIST的形式存在的。NET_BUFFER_LIST结构中的Next成员可以被用来描述网络空间列表的链。一般来说WFP仅仅只是给callout指定一个网络空间列表(netBufferList->Next == NULL),除了如下的情况。

WFP在数据流层可以指定网络空间列表的链。

当分类的IP包段被以组的形式转发到callout,WFP指定网络空间列表链。每一个在链中的网络空间列表代表一个段。 虽然，网络空间列表可以描述整个包，但是对于不同的层次，WFP指定不同的开始于IP头的偏移，对于接收的网络层，网络空间列表开始于IP头后面，如果是传输层，网络空间列表开始于传输层后面。IP和传输层的头一直在网络空间列表的第一个NET_BUFFER结构中描述。

网络空间列表中的偏移被指定给callout是通过FWPS_INCOMING_METADATA_VALUES0结构的ipHeaderSize和transportHeaderSize成员。Callout可以通过NDIS函数NdisRetreatNetBufferDataStart and NdisAdvanceNetBufferDataStart去调整网络空间列表的偏移。但是，callout必须在classifyFn返回之前对所做的偏移调整进行取消。

对于在分类函数中发出数据，NET_BUFFER_LIST包含多个NET_BUFFER结构，每一个描述一个IP包。如果有些在网络空间列表中的包被接受，但是另外没有 callout驱动必须做如下的事情：

1，复制阻塞整个网络空间列表。
2，构造一个新的网络空间列表来描述可以被接受的网络空间的子空间。
3，注入新的网络空间列表到发送路径中。

另外，callout可以取消和网络空间列表相连接的不想要的网络空间的连接。注入新的网络空间列表到发送的路径中。但是这这种情况下，callout驱动在调用FwpsFreeCloneNetBufferList0函数进行复制网络空间列表之前必须取消修改。callout驱动必须保存一些原始的网络空间链接信息做为它自己状态信息的一部分。

那些使用解密IPSec ESP的包的Callout必须使用NET_BUFFER中的DATA_LENGTH成员来代替MDL之中的数据来探测包的长度。

Types of Callouts

Inline Inspection Callout

这种类型的callout通常在其classifyFn函数中返回FWP_ACTION_CONTINUE，而且不会以任何方式修改网络传输。一种收集网络状态的callout就属于这种类型。

对于这种类型，FWPS_ACTION0结构中的Type成员需要设置为FWP_ACTION_CALLOUT_INSPECITON.

Out-of-band Inspection Callout

这种类型的callout也不修改网络传输。它在classifyFn函数的外部完成检查，在内部声明为pending，然后使用packer injection funciton重新将暂停的包送入TCP/IP协议栈中，首先复制指定的数据后暂停立即执行，然后在classifyFn函数中返回FWP_ACTION_BLOCK，并设置FWPS_CLASSIFY_OUT_FLAG_ABSORE位。

Inline Modification Callout

这种类型的callout修改网络传输，首先对指定的数据进行复制，然后修改复制的数据，然后重新注入修改后的数据重新送回到TCP/IP协议栈中。这种类型的callout在classifiFn中返回FWP_ACTION_BLOCK，并设置FWPS_CLASSIFY_OUT_FLAG_ABSORB位。

Out-of-band Modification Callout

这种类型的callout首先通过FwpsReferenceNetBufferList0函数引用指定的包，并将intenToModify设置为TRUE.在classifyFn函数中通过FWPS_CLASSIFY_OUT_ABSORE位设置返回FWP_ACTION_BLOCK.当包已经准备在classifyFn外被修改，callout驱动复制并引用包(克隆以后，立刻，原始包被解引用)。callout然后修改克隆而且检查修改包，将其
送回到TCP/IP协议栈中。

对于这种类型的callout应该设置为FWP_ACTION_CALLOUT_TERMINATING.

对于关于FWPS_CLASSIFY_OUT_FLAG_ABSORE的更多信息，看FWPS_CLASSIFY_OUT0.这个标志位在任何WFP丢弃层不再有效。在classifyFn函数中使用FWPS_CLASSIFY_OUT_FLAG_ABSORE位返回
FWP_ACTION_BLOCK
将导致包被默默的丢弃。以这种方式，不会进入任何WFP 丢弃层，也不会导致任何相关的事件产生。

虽然克隆网络空间列表可能被修改，举例来说，通过增加或者删除网络空间或MDL或两者。callout在调用FwpsFreeCloneNetBufferList0函数之前取消任何修改。

为了和另外的执行包的检查或修改的callout共存，在包因为引用或者克隆丢弃重新注入体系暂停之前，callout必须通过清除掉classifyFn函数FWPS_CLASSIFY_OUT0结构的right成员的FWPS_RIGHT_ACTION_WRITE位。当FWPS_RIGHT_ACTION_WRITE位被设置的时候classifyFn函数被调用(证明这个包可以被暂停后面重新注入或者修改)，callout禁止暂停指定也不能够改变当前的动作类型。它必须等待更高权重的callout注入已经修改过的克隆的包。

FwpsPendOperation0函数被用来暂停那些起源于FWPM_LAYER_ALE_RESOURCE_ASSIGNMENT_XXX, FWPM_LAYER_ALE_AUTH_LISTEN_XXX, or FWPM_LAYER_ALE_AUTH_CONNECT_XXX管理过滤层的包。

FwpsPendClassify0函数被用来暂停起源于如下run-time-filtering层的包。

FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V4
FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V6
FWPS_LAYER_ALE_CONNECT_REDIRECT_V4
FWPS_LAYER_ALE_CONNECT_REDIRECT_V6
FWPS_LAYER_ALE_BIND_REDIRECT_V4
FWPS_LAYER_ALE_BIND_REDIRECT_V6

Packet Injection Functions

callout驱动可以调用如下的WFP函数注入已经暂停或者修改的包数据重新到TCP/IP协议栈中。

Injection function	Applicable layer	Destination
FwpsInjectForwardAsync0	network layer	the forwarding data path
FwpsInjectNetworkReceiveAsync0	network layer	the receive data path
FwpsInjectNetworkSendAsync0	network layer	the send data path
FwpsInjectTransportReceiveAsync0	packet data from the transport, datagram data, ICMP error, or ALE layers	the receive data path
FwpsInjectTransportSendAsync0	packet data from the transport, datagram data, ICMP error, or ALE layers	the send data path
FwpsStreamInjectAsync0	TCP data segments	a data stream

另外，FwpsQueryPacketInjectionState0函数查询包数据的检查历史。

如果callout可以提供所有注入函数所需要的信息Cross-layer注入被使能，注入函数还有一个网络空间列表格式的要求。举例来说，callout可以在转发的路径上对包进行捕捉，修改本地计算的的目的地址，调用FwpsInjectTransportReceiveAsync0重定位包到本地TCP/IP协议栈中。

需要对流的注入，注入接收到的包从栈和WFP层的下层开始重新注入，注入发送的数据包从栈和WFP层的最上层开始注入。举例来说，UDP数据报的注入从接收的数据报层，从栈的网络层，参数层，ALE接收或接收层，回到数据报层。另外UDP数据报对发送的数据将重新注入从ALE,数据包层，传输层再回到网络层。
FwpsInjectTransportReceiveAsync0因为之前已经完成通过IPSec检验，将自动为重新注入的包直接越过IPSec的处理。

已经被WFP callout驱动注入的包将再次被callout重新指定，处理因为修改包导致一些原始的过滤条件丢失而不满足的情况。WFP提供 FwpsQueryPackerInjectionState0函数给callout驱动去查询是否包已经被callout注入。为了避免无线的循环，callout应该许可自己注册过的包。

Callout在修改了IP数据包后，必须调整IP或者传输层的checksum，或者两者，Callout可以设置IPv4包的checksum值为0.callout可以使用如下的逻辑调整相关的checksum的值。

```
NDIS_TCP_IP_CHECKSUM_PACKET_INFO ChecksumInfo;
ChecksumInfo.Value =
  (ULONG) (ULONG_PTR)NET_BUFFER_LIST_INFO(
      NetBufferList,TcpIpChecksumNetBufferListInfo);
```

如果ChecksumInfo.Transmit.NdisPackerTcpChecksum被设置为TRUE，TCP发送的操作将被卸载，如果ChecksumInfo.Transmit.NdisPacketUdpChecksum被设置为TRUE，UDP的发送操作将被卸载。

在Windows Vista with Service Pack 1 (SP1) and Windows Server 2008中，如果inMetaValues->headerIncludeHeaderLength比0大，发送的包是原始发送包含IP头的重新注入的包。在Windows Vista with SP1 and Windows Server 2008中执行包含IP头的原始发送注入操作，你必须通过inMetaValues->headerIncludeHeaderLength得到克隆的数量，拷贝inMetaValues->headerIncludeHeader 到新的扩展空间。然后，使用FwpsInjectTransportSendAsync0为包发送网络空间列表。并设置FWPS_TRANSPORT_SEND_PARAMS0参数为NULL.

Packet Modification Examples

```
HANDLE gInjectionHandle;
```

```
void
NTAPI
InjectionCompletionFn(
    IN void* context,
    IN OUT NET_BUFFER_LIST* netBufferList,
    IN BOOLEAN dispatchLevel
)
{
    FWPS_TRANSPORT_SEND_PARAMS0* tlSendArgs
     = (FWPS_TRANSPORT_SEND_PARAMS0*)context;


//
// TODO: Free tlSendArgs and embedded allocations.
//


//
// TODO: Check netBufferList->Status for injection result
//


     FwpsFreeCloneNetBufferList0(netBufferList, 0);
}


void
NTAPI
WfpTransportSendClassify(
    IN const FWPS_INCOMING_VALUES0* inFixedValues,
    IN const FWPS_INCOMING_METADATA_VALUES0* inMetaValues,
    IN OUT void* layerData,
    IN const FWPS_FILTER0* filter,
    IN UINT64 flowContext,
    OUT FWPS_CLASSIFY_OUT0* classifyOut
)
{
      NTSTATUS status;


      NET_BUFFER_LIST* netBufferList = (NET_BUFFER_LIST*)layerData;
      NET_BUFFER_LIST* clonedNetBufferList = NULL;
      FWPS_PACKET_INJECTION_STATE injectionState;
      FWPS_TRANSPORT_SEND_PARAMS0* tlSendArgs = NULL;
      ADDRESS_FAMILY af = AF_UNSPEC;
```

```
  injectionState = FwpsQueryPacketInjectionState0(
          gInjectionHandle,
          netBufferList,
               NULL);
  if (injectionState == FWPS_PACKET_INJECTED_BY_SELF ||
  injectionState == FWPS_PACKET_PREVIOUSLY_INJECTED_BY_SELF)
  {
        classifyOut->actionType = FWP_ACTION_PERMIT;
          goto Exit;
  }


 if (!(classifyOut->rights & FWPS_RIGHT_ACTION_WRITE))
{
       //
       // Cannot alter the action.
       //
          goto Exit;
}


      //
      // TODO: Allocate and populate tlSendArgs by using information from
      // inFixedValues and inMetaValues.
      // Note: 1) Remote address and controlData (if not NULL) must
      // be deep-copied.
      //       2) IPv4 address must be converted to network order.
      //       3) Handle allocation errors.


        ASSERT(tlSendArgs != NULL);


        status = FwpsAllocateCloneNetBufferList0(
              netBufferList,
              NULL,
              NULL,
              0,
              &clonedNetBufferList);


 if (!NT_SUCCESS(status))
{
    classifyOut->actionType = FWP_ACTION_BLOCK;
    classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;

    goto Exit;
 }
```

```
    }


    //
    // TODO: Perform modification to the cloned net buffer list here.
    //


    //
    // TODO: Set af based on inFixedValues->layerId.
    //
      ASSERT(af == AF_INET || af == AF_INET6);


    //
    // Note: For TCP traffic, FwpsInjectTransportReceiveAsync0 and
    // FwpsInjectTransportSendAsync0 must be queued and run by a DPC.
    //


     status = FwpsInjectTransportSendAsync0(
                    gInjectionHandle,
                    NULL,
                    inMetaValues->transportEndpointHandle,
                    0,
                    tlSendArgs,
                    af,
                    inMetaValues->compartmentId,
                    clonedNetBufferList,
                    InjectionCompletionFn,
                    tlSendArgs);


     if (!NT_SUCCESS(status))
     {
  classifyOut->actionType = FWP_ACTION_BLOCK;
  classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;


  goto Exit;
 }


  classifyOut->actionType = FWP_ACTION_BLOCK;
  classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
  classifyOut->flags |= FWPS_CLASSIFY_OUT_FLAG_ABSORB;


//
```

```
 // Ownership of clonedNetBufferList and tlSendArgs
 // now transferred to InjectionCompletionFn.
 //
   clonedNetBufferList = NULL;
   tlSendArgs = NULL;


Exit:


 if (clonedNetBufferList != NULL)
 {
  FwpsFreeCloneNetBufferList0(clonedNetBufferList, 0);
 }
 if (tlSendArgs != NULL)
 {
      //
      // TODO: Free tlSendArgs and embedded allocations.
      //
 }


   return;
}


 Out-of-band Packet Modification from Incoming Datagram Data Layers Copy

 typedef struct DD_RECV_CLASSIFY_INFO_ {
  NET_BUFFER_LIST* netBufferList;
  UINT32 nblOffset;
  UINT32 ipHeaderSize;
  UINT32 transportHeaderSize;
  ADDRESS_FAMILY af;
  COMPARTMENT_ID compartmentId;
  IF_INDEX interfaceIndex;
  IF_INDEX subInterfaceIndex;
 }DD_RECV_CLASSIFY_INFO;


 HANDLE gInjectionHandle;


 void NTAPI
 InjectionCompletionFn(
  IN void* context,
  IN OUT NET_BUFFER_LIST* netBufferList,
  IN BOOLEAN dispatchLevel
 )
```

```
)
{
 DD_RECV_CLASSIFY_INFO* classifyInfo
 = (DD_RECV_CLASSIFY_INFO*)context;


  //
  // TODO: Remove from queue and free classifyInfo.
  //


  //
  // TODO: Check netBufferList->Status for injection result.
  //


  FwpsFreeCloneNetBufferList0(netBufferList, 0);
}


void
DatagramDataReceiveWorker(
  DD_RECV_CLASSIFY_INFO* classifyInfo
  // ... and other parameters
  )
//
// To prevent WFP from making a deep clone (deep-copying MDLs,
// net buffers, net buffer lists, structures, and data mapped by MDLs,
// DatagramDataReceiveWorker should be run by a DPC targeting the
// processor to which the referenced net buffer list was first
// classified. See KeSetTargetProcessorDpc for DPC targeting.
//
{
  NTSTATUS status;
  NET_BUFFER_LIST* clonedNetBufferList;
  ULONG nblOffset =
    NET_BUFFER_DATA_OFFSET(NET_BUFFER_LIST_FIRST_NB(classifyInfo->netBufferList));


  //
  // The TCP/IP stack could have retreated the net buffer list by the
  // transportHeaderSize amount; detect the condition here to avoid
  // retreating two times.
  //
 if (nblOffset != classifyInfo->nblOffset)
  {
 ASSERT(classifyInfo->nblOffset - nblOffset == classifyInfo->transportHeaderSize);

 classifyInfo->transportHeaderSize = 0;
```

```
  }

  //
  // Adjust the net buffer list offset to start by using the IP header.
  //
NdisRetreatNetBufferDataStart(
    NET_BUFFER_LIST_FIRST_NB(classifyInfo->netBufferList),
classifyInfo->ipHeaderSize + classifyInfo->transportHeaderSize,
    0,
    NULL
    );


status = FwpsAllocateCloneNetBufferList0(
classifyInfo->netBufferList,
            NULL,
            NULL,
            0,
            &clonedNetBufferList);


if (!NT_SUCCESS(status))
  {
    // TODO: Handle error condition.
goto Exit;
  }


  //
  // Undo the adjustment on the original net buffer list.
  //


NdisAdvanceNetBufferDataStart(
    NET_BUFFER_LIST_FIRST_NB(classifyInfo->netBufferList),
classifyInfo->ipHeaderSize + classifyInfo->transportHeaderSize,
    FALSE,
    NULL);


  //
  // Because the clone references the original net buffer list,
  // undo the reference that was claimed during classifyFn.
  //
FwpsDereferenceNetBufferList0(
classifyInfo->netBufferList,
    FALSE);
classifyInfo->netBufferList = NULL;
```

```
  //
  // TODO: Modify the cloned net buffer list here.
  // Note: 1) The next protocol field of the IP header could be
  // AH/ESP, in which case the IP header must be rebuilt (and
  // the AH/ESP header removed).
  //       2) The callout must re-calculate the IP checksum.
  //


status = FwpsInjectTransportReceiveAsync0(
gInjectionHandle,
              NULL,
              NULL,
0,
classifyInfo->af,
classifyInfo->compartmentId,
classifyInfo->interfaceIndex,
classifyInfo->subInterfaceIndex,
clonedNetBufferList,
InjectionCompletionFn,
classifyInfo);

if (!NT_SUCCESS(status))
  {
     // TODO: Handle error condition.
goto Exit;
  }



  //
  // Ownership of clonedNetBufferList and classifyInfo is
  // now transferred to InjectionCompletionFn.
  //
clonedNetBufferList = NULL;
classifyInfo = NULL;



Exit:



if (clonedNetBufferList != NULL)
  {
FwpsFreeCloneNetBufferList0(clonedNetBufferList, 0);
  }
if (classifyInfo->netBufferList != NULL)
  {
FwpsDereferenceNetBufferList0(
```

```
 classifyInfo->netBufferList,
        FALSE);
   }


   // TODO: Free other resources on error.
}


void
NTAPI
WfpDatagramDataReceiveClassify(
   IN const FWPS_INCOMING_VALUES0* inFixedValues,
   IN const FWPS_INCOMING_METADATA_VALUES0* inMetaValues,
   IN OUT void* layerData,
   IN const FWPS_FILTER0* filter,
   IN UINT64 flowContext,
   OUT FWPS_CLASSIFY_OUT0* classifyOut
   )
{
   NTSTATUS status;


   NET_BUFFER_LIST* netBufferList = (NET_BUFFER_LIST*)layerData;
   FWPS_PACKET_INJECTION_STATE injectionState;
   DD_RECV_CLASSIFY_INFO* classifyInfo = NULL;


 injectionState = FwpsQueryPacketInjectionState0(
 gInjectionHandle,
 netBufferList,
                       NULL);
 if (injectionState == FWPS_PACKET_INJECTED_BY_SELF ||
 injectionState == FWPS_PACKET_PREVIOUSLY_INJECTED_BY_SELF)
   {
 classifyOut->actionType = FWP_ACTION_PERMIT;
 goto Exit;
   }


 if (!(classifyOut->rights & FWPS_RIGHT_ACTION_WRITE))
   {
      //
      // Cannot alter the action.
      //
 goto Exit;
   }
```

```
  //
  // TODO: Allocate and populate classifyInfo by using information
  // from inFixedValues and inMetaValues.
  //


 classifyInfo->nblOffset =
     NET_BUFFER_DATA_OFFSET(NET_BUFFER_LIST_FIRST_NB(netBufferList));


 ASSERT(classifyInfo != NULL);
 ASSERT(classifyInfo->netBufferList != NULL);


 FwpsReferenceNetBufferList0(
 classifyInfo->netBufferList,
     TRUE // intendToModify
     );


  //
  // TODO: Queue classifyInfo for out-of-band processing.
  //


 classifyInfo = NULL; // Ownership transferred on success.


 classifyOut->actionType = FWP_ACTION_BLOCK;
 classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
 classifyOut->flags |= FWPS_CLASSIFY_OUT_FLAG_ABSORB;


Exit:


 if (classifyInfo)
  {
     // TODO: Free object here.
  }


 return;
}
```

Non-intrusive Out-of-band Inspection from Incoming Transport Layer and ALE Receive/Accept Layers

The following is example code for an inspection procedure that views packet data without changing it.

Copy

```
typedef struct TL_ALE_RECV_CLASSIFY_INFO_ {
   BOOLEAN aleInfo;  // TRUE if information is gathered from Ale receive/accept layer
                     // FALSE if information is gathered from incoming transport layer


   NET_BUFFER_LIST* netBufferList;
   ADDRESS_FAMILY af;
   COMPARTMENT_ID compartmentId;
   IF_INDEX interfaceIndex;
   IF_INDEX subInterfaceIndex;


   HANDLE aleCompletionCtx;


}TL_ALE_RECV_CLASSIFY_INFO;


HANDLE gInjectionHandle;


void
NTAPI
InjectionCompletionFn(
   IN void* context,
   IN OUT NET_BUFFER_LIST* netBufferList,
   IN BOOLEAN dispatchLevel
   )
{
   TL_ALE_RECV_CLASSIFY_INFO* classifyInfo = (TL_ALE_RECV_CLASSIFY_INFO*)context;


   //
   // TODO: Remove from queue and free classifyInfo.
   //


   //
   // TODO: Check netBufferList->Status for injection result.
   //


 FwpsFreeCloneNetBufferList0(netBufferList, 0);
}
```

```
void
TlAleReceiveWorker(
   TL_ALE_RECV_CLASSIFY_INFO* classifyInfo
   // ... and other parameters
   )
{
   NTSTATUS status;


 if (classifyInfo->aleInfo)
   {
 FwpsCompleteOperation0(
 classifyInfo->aleCompletionCtx,
 classifyInfo->netBufferList);
   }


 status = FwpsInjectTransportReceiveAsync0(
 gInjectionHandle,
              NULL,
              NULL,
              0,
 classifyInfo->af,
 classifyInfo->compartmentId,
 classifyInfo->interfaceIndex,
 classifyInfo->subInterfaceIndex,
 classifyInfo->netBufferList,
 InjectionCompletionFn,
 classifyInfo);

 if (!NT_SUCCESS(status))
   {
      // TODO: Handle error condition.
 goto Exit;
   }


   //
   // Ownership of classifyInfo now transferred to InjectionCompletionFn.
   //
 classifyInfo = NULL;


Exit:


 if (classifyInfo != NULL)
```

```
  {
FwpsFreeCloneNetBufferList0(classifyInfo->netBufferList, 0);


    // TODO: Remove from queue and free classifyInfo.
  }


  // TODO: Free other resources on error.
}


void
NTAPI
WfpAleReceiveClassify(
  IN const FWPS_INCOMING_VALUES0* inFixedValues,
  IN const FWPS_INCOMING_METADATA_VALUES0* inMetaValues,
  IN OUT void* layerData,
  IN const FWPS_FILTER0* filter,
  IN UINT64 flowContext,
  OUT FWPS_CLASSIFY_OUT0* classifyOut
  )
{
  NTSTATUS status;


  NET_BUFFER_LIST* netBufferList = (NET_BUFFER_LIST*)layerData;
  NET_BUFFER_LIST* clonedNetBufferList = NULL;
  FWPS_PACKET_INJECTION_STATE injectionState;
TL_ALE_RECV_CLASSIFY_INFO* classifyInfo = NULL;


injectionState = FwpsQueryPacketInjectionState0(
gInjectionHandle,
netBufferList,
                    NULL);
if (injectionState == FWPS_PACKET_INJECTED_BY_SELF ||
injectionState == FWPS_PACKET_PREVIOUSLY_INJECTED_BY_SELF)
  {
classifyOut->actionType = FWP_ACTION_PERMIT;
goto Exit;
  }


if (!(classifyOut->rights & FWPS_RIGHT_ACTION_WRITE))
  {
    //
    // Cannot alter the action.
```

```
    //
goto Exit;
  }
  //
  // Adjust the net buffer list offset so that it starts with the IP header.
  //
NdisRetreatNetBufferDataStart(
    NET_BUFFER_LIST_FIRST_NB(netBufferList),
inMetaValues->ipHeaderSize + inMetaValues->transportHeaderSize,
    0,
    NULL
    );


status = FwpsAllocateCloneNetBufferList0(
netBufferList,
            NULL,
            NULL,
            0,
            &clonedNetBufferList);


if (!NT_SUCCESS(status))
  {
classifyOut->actionType = FWP_ACTION_BLOCK;
classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;


goto Exit;
  }


  //
  // Undo the adjustment on the original net buffer list.
  //


NdisAdvanceNetBufferDataStart(
    NET_BUFFER_LIST_FIRST_NB(netBufferList),
inMetaValues->ipHeaderSize + inMetaValues->transportHeaderSize,
    FALSE,
    NULL);


  //
  // Note: 1) The next protocol field of the IP header in the clone net
  // buffer list could be AH/ESP, in which case the IP header must be
  // rebuilt (and AH/ESP header removed).
  //       2) The callout must re-calculate the IP checksum
```

```
  //        2) The callout must re-calculate the IP checksum.


  //
  // TODO: Allocate and populate classifyInfo by using information from
  // inFixedValues and inMetaValues.
  //


ASSERT(classifyInfo != NULL);

classifyInfo->aleInfo = TRUE;


classifyInfo->netBufferList = clonedNetBufferList;
clonedNetBufferList = NULL; // Ownership transferred.


status = FwpsPendOperation0(
inMetaValues->completionHandle,
              &classifyInfo->aleCompletionCtx);


if (!NT_SUCCESS(status))
  {
classifyOut->actionType = FWP_ACTION_BLOCK;
classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;


goto Exit;
  }


  //
  // TODO: Queue classifyInfo for out-of-band processing.
  //


classifyInfo = NULL; // Ownership transferred on success.


classifyOut->actionType = FWP_ACTION_BLOCK;
classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
classifyOut->flags |= FWPS_CLASSIFY_OUT_FLAG_ABSORB;


Exit:
```

```
 if (clonedNetBufferList != NULL)
   {
 FwpsFreeCloneNetBufferList0(clonedNetBufferList, 0);
   }
 if (classifyInfo)
   {
 if (classifyInfo->netBufferList)
      {
 FwpsFreeCloneNetBufferList0(classifyInfo->netBufferList, 0);
      }
      // TODO: Free object here.
   }


 return;
}


void
NTAPI
WfpTransportReceiveClassify(
   IN const FWPS_INCOMING_VALUES0* inFixedValues,
   IN const FWPS_INCOMING_METADATA_VALUES0* inMetaValues,
   IN OUT void* layerData,
   IN const FWPS_FILTER0* filter,
   IN UINT64 flowContext,
   OUT FWPS_CLASSIFY_OUT0* classifyOut
   )
{
   NTSTATUS status;


   NET_BUFFER_LIST* netBufferList = (NET_BUFFER_LIST*)layerData;
   NET_BUFFER_LIST* clonedNetBufferList = NULL;
   FWPS_PACKET_INJECTION_STATE injectionState;
 TL_ALE_RECV_CLASSIFY_INFO* classifyInfo = NULL;


 injectionState = FwpsQueryPacketInjectionState0(
 gInjectionHandle,
 netBufferList,
                       NULL);
 if (injectionState == FWPS_PACKET_INJECTED_BY_SELF ||
 injectionState == FWPS_PACKET_PREVIOUSLY_INJECTED_BY_SELF)
   {
 classifyOut->actionType = FWP_ACTION_PERMIT;
 goto Exit;
   }
```

```
 if (!(classifyOut->rights & FWPS_RIGHT_ACTION_WRITE))
   {
     //
     // Cannot alter the action.
     //
 goto Exit;
   }


   //
   // Let go of the packet if it requires ALE classify; the packet can
   // be inspected from the ALE receive/accept layer. Alternatively,
   // the callout can use the combination of
   // FWP_CONDITION_FLAG_REQUIRES_ALE_CLASSIFY and
   // FWP_MATCH_FLAGS_NONE_SET when you set up
   // filter conditions for the incoming transport layer.
   //
   // Beginning with Windows Vista SP1 and Windows Server 2008,
   // do not use FWP_CONDITION_FLAG_REQUIRES_ALE_CLASSIFY.
   // Use FWPS_IS_METADATA_FIELD_PRESENT macro to check for
   // metadata fields.
   //
#if (NTDDI_VERSION >= NTDDI_WIN6SP1)
 if (FWPS_IS_METADATA_FIELD_PRESENT(inMetaValues,
                                    FWPS_METADATA_FIELD_ALE_CLASSIFY_REQUIRED))
#else
 if ((inFixedValues->layerId == FWPS_LAYER_INBOUND_TRANSPORT_V4 &&
         (inFixedValues->incomingValue[FWPS_FIELD_INBOUND_TRANSPORT_V4_FLAGS].value.uint32 &
          FWP_CONDITION_FLAG_REQUIRES_ALE_CLASSIFY)) ||
        (inFixedValues->layerId == FWPS_LAYER_INBOUND_TRANSPORT_V6 &&
         (inFixedValues->incomingValue[FWPS_FIELD_INBOUND_TRANSPORT_V6_FLAGS].value.uint32 &
          FWP_CONDITION_FLAG_REQUIRES_ALE_CLASSIFY)))
#endif
   {
 classifyOut->actionType = FWP_ACTION_PERMIT;
 goto Exit;
   }
   //
   // Adjust the net buffer list offset so that it starts with the IP header.
   //
 NdisRetreatNetBufferDataStart(
     NET_BUFFER_LIST_FIRST_NB(netBufferList),
 inMetaValues->ipHeaderSize + inMetaValues->transportHeaderSize,
     0,
     NULL
     );
```

```
status = FwpsAllocateCloneNetBufferList0(
netBufferList,
            NULL,
            NULL,
            0,
            &clonedNetBufferList);


if (!NT_SUCCESS(status))
  {
classifyOut->actionType = FWP_ACTION_BLOCK;
classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;


goto Exit;
  }


  //
  // Undo the adjustment on the original net buffer list.
  //


NdisAdvanceNetBufferDataStart(
    NET_BUFFER_LIST_FIRST_NB(netBufferList),
inMetaValues->ipHeaderSize + inMetaValues->transportHeaderSize,
    FALSE,
    NULL);


  //
  // Notes: 1) The next protocol field of the IP header in the clone net
  // buffer list could be AH/ESP, in which case the IP header must be
  // rebuilt (and AH/ESP header removed).
  //        2) The callout must re-calculate the IP checksum.


  //
  // TODO: Allocate and populate classifyInfo by using information from
  // inFixedValues and inMetaValues.
  //


ASSERT(classifyInfo != NULL);

classifyInfo->aleInfo = FALSE;
```

```
 classifyInfo->netBufferList = clonedNetBufferList;
 clonedNetBufferList = NULL; // ownership transferred


   //
   // TODO: Queue classifyInfo for out-of-band processing.
   //


 classifyInfo = NULL; // Ownership transferred on success.


 classifyOut->actionType = FWP_ACTION_BLOCK;
 classifyOut->rights &= ~FWPS_RIGHT_ACTION_WRITE;
 classifyOut->flags |= FWPS_CLASSIFY_OUT_FLAG_ABSORB;


Exit:


 if (clonedNetBufferList != NULL)
   {
 FwpsFreeCloneNetBufferList0(clonedNetBufferList, 0);
   }
 if (classifyInfo)
   {
 if (classifyInfo->netBufferList)
     {
 FwpsFreeCloneNetBufferList0(classifyInfo->netBufferList, 0);
     }
     // TODO: Free object here.
   }


 return;
}
```

Stream Inspection

Inline Stream Inspection

流数据的修改可以通过在classifyFn函数中返回FWP_ACTION_PERMIT or FWP_ACTION_BLOCK,并修改FWPS_STREAM_CALLOUT_IO_PACKET0结构的countByteEnforced成员，进行执行流数据中的部分数据的许可或者阻塞，来编辑流数据。我们也可以调用FwpsStreamInjectAsync0函数给流增加新的内容。这些内容可以是新的或者可以替换被阻塞的数据。

为了去替换指定的段中的一部分p部分(比如，n byte + P byte + m byte)。

1，callout的classifyFn函数被调用时候数据为n+p+m bytes.

2, callout返回FWP_ACTION_PERMIT，将countBytesEnforced 设置为n.

3, callout的claffifyFn函数在数据为p+m bytes时候再次被调用，WFP发现countBytesEnforced 的值小于数据总量的时候会再次调用classifyFn函数。

4，classifyFn函数，callout调用FwpStreamInjectAsync0 函数注入替换p,然后callout返FWP_ACTION_BLOCK，将countBytesEnforced 设置为p.

5, callout的classifyFn函数被再次调用，这是m.

6, callout返回FWP_ACTION_BLOCK，将countBytesEnforced 设置为m.

如果指定的数据不足够让callout做查询的决定，它就应该设置FWPS_STREAM_CALLOUT_IO_PACKET0的streamAction成员设置为FWPS_STREAM_ACTION_NEED_MORE_DATA。设定在数据指定给callout之前，还需要设定countBytesRequired指定最少需要多少量的数据。当streamAction被设置，callout应该在classifyFn函数中返回FWP_ACTION_NONE.

当 FWPS_STREAM_ACTION_NEED_MORE_DATA 被设置，WFP可以积累到8MB的流数据。WFP在调用callout的classifyFn函数的时候，WFP将设置FWPS_CLASSIFY_OUT_FLAG_BUFFER_LIMIT_REACHED这个标志。如果FWPS_CLASSIFY_OUT_FLAG_NO_MORE_DATA标志被设置，callout不能返回FWPS_STREAM_ACTION_NEED_MORE_DATA。

为了流数据可以在平坦的空间中被扫描，WFP提供了FwpsCopyStreamDataToBuffer0函数，拷贝指定的数据到持续的空间中。

Out-of-Band Stream Inspection

对于外部的数据的检查或者修改，流callout和包检查callout类似，首先克隆所有的流数据延后处理，然后阻塞所有的段。检查或者修改的数据后面重新注入会数据流。当外部注入，callout必须返回FWP_ACTION_BLOCK在所有指定的段保证流结果的公平。外部检查模块禁止武端的插入一个FIN(声明从发送段不需要更多的数据)到发送的流中。
如果模块必须停止连接，它的classifyFn函数必须设置FWPS_STREAM_CALLOUT_IO_PACKET0结构体的成员streamAction 为FWPS_STREAM_ACTION_DROP_CONNECTION.

因为流数据可以作为NET_BUFFER_LIST链被显示，WFP提供了FwpsCloneStreamData0 and FwpsDiscardClonedStreamData0 操作网络空间列表链。

WFP也可以支持流数据来的方向的节约带宽。如果callout不能保持网络数据过来的速率，它应该返回FWPS_STREAM_ACTION_DEFER暂停流。流可以通过 FwpsStreamContinue0函数重新恢复。用这个函数延迟流的传输将导致TCP/IP协议栈停止对过来的数据的ACK.这将导致TCP的滑动窗口下降到0.

对于外部的流检查的callout，在FwpStreamInjectAsync0被调用的时候不能调用 FwpsStreamContinue0。

重新注入的流不能被callout重新指定声明，但是它可以被下层轻量级的子层再次应用。

Windows Server 2008 及以后的操作系统，在如下的流程中不能删除 [illegible] 的过滤器

Windows Server 2008 及以后的操作系统，在如下的流程中不能删除stream的过滤器。

callout正在执行外部包的注入。

callout正在请求更多的数据，通过设置FWPS_STREAM_CALLOUT_IO_PACKET0的成员streamAction为FWPS_STREAM_ACTION_NEED_MORE_DATA.

callout正在延迟流通过设置，通过设置FWPS_STREAM_CALLOUT_IO_PACKET0的成员streamAction为FWPS_STREAM_ACTION_DEFER。

动态流的检查

Windows 7及其以后的系统支持动态流的检查。动态流的检查操作在现有的流数据流程中，而不是创建或者拆除一个新的。可以支持动态流检查的callout驱动应该将FWPS_CALLOUT1 or FWPS_CALLOUT2 结构体中的Flags成员设置为FWP_CALLOUT_FLAG_ALLOW_MID_STREAM_INSPECTION 。

Avoiding Unnecessary Inspections

仅仅去执行那些驱动感兴趣的在连接上的流检查，callout可以设置FWPS_CALLOUT0的Flags成员为FWP_CALLOUT_FLAG_CONDITIONAL_ON_FLOW。这个callout将省略其他所有的连接。系统性能就被提供，驱动不需要维护一些不需要的数据状态。

Modifying Stream Data

当callout在流这一层处理数据的时候，它的classifyFn函数可以修改数据流中的数据。callout的classifyFn函数可以许可流中的数据原封不动的传递，阻塞并删除数据，注入新的数据到流中。

callout可以阻塞将要替换的数据，并用新的数据将其替换，同事，注入新的数据到流中。在这个条件下，将新数据注入到流中和阻塞删除流中的数据在同一个点上。

对于callout驱动注入数据到数据流中，它必须首先创建一个注入句柄，这跟为了修改包数据创建的注入handle是同一个，利用它将数据再次传递到网络协议栈中。

Data Logging

举例来说，如果callout需要追踪有多少IPV4包被过滤器在网络层丢弃，callout被加入到过滤引擎的FWPM_LAYER_INBOUND_IPPACKET_V4_DISCARD层。在这种条件下，callout的classifyFn函数可以按照如下的例子重新组装。

```
  // classifyFn callout function
VOID NTAPI
 ClassifyFn(
    IN const FWPS_INCOMING_VALUES0  *inFixedValues,
    IN const FWPS_INCOMING_METADATA_VALUES0  *inMetaValues,
    IN OUT VOID  *layerData,
    IN const FWPS_FILTER0  *filter,
    IN UINT64  flowContext
```

```
    IN UINT64 flowContext,
    OUT FWPS_CLASSIFY_OUT  *classifyOut
    )
{
  // Increment the total count of discarded packets
  InterlockedIncrement(&TotalDiscardCount);

  // Check whether a discard reason metadata field is present
 if (FWPS_IS_METADATA_FIELD_PRESENT(inMetaValues,
       FWPS_METADATA_FIELD_DISCARD_REASON))
  {
    // Check whether it is a general discard reason
    if (inMetaValues->discardMetadata.discardModule ==
       FWPS_DISCARD_MODULE_GENERAL)
    {
      // Check whether discarded by a filter
      if (inMetaValues->discardMetadata.discardReason ==
          FWPS_DISCARD_FIREWALL_POLICY)
      {
        // Increment the count of packets discarded by a filter
        InterlockedIncrement(&FilterDiscardCount);
      }
    }
  }

  // Take no action on the data
 classifyOut->actionType = FWP_ACTION_CONTINUE;
}
```

Associating Context with a Data Flow

对于在过滤层支持数据数据流的callout，callout驱动可以为每一个数据都关联一个上下文空间，这个空间对过滤引擎是不透明的。callout的classifyFn函数可以使用这个上下文保存指定数据流的状态信息。为了callout下次再次处理数据流使用。过滤引擎传递上下文是通过classifyFn函数的flowContext参数。如果没有指定上下文，flowContext参数就是0.

给数据流关联一个上下文空间，callout的classifyFn函数调用FwpsFlowAssociateContext0函数。举例来说：

```
 // Context structure to be associated with data flows
typedef struct FLOW_CONTEXT_ {
  .
  .  // Driver-specific content
  .
} FLOW_CONTEXT, *PFLOW_CONTEXT;


#define FLOW_CONTEXT_POOL_TAG 'fcpt'
```

```
// classifyFn callout function
VOID NTAPI
 ClassifyFn(
    IN const FWPS_INCOMING_VALUES0  *inFixedValues,
    IN const FWPS_INCOMING_METADATA_VALUES0  *inMetaValues,
    IN OUT VOID  *layerData,
    IN const FWPS_FILTER0  *filter,
    IN UINT64  flowContext,
    OUT FWPS_CLASSIFY_OUT  *classifyOut
    )
{
  PFLOW_CONTEXT context;
  UINT64 flowHandle;
  NTSTATUS status;


  ...


  // Check for the flow handle in the metadata
 if (FWPS_IS_METADATA_FIELD_PRESENT(
 inMetaValues,
        FWPS_METADATA_FIELD_FLOW_HANDLE))
  {
    // Get the flow handle
 flowHandle = inMetaValues->flowHandle;


    // Allocate the flow context structure
 context =
      (PFLOW_CONTEXT)ExAllocatePoolWithTag(
 NonPagedPool,
 sizeof(FLOW_CONTEXT),
        FLOW_CONTEXT_POOL_TAG
        );


    // Check the result of the memory allocation
 if (context == NULL) {

      // Handle memory allocation error
      ...
    }
 else
    {
```

```
      // Initialize the flow context structure
      ...


      // Associate the flow context structure with the data flow
 status = FwpsFlowAssociateContext0(
 flowHandle,
        FWPS_LAYER_INBOUND_IPPACKET_V4,
 calloutId,
        (UINT64)context
        );


      // Check the result
 if (status != STATUS_SUCCESS)
      {
        // Handle error
        ...
      }
    }
  }


  ...


}
```

如果数据已经关联了一个上下文空间，它必须在关联一个新的上下文空间之前，将之前关联的删掉。从数据流中删除一个上下文，classifyFn函数调用FwpsFlowRemoveContext0函数。

```
 // Context structure to be associated with data flows
typedef struct FLOW_CONTEXT_ {
  ...
} FLOW_CONTEXT, *PFLOW_CONTEXT;


#define FLOW_CONTEXT_POOL_TAG 'fcpt'


// classifyFn callout function
VOID NTAPI
 ClassifyFn(
    IN const FWPS_INCOMING_VALUES0  *inFixedValues,
    IN const FWPS_INCOMING_METADATA_VALUES0  *inMetaValues,
    IN OUT VOID  *layerData,
    IN const FWPS_FILTER0  *filter,
    IN UINT64  flowContext,
```

```
    OUT FWPS_CLASSIFY_OUT  *classifyOut
    )
{
  PFLOW_CONTEXT context;
  UINT64 flowHandle;
  NTSTATUS status;


  ...


  // Check for the flow handle in the metadata
 if (FWPS_IS_METADATA_FIELD_PRESENT(
 inMetaValues,
        FWPS_METADATA_FIELD_FLOW_HANDLE))
  {
    // Get the flow handle
 flowHandle = inMetaValues->flowHandle;


    // Check whether there is a context associated with the data flow
 if (flowContext != 0) {


      // Get a pointer to the flow context structure
 context = (PFLOW_CONTEXT)flowContext;


      // Remove the flow context structure from the data flow
 status = FwpsFlowRemoveContext0(
 flowHandle,
        FWPS_LAYER_INBOUND_IPPACKET_V4,
 calloutId
        );


      // Check the result
 if (status != STATUS_SUCCESS)
      {
        // Handle error
        ...
      }


      // Cleanup the flow context structure
      ...


      // Free the memory for the flow context structure
```

```
ExFreePoolWithTag(
context,
      FLOW_CONTEXT_POOL_TAG
      );
  }
 }


  ...


}
```

Processing Classify Callouts Asynchronously

WFP callout驱动可以在其classifyFn的callout函数中通过返回FWP_ACTION_PERMIT, FWP_ACTION_CONTINUE, or FWP_ACTION_BLOCK,来授权或否认网络操作，承认或丢弃网络包。通常，callout驱动不能在其classifyFn函数中返回一个检查决定直到一些信息被指定，比如分类的域，媒体数据，包，能够被转发到另一个部件进行处理，比如一个用户模式应用程序。在这种条件下，决定不得不在后面的时间异步做出。

异步处理的通用规则。

WFP支持classifyFn函数的异步处理，然而，在不同层次的其体系结构也不一样。

Asynchronous ALE Classify

在classifyFn函数中callout驱动必须调用FwpsPendOperation0.异步操作必须在调用FwpsCompleteOperation0函数。

Asynchronous Packet Classify

callout驱动应该在其classifyFn函数中返回FWP_ACTION_BLOCK,设置FWPS_CLASSIFY_OUT_FLAG_ABSORB标志。网络包必须被引用或者克隆。异步操作通过重新注入克隆或修改网络包或静默丢弃包来结束。

Asynchronous ALE Classify That Includes Packets

上面两种情况的组合，classify 操作被暂停而且包被引用或克隆，过段时间，classifyFn被结束，克隆的包被重新注入或者丢弃。

Special Cases and Considerations

ALE Connect vs. Receive/Accept Layers

当在ALE连接层(FWPS_LAYER_ALE_AUTH_CONNECT_V4 or FWPS_LAYER_ALE_AUTH_CONNECT_V6)上FwpsCompleteOperation0被调用去结束暂停的classify操作，ALE重新验证操作在各自的ALE连接层上被触发。

callout驱动应该为重新身份验证操作返回一个检查决定。你可以探测重新身份验证操作通过检查FWP_CONDITION_FLAG_IS_REAUTHORIZE 这个标志。

callout驱动必须为每一个在FwpsCompleteOperation0操作中触发的重新身份验证操作所做的检查决定维持一个唯一的暂停ALE_AUTH_CONNETCT的状态。如果包在暂停ALE_AUTH_CONNECT操作中被引用或克隆。(举例来说，非TCP连接)，它们可能被在重新身份验证后重新被注入。

当FwpsCompleteOperation0在ALE receive/accept层(FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4 or FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6)的分类操作中被调用,FwpsCompleteOperation0不会触发ALE的重新身份验证操作。而是，由于修改没有足够克隆包被重新注入classifyFn会被再调用一次。许可在ALE_RECV_ACCEPT层上自注入克隆对于接收到包是有效的授权。如果接收的连接不被允许，调用FwpsCompleteOperation0后将接收的包丢弃。

ALE Reauthorization

callout驱动可能在ALE connect或receive/accept层被重新分类，由于一些事件，比如政策改变(在层上增加或者删除过滤器)，发现新到的接口，使用IPSec重新连接。这些重新的身份验证在调用FwpsCompleteOperation0
不能被暂停，其实也不需要这样做。Callout驱动应该使用之前的流程来处理重新身份验证的情况。

对于接收到的和发出的包在ALE_AUTH_CONNECT或ALE_RECV_ACCEPT层上都可能被重新做身份验证。举例来说，接收到的包可能在ALE_AUTH_CONNECT层被重新身份验证。callout驱动不要假设包的流动方向和连接的方向一样。

ALE_FLOW_ESTABLISHED Layers

异步处理在这些层上不被支持(FWPS_LAYER_ALE_FLOW_ESTABLISHED_V4 or FWPS_LAYER_ALE_FLOW_ESTABLISHED_V6).

INBOUND_TRANSPORT Layers

Callout驱动不能够异步处理那些需要ALE分类处理来自传输层的包(FWPS_LAYER_INBOUND_TRANSPORT_V4 or FWPS_LAYER_INBOUND_TRANSPORT_V6).这样做会干涉到流的创建。当WFP在接收传输层上调用classifyFn函数，为了声明需要ALE分类处理，会设置FWPS_METADATA_FIELD_ALE_CLASSIFY_REQUIRED标志。callout驱动应该许可来自INBOUND_TRANSPORT层的包，并延缓其处理直到他们到达ALE_RECV_ACCETP层。

STREAM Layers

在数据流层(FWPS_LAYER_STREAM_V4 or FWPS_LAYER_STREAM_V6),TCP数据段被指定替换层IP或者TCP的头。数量流层可以在调用classifyFn函数中指定网络空间列表链的位置。WFP给出了指定的克隆和注入函数，FwpsCloneStreamData0 and FwpsStreamInjectAsync0给数据流层callout使用。

因为在数据流层上的数据分发的顺序的原因，callout驱动必须根据流数据被暂停的长度持续的克隆和吸收数据。混合的同步和异步操作将给数据流造成不可预知的行为。

Using Bind or Connect Redirection

WFP的连接或者重定位功能使能让ALE callout驱动可以检查，如果需要的话，可以重定位连接。

WFP的连接域管理定位功能使能在ALE callout驱动可以检查，如果需要的话，可以重定位连接。

在Windows 7之前，仅有的一种代理TCP连接的方式是，在存在的传输层上，对其每个包进行完成，丢弃，修改，重新注入。由于connect/bind 重定位的支持，callout驱动在TCP控制块(TCB)形成之前，修改任何的TCP 4元组。

因为绑定重定位的支持，在连接重定位中对本地地址和端口上的修改的支持不再需要了。修改连接重定位中的本地地址和端口不再支持。

Layers Used for Redirection

重定位可以被callout驱动在如下的层次上执行：

FWPS_LAYER_ALE_BIND_REDIRECT_V4 (FWPM_LAYER_ALE_BIND_REDIRECT_V4)

FWPS_LAYER_ALE_BIND_REDIRECT_V6 (FWPM_LAYER_ALE_BIND_REDIRECT_V6)

FWPS_LAYER_ALE_CONNECT_REDIRECT_V4 (FWPM_LAYER_ALE_CONNECT_REDIRECT_V4)

FWPS_LAYER_ALE_CONNECT_REDIRECT_V6 (FWPM_LAYER_ALE_CONNECT_REDIRECT_V6)

在哪一层上执行会有不同的影响范围。在连接层修改4元组，仅仅只会影响已经连接的流。绑定层的修改会影响所有的连接。

重定位层只在Windows 7以后的操作系统上适用。需要支持这种的callout，必须使用FwpsCalloutRegister1而不是FwpsCalloutRegister0注册。

注意：重定位不是适用于网络传输的所有的层。重定位支持的包的类型如下面列表所示：

TCP

UDP

Raw UDPv4 without the header include option

Raw ICMP

Performing Redirection

为了重定位连接，callout驱动必须得到一个可写的4-tuple信息的拷贝，做任何需要的修改来适应改变。一组新的函数被提供去得到writable layer数据，可以通过过滤引擎去使用。Callout驱动可以在classifyFn中做出修改，或者异步在另外的函数中做出修改。 为了使用classifyFn1，callout必须注册FwpsCalloutRegister1，而不是更老的FwpsCalloutRegister0.

为了去执行内联内部的重定位，callout驱动必须在classifyFn1函数中按如下步骤执行：

1，调用FwpsAcquireClassifyHandle0得到一个句柄，这个句柄将用于一些下面子函数的调用。

2，调用FwpsAcquireWritableLayerDataPointer0,为classifyFn1被调用的层获得一个可写数据结构的指针。输出参数writableLayerData的头响应这个层，是FWPS_BIND_REQUEST0或FWPS_CONNECT_REQUEST0.

注意：当修改FWPS_CONNECT_REQUEST0结构的数据重定位到loopback时候，callout驱动必须填充一个将接收连接的进程ID到localRedirectTargetPID成员中。

开始于Windows 8，你必须调用FwpsRedirectHandleCreate0去填充FWPS_CONNECT_REQUEST0结构的localRedirectHandle成员去完成一些代理工作。

3，修改这一层需要修改的数据。

4，调用FwpsApplyModifiedLayerData0去应用于修改的数据。

5，调用FwpsReleaseClassifyHandle0去释放在第一步中得到的句柄。

为了去执行异步重定位，callout必须执行如下的步骤。

1，调用FwpsAcquireClassifyHandle0得到一个句柄，这个句柄将用于一些下面子函数的调用。这一步，步骤2和3都在classifyFn函数中执行。

2，调用FwpsAcquireWritableLayerDataPointer0,为classifyFn1被调用的层获得一个可写数据结构的指针。输出参数writableLayerData的头响应这个层，是FWPS_BIND_REQUEST0或FWPS_CONNECT_REQUEST0.

注意：当修改FWPS_CONNECT_REQUEST0结构的数据重定位到loopback时候，callout驱动必须填充一个将接收连接的进程ID到localRedirectTargetPID成员中。

开始于Windows 8，你必须调用FwpsRedirectHandleCreate0去填充FWPS_CONNECT_REQUEST0结构的localRedirectHandle成员去完成一些代理工作。

3，调用FwpsPendClasify0将classification放入暂停的状态。

4，发送classificaiton的句柄和可写层的数据发送到另一个数据进行异步处理。下面的函数将在另外的函数中执行，而不是在classifyFn中执行。

5，对数据做出所需的修改。

6，调用FwpsApplyModifiedLayerData0应用修改的数据。

7，调用FwpsCompleteClassify0完成classify操作。

8，调用FwpsReleaseClassifyHandle0去释放在第一步中得到句柄。

Handling Connect Redirection from Multiple Callouts

可能不止一个callout驱动对同一个流进行连接重定位操作。执行连接重定位的callout应该知道其他的一些请求和相应的响应。使用连接重定位的callout驱动应该在ALE 身份验证连接层(FWPS_LAYER_ALE_AUTH_CONNECT_V4 or FWPS_LAYER_ALE_AUTH_CONNECT_V6)进行注册.当FWP_CONDITION_FLAG_IS_CONNECTION_REDIRECTED标志设置以后，为了查明检查如下的媒体数据值

FWP_CONDITION_FLAG_IS_CONNECTION_REDIRECTED标志设置以后，为了声明检查如下的媒体数据值。
FWPS_METADATA_FIELD_LOCAL_REDIRECT_TARGET_PID，包含一个响应重定位流的进程标识。
FWPS_METADATA_FIELD_ORIGINAL_DESTINATION，包含一个原始的目的地址。

为了监视重定位连接，callout驱动可以维护一个重定位流和对他们处理的列表。这个列表后面可以给callout驱动关于重定位的决定进行参考。

FWPS_CONNECT_REQUEST0结构体包含一个成员变量叫做localRedirectTargetPID.对于任何loopback连接重定位都是有效的，这个域必须由响应这个重定位流的进程PID来填充。这个值更引擎传递给ALE身份验证连接层的FWPS_METADATA_FIELD_LOCAL_REDIRECT_TARGET_ID.

开始于Windows 8,代理服务需要发出 SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS and SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT IOCTL,使用WSAIoctl,对于原始的代理服务。
另外，SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS IOCTL必须发出，使用WSAIoctl，在新的socket上。

ALE Endpoint Lifetime Management

对于支持ALE的callout驱动，可能需要为处理相应的事务分配资源。下面我们讲到当关联的端口被关闭已经怎样配置callout驱动去释放这些资源。ALE端口生命周期的支持开始Windows7.

为了去管理和ALE端口相关联的资源，callout驱动可以注册在如下的层。

FWPS_LAYER_ALE_RESOURCE_RELEASE_V4 (FWPM_LAYER_ALE_RESOURCE_RELEASE_V4)

FWPS_LAYER_ALE_RESOURCE_RELEASE_V6 (FWPM_LAYER_ALE_RESOURCE_RELEASE_V6)

FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V4 (FWPM_LAYER_ALE_ENDPOINT_CLOSURE_V4)

FWPS_LAYER_ALE_ENDPOINT_CLOSURE_V6 (FWPM_LAYER_ALE_ENDPOINT_CLOSURE_V6)

ALE资源释放层被声明共每个响应的ALE资源分配层使用(举例来说，FWPS_ALE_ENDPOINT_RESOURCE_ASSIGNMENT_V4).为了确定callout驱动的释放层和分配层相对应。
FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE于被提供在两层，每一个端口分配一个唯一的句柄。

ALE端点关闭层被调用不同是取决于端点的类型。对于TCP连接，ALE端点被关闭会同知道ALE验证的连接层(举例来说，FWPS_LAYER_ALE_AUTH_CONNECT_V4)或ALE验证的receive accept层(举例来说，FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4).通过ALE资源释放声明，引擎为每一个传递到FWPS_METADATA_FIELD_TRANSPORT_ENDPOINT_HANDLE 的端口分配唯一的句柄。对于不是TCP端口，ALE端口关闭层被调用对于每一个端口，无论远程socket连接的配对的多少。ALE端口关闭层也会为每一个TCP监听数据报被调用。

为ALE端口关闭层注册的callout可以被暂停操作。这就可以让callout在端口关闭前注入任何包队列进行异步处理。为了暂停，callout驱动必须调用FwpsPendClassify0，然后调用FwpsCompleteClassify0当处理完成的时候。

当应用的时候，引擎会为每一个在FWPS_METADATA_FIELD_PARENT_ENDPOINT_HANDLE域中的父端口指定一个唯一的句柄。这就可以，在需要的时候，callout驱动可以追踪父子之间的联系。

Processing Flow Delete Callouts

当正在处理的数据流由于callout被暂停，如果callout驱动预前有和数据流关联一个上下文，过滤引擎就会调用callout的flowDeleteFn函数。在数据流被暂停之前，callout的flowDeleteFn函数执行任何清除和数据相关联的上下文的操作。

```
 // Context structure to be associated with data flows
typedef struct FLOW_CONTEXT_ {
  ...
} FLOW_CONTEXT, *PFLOW_CONTEXT;


#define FLOW_CONTEXT_POOL_TAG 'fcpt'


// flowDeleteFn callout function
VOID NTAPI
 FlowDeleteFn(
    IN UINT16  layerId,
    IN UINT32  calloutId,
    IN UINT64  flowContext
    )
{
  PFLOW_CONTEXT context;


  // Get the flow context structure
 context = (PFLOW_CONTEXT)flowContext;


  // Cleanup the flow context structure
  ...


  // Free the memory for the flow context structure
 ExFreePoolWithTag(
 context,
    FLOW_CONTEXT_POOL_TAG
    );
}
```

当数据流被停止的时候，过滤引擎会自动和数据流相关联的上下文。因此，callout不需要在flowDeleteFn函数中调用FwpsFlowRemoveContext0函数删除数据的上下文空间。

Using Packet Tagging

callout可以对感兴趣的包做上标记，当标记的包发生相关的时间，callout驱动可以得到通知。包的标记支持从Windows 7及其以后的系统

支持从Windows 7及其以后的系统。

为了使用标记包，callout驱动必须实现FWPS_NET_BUFFER_LIST_NOTIFY_FN0回调函数。这个函数会接收到所有和标记的包相关的状态通知。在个人的包被标记之前，callout驱动可以调用FwpsNetBufferListGetTagForContext0得到指定的上下文标记。callout驱动可以为一些或者所有的已经标记的包使用同一个上下文标记。举例来说，callout驱动可能对不同类型的标记的包使用不同的上下文标记。

为了标记包，callout使用NET_BUFFER_LIST结构。callout驱动调用FwpsNetBufferListAssociateContext0去标记NET_BUFFER_LIST结构。callout驱动和包关联的上下文是一个64位的值。当相关的事件被触发，FWPS_NET_BUFFER_LIST_NOTIFY_FN0回调函数传递一个上下文空间值作为输入参数传递给callout驱动，callout驱动可以标识标记的包。过滤引擎不使用或者评估资格上下文。
仅仅只是传递给callout驱动。

当被标记的包被克隆或复制，callout驱动可以移动或者拷贝上下文到克隆或者复制的包里面。为了移动上下文(对于克隆包)，callout驱动必须调用 FwpsNetBufferListRemoveContext0，传递removeContext参数为TRUE.然后，上下文就可以和新的包相关联。对于拷贝上下文(复制包)是同样的操作，只是需要将FwpsNetBufferListRemoveContext0的参数removeContext设置为FALSE.

可以被触发的事件被定义在FWPS_NET_BUFFER_LIST_EVENT_TYPE0中。

在TCP/IP层包的触发可以被NDIS过滤驱动检索到。相反也是。在数据流层除了数据段没有其他的包被指定，包的触发是不能使用的。

callout驱动可以在 FWPS_NET_BUFFER_LIST_NOTIFY_FN0 函数的外部，调用FwpsNetBufferListRetrieveContext0得到包的上下文。通常，callout驱动在classifyFn回调函数中得到上下文。

Using Layer 2 Filtering

Layer 2层的过滤，从Windows 8开始支持。

WFP允许在Layer 2 MAC header上面过滤。这些层在所有对主机的包的接收或发送操作中都会被调用。这些层在接收的包在重新组装之间，发送的包在碎片化以后。这些层被从NDIS轻量级(LWF)过滤驱动进入.

注意：callout如果在那一层没有准备好的相应的过滤器，就不应该注入包。NET_BUFFER_LIST结构的注入应该和过滤器的增加和删除合作，当过滤器存在相应的层，因此注入仅仅被执行。

Injecting MAC Frames
Classifying Chained Network Buffer Lists
WFP Layer 2 Layers and Fields

Injecting MAC Frames

callout驱动调用 FwpsInjectMacReceiveAsync0 函数去重新注入预前吸收的MAC帧回到截获的接收路径中去，或者注入一个新的MAC帧在接收的数据路径中。

callout驱动调用 FwpsInjectMacSendAsync0 函数去重新注入预前吸收的MAC帧回到截获的发送路径中去，或者注入一个新的MAC帧在发送的数据路径中。netBufferLists参数可以是NET_BUFFER_LIST链。但是其完成例程在每一个链中的段结束的时候都会被调用。

如果注入的包和原始的过滤条件再次吻合，注入的帧可能被再次重新分类。因此，在IP层，Layer 2层的callout应该调用FwpsQueryPacketInjectState0保护无线循环。

当然，你必须在你注入的层有callout存在。否则，你注入的NET_BUFFER_LIST不会在你的完成例程中完成，NET_BUFFER_LIST将在栈中进一步上升。因为在这种情况下，行为是不被定义的，因为NDIS将尝试传递已经注入的NET_BUFFER_LIST到栈中的下一个部件。

NET_BUFFER_LIST中的Status成员包含栈注入的状态结果。栈注入状态结果是在WFP注入函数返回成功后NET_BUFFER_LIST放入栈中的状态。你应该使用NT_SUCCESS宏去检查栈注入状态的Status成员。如果Status值是STATUS_SUCCESS,注入成功没有其他的含义。Status成员的值比STATUS_SUCCESS值大意味着成功，但是还有需要其他的信息需要考虑。Status成员值比STATUS_SUCCESS值小，Status意味着注入失败的原因。

Classifying Chained Network Buffer Lists

一般来说，callout驱动仅仅只能独立分类网络空间列表。然而，callout驱动可以为了更好的表现分类网络空间列表链，通过如下的两个步骤。

指定FWPS_CALLOUT2结构中的Flags为FWP_CALLOUT_FLAG_ALLOW_L2_BATCH_CLASSIFY

注册classifyFn2函数指定NET_BUFFER_LIST链。

WFP Layer 2 Layers and Fields

实时过滤层对virtual switch filtering 包含：

FWPS_LAYER_INBOUND_MAC_FRAME_ETHERNET

FWPS_LAYER_OUTBOUND_MAC_FRAME_ETHERNET

FWPS_LAYER_INBOUND_MAC_FRAME_NATIVE

FWPS_LAYER_OUTBOUND_MAC_FRAME_NATIVE

数据区域对virtual switch filtering的标识

FWPS_FIELDS_INBOUND_MAC_FRAME_ETHERNET

FWPS_FIELDS_OUTBOUND_MAC_FRAME_ETHERNET

FWPS_FIELDS_INBOUND_MAC_FRAME_NATIVE

FWPS_FIELDS_OUTBOUND_MAC_FRAME_NATIVE

Using Proxied Connections Tracking

代理连接追踪支持于Windows 8及其以后的操作系统中。

WFP在追踪从重定位的远处到最终到目的地址的连接的所有的重定位记录

WFP有追踪从重定位的初始化到最终到目的地址的连接的所有的重定位记录。

Proxied Connections Tracking

现在的存在的多个代理，从一方的连接到最后的目的连接，可能被中间的另一方连接重定位，新的连接可能比被原始方重定位。没有连接的追踪，原始的连接可能从来都不能到达目的地，而陷入无限循环中。

增加数据域标识去支持连接追踪包括：

FWPS_FIELD_Xxx_ALE_ORIGINAL_APP_ID

代理连接的全路径原始应用。如果应用没有被代理，路径的标识为 xxx_ALE_APP_ID

FWPS_FIELD_Xxx_PACKAGE_ID

包的标识是一个安全标识符，它和AppContainer进程相关联。

Redirecting Connections

callout驱动可以调用 FwpsRedirectHandleCreate0 函数，去创建一个用于重定位TCP连接的句柄。

Using a Redirection Handle

在ALE连接重定位callout可以重定位连接到本地进程之前，它必须得到一个FwpsRedirectHandleCreate0函数创建的句柄，并将句柄放入FWPS_CONNECT_REQUEST0结构中。callout为ALE连接重定位层在其classifyFn修改这个结构。

localRedirectHandle 这个句柄是通过callout驱动调用FwpsRedirectHandleCreate0函数创建。

localRedirectContext callout上下文空间，是callout驱动调用ExAllocatePoolWithTag 进行的分配。

localRedirectContextSize callout提供的上下文空间的长度。

在callout驱动已经结束使用重定位的句柄的时候，可以调用 FwpsRedirectHandleDestroy0 函数释放句柄。

Querying the Redirect State

callout驱动调用 FwpsQueryConnectionRedirectState0 函数可以得到重定位连接的状态。FwpsQueryConnectionRedirectState0将返回一个类型为FWPS_CONNECTION_REDIRECT_STATE的枚举值
如果重定位状态为 FWPS_CONNECTION_NOT_REDIRECTED，ALE_CONNECT_REDIRECT callout可以处理代理连接。

如果重定位的状态为FWPS_CONNECTION_REDIRECTED_BY_SELF，ALE_CONNECT_REDIRECT callout应该返回FWP_ACTION_PERMIT/FWP_ACTION_CONTINUE如果重定位的状态为FWPS_CONNECTION_REDIRECTED_BY_OTHER，ALE_CONNECT_REDIRECT callout如果它不相信其他的检查结果，它可以处理代理连接。

如果重定位的状态为FWPS_CONNECTION_PREVIOUSLY_REDIRECTED_BY_SELF，ALE_CONNECT_REDIRECT callout不能执行重定位，甚至另外的检查结果也是不可接受的。在这种情况下，它必须许可或者阻止连接(在ALE_AUTH_CONNECT层)

Using Virtual Switch Filtering

Virtual Switch Filtering 开始支持与Window 8及其以后的操作系统中。

WFP允许在MAC头，IP头，和一些上层协议端口，比如虚拟接口，虚拟端口(VPort)，虚拟机标识(VM ID)上的过滤。当每一个基本的包传递到virtual switch的时候，这些层都将被调用。这些层可以被virtual switch extension filter-一种NDIS轻量级过滤驱动进入。

callout驱动可以调用 FwpsvSwitchEventsSubscribe0 函数为virtual switch层时间注册一些回调函数的入口点。
回调通知函数的入口点在 FWPS_VSWITCH_EVENT_DISPATCH_TABLE0 结构中指定。这些可以应用的回调函数包括：

FWPS_VSWITCH_FILTER_ENGINE_REORDER_CALLBACK0
FWPS_VSWITCH_INTERFACE_EVENT_CALLBACK0
FWPS_VSWITCH_LIFETIME_EVENT_CALLBACK0
FWPS_VSWITCH_POLICY_EVENT_CALLBACK0
FWPS_VSWITCH_PORT_EVENT_CALLBACK0
FWPS_VSWITCH_RUNTIME_STATE_RESTORE_CALLBACK0
FWPS_VSWITCH_RUNTIME_STATE_SAVE_CALLBACK0

FWPS_VSWITCH_EVENT_TYPE 枚举值定义了一些eventType参数值给virtual switch通知函数。

callout驱动必须调用 FwpsvSwitchEventsUnsubscribe0 去释放系统资源。

如果callout驱动在其WFP通知函数中返回STATUS_PENDING,WFP将返回STATUS_PENDING给OID请求。callout驱动必须调用 FwpsvSwitchNotifyComplete0 函数去结束暂停的操作。在 FwpsvSwitchNotifyComplete0 调用以后，WFP调用 NdisFOidRequestComplete 函数去结束对于virtual switch的OID请求。

Callout不应该在通知函数的上下文中同步的增加或删除WFP过滤器。另外，如果通知函数允许callback返回STATUS_PEINDING，callout返回STATUS_PENDING,callout不应该在结束通知之前增加或删除WFP过滤器。

WFP Virtual Switch Filter Layer and Fields

对于virtual switch filtering,实时过滤层包括：

FWPS_LAYER_INGRESS_VSWITCH_ETHERNET

FWPS_LAYER_EGRESS_VSWITCH_ETHERNET

FWPS_LAYER_INGRESS_VSWITCH_TRANSPORT_V4

FWPS_LAYER_INGRESS_VSWITCH_TRANSPORT_V6

FWPS_LAYER_EGRESS_VSWITCH_TRANSPORT_V4

FWPS_LAYER_EGRESS_VSWITCH_TRANSPORT_V6

对于virtual switch filtering，数据域标识包含；

FWPS_FIELDS_EGRESS_VSWITCH_ETHERNET

FWPS_FIELDS_EGRESS_VSWITCH_TRANSPORT_V4

FWPS_FIELDS_EGRESS_VSWITCH_TRANSPORT_V6

FWPS_FIELDS_INGRESS_VSWITCH_ETHERNET

FWPS_FIELDS_INGRESS_VSWITCH_TRANSPORT_V4

FWPS_FIELDS_INGRESS_VSWITCH_TRANSPORT_V6

Unloading a Callout Driver

为了卸载callout驱动，操作系统调用驱动的卸载函数。

在卸载函数中，在callout驱动被从系统内存卸载之前，callout驱动将给过滤引擎注册的callout进行卸载。callout驱动调用 FwpsCalloutUnregisterById0或FwpsCalloutUnregisterByKey0向过滤引擎卸载callout，callout驱动在其卸载函数中不应该返回，直到向过滤引擎成功卸载callout以后。

当callout驱动已从过滤引擎中卸载其callout，它必须删除其原始注册时候创建的设备对象。基于WDM的callout驱动，调用IoDeleteDevice函数去删除设备对象，基于WDF的callout驱动调用WdfObjectDelete函数去删除框架的设备对象。

callout驱动在其卸载函数返回之前也必须删除任何之前通过调用FwpsInjectionHandleDestroy0创建的包注入的句柄。

基于WMD的callout的卸载函数

```
// Device object
PDEVICE_OBJECT deviceObject;


// Variable for the run-time callout identifier
UINT32 CalloutId;


// Injection handle
HANDLE injectionHandle;
```

```
// Unload function
VOID
 Unload(
    IN PDRIVER_OBJECT DriverObject
    )
{
  NTSTATUS status;


  // Unregister the callout
 status =
 FwpsCalloutUnregisterById0(
 CalloutId
      );


  // Check result
 if (status == STATUS_DEVICE_BUSY)
  {
    // For each data flow that is being processed by the
    // callout that has an associated context, clean up
    // the context and then call FwpsFlowRemoveContext0
    // to remove the context from the data flow.
    ...


    // Finish unregistering the callout
 status =
 FwpsCalloutUnregisterById0(
 CalloutId
       );
  }


  // Check status
 if (status != STATUS_SUCCESS)
  {
    // Handle error
    ...
  }


  // Delete the device object
 IoDeleteDevice(
 deviceObject
    );
```

```
 // Destroy the injection handle
 status =
 FwpsInjectionHandleDestroy0(
 injectionHandle
      );


 // Check status
 if (status != STATUS_SUCCESS)
 {
   // Handle error
   ...
 }
}
```

基于WDF的卸载函数。

```
WDFDEVICE wdfDevice;


VOID
 Unload(
   IN WDFDRIVER Driver;
   )
{


  ...


 // Delete the framework device object
 WdfObjectDelete(
 wdfDevice
   );


  ...
}
```

INF Files for Callout Drivers

Windows Filtering Platform Callout驱动通过INF文件安装。对于callout驱动的INF文件包含如下的INF文件节：

INF Version Section

INF SourceDisksNames Section

INF SourceDisksFiles Section

INF DestinationDirs Section

INF DefaultInstall Section

INF DefaultInstall.Services Section

INF Strings Section

;
; Example callout driver INF file
;

```
[Version]
Signature = "$Windows NT$"
Provider = %Msft%
CatalogFile = "ExampleCalloutDriver.cat"
DriverVer = 01/15/05,1.0


[SourceDiskNames]
1 = %DiskName%


[SourceDiskFiles]
ExampleCalloutDriver.sys = 1


[DestinationDirs]
DefaultDestDir = 12 ; %windir%\system32\drivers
ExampleCalloutDriver.DriverFiles = 12 ; %windir%\system32\drivers


[DefaultInstall]
OptionDesc = %Description%
CopyFiles = ExampleCalloutDriver.DriverFiles


[DefaultInstall.Services]
AddService = %ServiceName%,,ExampleCalloutDriver.Service


[DefaultUninstall]
```

```
DelFiles = ExampleCalloutDriver.DriverFiles


[DefaultUninstall.Services]
DelService = ExampleCalloutDriver,0x200 ; SPSVCINST_STOPSERVICE


[ExampleCalloutDriver.DriverFiles]
ExampleCalloutDriver.sys,,,0x00000040 ; COPYFLG_OVERWRITE_OLDER_ONLY


[ExampleCalloutDriver.Service]
DisplayName = %ServiceName%
Description = %ServiceDesc%
ServiceType = 1  ; SERVICE_KERNEL_DRIVER
StartType = 0    ; SERVICE_BOOT_START
ErrorControl = 1 ; SERVICE_ERROR_NORMAL
ServiceBinary = %12%\ExampleCalloutDriver.sys


[Strings]
%Msft% = "Microsoft Corporation"
%DiskName% = "Example Callout Driver Installation Disk"
%Description% = "Example Callout Driver"
%ServiceName% = "ExampleCalloutDriver"
%ServiceDesc% = "Example Callout Driver"
```

Installation of Callout Drivers

callout驱动可以通过右击INF文件安装。

在callout驱动已经被成功安装了，它可以通过如下的命令加载。

```
net start drivername
```

取决于inf文件中[drivername.Service]的节的StartType值，callout驱动可能在系统重新启动后自动被加载到系统中，callout驱动应该通常应该指定这个值为0(SERVICE_BOOT_START)，在过滤引擎开始之前，加载驱动和注册callout驱动。。

callout驱动的卸载可以通过如下的命令：

```
net stop drivername
```

当callout驱动已经被安装和加载，它将出现在设备管理器中。设备管理器可以被用来启动，停止，卸载驱动callout驱动。Callout驱动在设备管理器的非即插即用类别中显示，
默认不会显示出来。

callout驱动也可以通过Win32的服务管理器的API，来安装，加载，卸载。CreateService, OpenService, StartService, ControlService, and DeleteService

Digital Signatures for Callout Drivers

关于数字签名，更其他的驱动一样。

Callout Driver Programming Considerations

User Mode vs. Kernel Mode

如果过滤操作可以通过标准的Windows Filtering Platform完成，ISV 应该考虑开发一个用户模式的管理程序去配置过滤引擎，而不是去开发一个内核模式的callout驱动。开发内核模式的callout驱动只在你需要完成的功能，不能通过已经内置的标准过滤功能完成的情况下选择。

Choice of Filtering Layer

callout驱动应该在网络堆栈的最上层的过滤层完成网络数据的过滤。举例来说，如果发配的过滤任务可以在数据流层完成，就不应该在网络层进行执行。

Blocking at the Application Layer Enforcement (ALE) Flow Established Layers

通常，如果已经加入到过滤引擎的callout ALE flow established filtering layers，(FWPM_LAYER_ALE_FLOW_ESTABLISHED_V4 or FWPM_LAYER_ALE_FLOW_ESTABLISHED_V6),它的classifyFn callout函数不应该返回ACTION_BLOCK。一个身份验证或拒绝连接的决定不应该在ALE flow established filtering layers上做出。这些决定应该在另外的ALE过滤层上做出。

究其原因是classifyFn函数返回FWP_ACTION_BLOCK会有潜在的安全隐患在建立连接还没有结束的情况下。

Callout Function Execution Time

因为过滤引起通常调用callout的函数在IRQL=DISPATCH_LEVEL上，确保这些函数的执行尽可能的快，让系统更有效率。

Injecting Into the Receive Data Path

callout应该在调用包注入函数送入到接收数据路径之前重新计算IP的checksum。因为当包被重新组装后IP包段的原始的checksum的值可能不正确。

Inline Injection of TCP Packet from Transport Layers

因为TCP栈的锁定行为，传输层的callout不能在classifyFn中注入一个新的或者克隆的TCP包。如果是内联注入，callout必须排队一个DPC去处理。

Outgoing IP Header Alignment

描述网络空间列表中的IP头的MDL应该和包注入函数的注入到发送路径的包数据应该指针对齐。因为，结束的IP头的包MDL可能是指针对齐的，callout必须重新建立一个IP头(如果没有对齐)当注入一个接收包到一个

结束的IP头的包NBL可能是指针对齐的，callout必须重新建立一个IP头(如果没有对齐)当注入一个接收包到一个发送路径中。

Developing IPsec-Compatible Callout Drivers

为了去全部兼容开始于Windows Vista和Windows 2008的IPSec，callout驱动应该被如下的实时过滤层。

TCP Packet Filtering

Stream Layers:

FWPS_LAYER_STREAM_V4

FWPS_LAYER_STREAM_V6

Non-TCP and Non-Error ICMP Packet Filtering

Datagram-Data Layers:

FWPS_LAYER_DATAGRAM_DATA_V4

FWPS_LAYER_DATAGRAM_DATA_V6

FWPS_LAYER_DATAGRAM_DATA_V4_DISCARD

FWPS_LAYER_DATAGRAM_DATA_V6_DISCARD

当这些包被从数据包数据层被接收注入之前，包必须重新构造的条件之外。在这些层注册的callout驱动和IPsec就是兼容的。

Layers That Are Incompatible With IPsec

网络和转发层和IPSec是不兼容的，因为在这些层IPsec传输还没有被解密或检查。IPSec政策在网络层分类以后，在传输层是强制操作的。

如下的实时过滤层和IPSec是不兼容的，因为IPsec将要在这些层处理。

FWPS_LAYER_INBOUND_IPPACKET_V4

FWPS_LAYER_INBOUND_IPPACKET_V6

FWPS_LAYER_INBOUND_IPPACKET_V4_DISCARD

FWPS_LAYER_INBOUND_IPPACKET_V6_DISCARD

FWPS_LAYER_OUTBOUND_IPPACKET_V4

FWPS_LAYER_OUTBOUND_IPPACKET_V6

FWPS_LAYER_OUTBOUND_IPPACKET_V4_DISCARD

FWPS_LAYER_OUTBOUND_IPPACKET_V6_DISCARD

Special Considerations for Transport Layers

为了在传输层(FWPS_LAYER_XXX_TRANSPORT_V4 or _V6)注册的callout驱动和IPsec兼容，遵照如下的规则：

1，在ALE receive/accept层(FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4 or FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6)注册callout,除了传输层(FWPS_LAYER_XXX_TRANSPORT_V4 or _V6)

2, 为了避免Windows内部IPSec处理的干扰，把callout注册在比FWPM_SUBLAYER_UNIVERSAL更轻量级的子层。使用 FwpmSubLayerEnum0 得到子层的重量。

3，对于接收到的传输层过来的需要ALE分类的包，必须在ALE authorize receive/accept layers (FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V4 or FWPS_LAYER_ALE_AUTH_RECV_ACCEPT_V6层检查。这些包在过来的传输层必须得到许可。

开始于vista sp1和windows 2008，使用FWPS_METADATA_FIELD_ALE_CLASSIFY_REQUIRED标志去探测是否接收的包是否被指定FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V4或FWPM_LAYER_ALE_AUTH_RECV_ACCEPT_V6层。这个标志在Windows Vista中被FWP_CONDITION_FLAG_REQUIRES_ALE_CLASSIFY条件标志替换。

4，为了避免对Windows 内部IPSec的干扰，不要在传输层在IPsec传输还没有移除隧道的时候对IPSec 隧道模式的传输进行截断，下面的代码演示如何跳过这些包。

```
  FWPS_PACKET_LIST_INFORMATION0 packetInfo = {0};


FwpsGetPacketListSecurityInformation0(
 layerData,
    FWPS_PACKET_LIST_INFORMATION_QUERY_IPSEC |
    FWPS_PACKET_LIST_INFORMATION_QUERY_INBOUND,
    &packetInfo
```

```
    );


if (packetInfo.ipsecInformation.inbound.isTunnelMode &&
    !packetInfo.ipsecInformation.inbound.isDeTunneled)
{
 classifyOut->actionType = FWP_ACTION_PERMIT;
 goto Exit;
}
```

5，在IPSec保护的包在传输已经被解密和检查以后，AH/ESP头将保留在IP头中。如果这样的包不得不重新注入回到TCP/IP的协议栈中，IP头必须删除掉AH/ESP进行重新构造。
开始于Windows Vista SP1和Windows Server 2008以后，你可以通过克隆包并调用 FwpsConstructIpHeaderForTransportPacket0 函数设置headerIncludeHeaderSize去设置IP克隆包的头的大小来完成。

6，在ALE receive/accept 层，callout可以通过检查FWP_CONDITION_FLAG_IS_IPSEC_SECURED标志是否设置来探测IPSec的保护传输。在传输层，callout可以通过调用 FwpsGetPacketListSecurityInformation0 函数去检查queryFlags参数是否被设置为 FWPS_PACKET_LIST_INFORMATION0来探测IPSec保护传输。

Working With IPsec ESP Packets

当引擎指定解密decrypted encapsulating security payload(ESP)包，将会截断排除尾随ESP数据。因为引起处理这些包的方式，NET_BUFFER结构中的MDL数据不能正确反映包的长度。正确的长度可以在NET_BUFFER结构的DATA_LENGTH中得到。

Calling Other Windows Filtering Platform Functions

很多另外的Windows Filtering Platform 函数可以被用户模式的应用程序使用，也可以被callout驱动使用。这就让callout驱动也可以去管理，比如，增加过滤器到过滤引擎的任务。唯一的不同是这些函数的返回值不一样，用户模式的函数返回WIN32的错误代码，内核模式函数返回NTSTATUS代码。

大多数Windows Filtering Platform管理函数需要一个句柄作为参数去打开和过滤引擎的会话。

Opening a Session to the Filter Engine

callout驱动必须打开一个到过滤引擎的会话，去执行一些比如增加过滤器到过滤引擎的任务。callout驱动可以通过FwpmEngineOpen0函数去打开一个到过滤引擎的会话。

```
 HANDLE engineHandle;
NTSTATUS status;


// Open a session to the filter engine
status =
 FwpmEngineOpen0(
    NULL,             // The filter engine on the local system
    RPC_C_AUTHN_WINNT, // Use the Windows authentication service
```

```
    RPC_C_AUTHN_WINNT,// use the Windows authentication service
    NULL,             // Use the calling thread's credentials
    NULL,             // There are no session-specific parameters
    &engineHandle     // Pointer to a variable to receive the handle
    );
```

当callout成功打开一个到过滤引擎的会话以后，它就可以使用返回的句柄去调用另外的Windows Filtering Platform管理函数。

Closing a Session to the Filter Engine

当callout驱动已经完成了管理任务，它应该关闭到过滤引擎的会话。callout驱动可以通过调用FwpmEngineClose0函数完成这个功能。

```
status =
 FwpmEngineClose0(
 engineHandle  // An handle to the open session
    );
```