# WFP框架是微软推出来替代TDIHOOK传输层驱动接口网络通信的方案，其默认被设计为分层结构，该框架分别提供了用户态与内核态相同的AIP函数，在两种模式下均可以开发防火墙产品，以下代码我实现了 一个简单的驱动过滤防火墙。

WFP 框架分为两大层次模块,用户态基础过滤引擎 BFE (BaseFilteringEngine) ,以及内核态过滤引擎KMFE (KMFilteringEngine) ,基础过滤引擎对上提供C语言调用方式的API以及RPC接口,这些接口都被封装在 FWPUCLNT.dll 模块中,开发时可以调用该模块中的导出函数.

# WFP程序工作流程:

使用 FwpmEngineOpen() 开启 WFP 引擎,获得WFP使用句柄

使用 FwpmTransactionBegin() 设置对网络通信内容的过滤权限 (只读/允许修改)

使用 FwpsCalloutRegister(),FwpmCalloutAdd(),FwpmFilterAdd() 选择要过滤的内容,并添加过滤器对象和回调函数.

使用 FwpmTransactionCommit() 确认刚才的内容,让刚才添加的回调函数开始生效.

使用 FwpmFilterDeleteById(),FwpmCalloutDeleteById(),FwpsCalloutUnregisterById()函数撤销对象和回调函数.

使用 FwpmEngineClose() 关闭WFP引擎类句柄.

默认情况下WFP一次需要注册3个回调函数,只有一个是事前回调,另外两个是事后回调,通常情况下我们只关注事前回调即可,此外WFP能过滤很对内容,我们需要指定过滤条件标志来输出我们所需要的数据.

一般可设置为 FWPM\_LAYER\_ALE\_AUTH\_CONNECT\_V4 意思是设置IPV4过滤.

还需要设置一个GUID值,该值可随意设置,名称为 GUID\_ALE\_AUTH\_CONNECT\_CALLOUT\_V4 宏.

# 首先我们通过上方的流程实现一个简单的网络控制驱动，该驱动运行后可对自身机器访问指定地址端口 进行控制，例如实现指定应用断网，禁止指定页面被访问等，在配置WFP开发环境时需要在链接器选项 卡中的附加依赖项中增加 fwpkclnt.lib，uuid.lib 这两个库文件，并且需要使用WDM开发模板，否则编译将不通过。

// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#define NDIS\_SUPPORT\_NDIS6 1

#define DEV\_NAME L"\\Device\\MY\_WFP\_DEV\_NAME" #define SYM\_NAME L"\\DosDevices\\MY\_WFP\_SYM\_NAME"

#include <ntifs.h> #include <fwpsk.h> #include <fwpmk.h> #include <stdio.h>

// 过滤器引擎句柄

HANDLE g\_hEngine;

// 过滤器引擎中的callout的运行时标识符

ULONG32 g\_AleConnectCalloutId;

// 过滤器的运行时标识符

ULONG64 g\_AleConnectFilterId;

// 指定唯一UUID值(只要不冲突即可,内容可随意)



GUID GUID\_ALE\_AUTH\_CONNECT\_CALLOUT\_V4 = { 0x6812fc83, 0x7d3e, 0x499a, 0xa0, 0x12,

0x55, 0xe0, 0xd8, 0x5f, 0x34, 0x8b };

// ------------------------------------------------------------------------------

// 头部函数声明

// ------------------------------------------------------------------------------

// 注册Callout并设置过滤点

NTSTATUS RegisterCalloutForLayer( IN PDEVICE\_OBJECT pDevObj,

IN const GUID layerKey, IN const GUID calloutKey,

IN FWPS\_CALLOUT\_CLASSIFY\_FN classifyFn,

IN FWPS\_CALLOUT\_NOTIFY\_FN notifyFn,

IN FWPS\_CALLOUT\_FLOW\_DELETE\_NOTIFY\_FN flowDeleteNotifyFn, OUT ULONG32 calloutId,

OUT ULONG64 filterId,

OUT HANDLE engine);

// 注册Callout

NTSTATUS RegisterCallout( PDEVICE\_OBJECT pDevObj,

IN const GUID calloutKey,

IN FWPS\_CALLOUT\_CLASSIFY\_FN classifyFn, IN FWPS\_CALLOUT\_NOTIFY\_FN notifyFn,

IN FWPS\_CALLOUT\_FLOW\_DELETE\_NOTIFY\_FN flowDeleteNotifyFn, OUT ULONG32 calloutId);

// 设置过滤点

NTSTATUS SetFilter(

IN const GUID layerKey, IN const GUID calloutKey, OUT ULONG64 filterId,

OUT HANDLE engine);

// Callout函数 flowDeleteFn VOID NTAPI flowDeleteFn(

\_In\_ UINT16 layerId,

\_In\_ UINT32 calloutId,

\_In\_ UINT64 flowContext

);

// Callout函数 classifyFn

#if (NTDDI\_VERSION >= NTDDI\_WIN8)

VOID NTAPI classifyFn(

\_In\_ const FWPS\_INCOMING\_VALUES0 inFixedValues,

\_In\_ const FWPS\_INCOMING\_METADATA\_VALUES0 inMetaValues,

\_Inout\_opt\_ void layerData,

\_In\_opt\_ const void classifyContext,

\_In\_ const FWPS\_FILTER2 filter,

\_In\_ UINT64 flowContext,

\_Inout\_ FWPS\_CLASSIFY\_OUT0 classifyOut

);

#elif (NTDDI\_VERSION >= NTDDI\_WIN7)

VOID NTAPI classifyFn(

\_In\_ const FWPS\_INCOMING\_VALUES0 inFixedValues,



\_In\_ const FWPS\_INCOMING\_METADATA\_VALUES0 inMetaValues,

\_Inout\_opt\_ void layerData,

\_In\_opt\_ const void classifyContext,

\_In\_ const FWPS\_FILTER1 filter,

\_In\_ UINT64 flowContext,

\_Inout\_ FWPS\_CLASSIFY\_OUT0 classifyOut

);

#else

VOID NTAPI classifyFn(

\_In\_ const FWPS\_INCOMING\_VALUES0 inFixedValues,

\_In\_ const FWPS\_INCOMING\_METADATA\_VALUES0 inMetaValues,

\_Inout\_opt\_ void layerData,

\_In\_ const FWPS\_FILTER0 filter,

\_In\_ UINT64 flowContext,

\_Inout\_ FWPS\_CLASSIFY\_OUT0 classifyOut

);

#endif

// Callout函数 notifyFn

#if (NTDDI\_VERSION >= NTDDI\_WIN8)

NTSTATUS NTAPI notifyFn(

\_In\_ FWPS\_CALLOUT\_NOTIFY\_TYPE notifyType,

\_In\_ const GUID filterKey,

\_Inout\_ FWPS\_FILTER2 filter

);

#elif (NTDDI\_VERSION >= NTDDI\_WIN7)

NTSTATUS NTAPI notifyFn(

\_In\_ FWPS\_CALLOUT\_NOTIFY\_TYPE notifyType,

\_In\_ const GUID filterKey,

\_Inout\_ FWPS\_FILTER1 filter

);

#else

NTSTATUS NTAPI notifyFn(

\_In\_ FWPS\_CALLOUT\_NOTIFY\_TYPE notifyType,

\_In\_ const GUID filterKey,

\_Inout\_ FWPS\_FILTER0 filter

);

#endif

// ------------------------------------------------------------------------------

// 函数实现部分

// ------------------------------------------------------------------------------

// 协议判断

NTSTATUS ProtocalIdToName(UINT16 protocalId, PCHAR lpszProtocalName)

{

NTSTATUS status = STATUS\_SUCCESS;

switch (protocalId)

{

case 1:

{

// ICMP

RtlCopyMemory(lpszProtocalName, "ICMP", 5); break;

}

case 2:

{

// IGMP

RtlCopyMemory(lpszProtocalName, "IGMP", 5); break;

}

case 6:

{

// TCP

RtlCopyMemory(lpszProtocalName, "TCP", 4); break;

}

case 17:

{

// UDP

RtlCopyMemory(lpszProtocalName, "UDP", 4); break;

}

case 27:

{

// RDP

RtlCopyMemory(lpszProtocalName, "RDP", 6); break;

}

default:

{

// UNKNOW

RtlCopyMemory(lpszProtocalName, "UNKNOWN", 8); break;

}

}

return status;

}

// 启动WFP

NTSTATUS WfpLoad(PDEVICE\_OBJECT pDevObj)

{

NTSTATUS status = STATUS\_SUCCESS;

// 注册Callout并设置过滤点

// classifyFn, notifyFn, flowDeleteFn 注册三个回调函数,一个事前回调,两个事后回调

status = RegisterCalloutForLayer(pDevObj, &FWPM\_LAYER\_ALE\_AUTH\_CONNECT\_V4, &GUID\_ALE\_AUTH\_CONNECT\_CALLOUT\_V4,

classifyFn, notifyFn, flowDeleteFn, &g\_AleConnectCalloutId,

&g\_AleConnectFilterId, &g\_hEngine); if (!NT\_SUCCESS(status))

{

DbgPrint("注册回调失败 \n"); return status;

}

return status;

}

// 卸载WFP



NTSTATUS WfpUnload()

{

if (NULL != g\_hEngine)

{

// 删除FilterId

FwpmFilterDeleteById(g\_hEngine, g\_AleConnectFilterId);

// 删除CalloutId

FwpmCalloutDeleteById(g\_hEngine, g\_AleConnectCalloutId);

// 清 空 Filter g\_AleConnectFilterId = 0;

// 反 注 册 CalloutId FwpsCalloutUnregisterById(g\_AleConnectCalloutId);

// 清空CalloutId

g\_AleConnectCalloutId = 0;

// 关闭引擎FwpmEngineClose(g\_hEngine); g\_hEngine = NULL;

}

return STATUS\_SUCCESS;

}

// 注册Callout并设置过滤点

NTSTATUS RegisterCalloutForLayer(IN PDEVICE\_OBJECT pDevObj, IN const GUID layerKey, IN const GUID calloutKey, IN FWPS\_CALLOUT\_CLASSIFY\_FN classifyFn, IN

FWPS\_CALLOUT\_NOTIFY\_FN notifyFn, IN FWPS\_CALLOUT\_FLOW\_DELETE\_NOTIFY\_FN

flowDeleteNotifyFn, OUT ULONG32 calloutId, OUT ULONG64 filterId, OUT HANDLE engine)

{

NTSTATUS status = STATUS\_SUCCESS;

// 注册Callout

status = RegisterCallout(pDevObj, calloutKey, classifyFn, notifyFn, flowDeleteNotifyFn, calloutId);

if (!NT\_SUCCESS(status))

{

return status;

}

// 设置过滤点

status = SetFilter(layerKey, calloutKey, filterId, engine); if (!NT\_SUCCESS(status))

{

return status;

}

return status;

}

// 注册Callout

NTSTATUS RegisterCallout(PDEVICE\_OBJECT pDevObj, IN const GUID calloutKey, IN FWPS\_CALLOUT\_CLASSIFY\_FN classifyFn, IN FWPS\_CALLOUT\_NOTIFY\_FN notifyFn, IN FWPS\_CALLOUT\_FLOW\_DELETE\_NOTIFY\_FN flowDeleteNotifyFn, OUT ULONG32 calloutId)

{

NTSTATUS status = STATUS\_SUCCESS;

FWPS\_CALLOUT sCallout = { 0 };



// 设置Callout

sCallout.calloutKey = calloutKey; sCallout.classifyFn = classifyFn; sCallout.flowDeleteFn = flowDeleteNotifyFn; sCallout.notifyFn = notifyFn;

// 注册Callout

status = FwpsCalloutRegister(pDevObj, &sCallout, calloutId); if (!NT\_SUCCESS(status))

{

DbgPrint("注册Callout失败 \n"); return status;

}

return status;

}

// 设置过滤点

NTSTATUS SetFilter(IN const GUID layerKey, IN const GUID calloutKey, OUT ULONG64 filterId, OUT HANDLE engine)

{

HANDLE hEngine = NULL;

NTSTATUS status = STATUS\_SUCCESS;

FWPM\_SESSION session = { 0 }; FWPM\_FILTER mFilter = { 0 }; FWPM\_CALLOUT mCallout = { 0 }; FWPM\_DISPLAY\_DATA mDispData = { 0 };

// 创建Session

session.flags = FWPM\_SESSION\_FLAG\_DYNAMIC;

status = FwpmEngineOpen(NULL, RPC\_C\_AUTHN\_WINNT, NULL, &session, &hEngine); if (!NT\_SUCCESS(status))

{

return status;

}

// 开始事务

status = FwpmTransactionBegin(hEngine, 0); if (!NT\_SUCCESS(status))

{

return status;

}

// 设置Callout参数

mDispData.name = L"MY WFP LyShark"; mDispData.description = L"WORLD OF DEMON"; mCallout.applicableLayer = layerKey; mCallout.calloutKey = calloutKey; mCallout.displayData = mDispData;

// 添加Callout到Session中

status = FwpmCalloutAdd(hEngine, &mCallout, NULL, NULL); if (!NT\_SUCCESS(status))

{

return status;



}

// 设置过滤器参数

mFilter.action.calloutKey = calloutKey; mFilter.action.type = FWP\_ACTION\_CALLOUT\_TERMINATING; mFilter.displayData.name = L"MY WFP LyShark"; mFilter.displayData.description = L"WORLD OF DEMON"; mFilter.layerKey = layerKey;

mFilter.subLayerKey = FWPM\_SUBLAYER\_UNIVERSAL; mFilter.weight.type = FWP\_EMPTY;

// 添加过滤器

status = FwpmFilterAdd(hEngine, &mFilter, NULL, filterId); if (!NT\_SUCCESS(status))

{

return status;

}

// 提交事务

status = FwpmTransactionCommit(hEngine); if (!NT\_SUCCESS(status))

{

return status;

}

engine = hEngine; return status;

}

// Callout函数 classifyFn 事前回调函数

VOID NTAPI classifyFn(\_In\_ const FWPS\_INCOMING\_VALUES0 inFixedValues, \_In\_ const FWPS\_INCOMING\_METADATA\_VALUES0 inMetaValues, \_Inout\_opt\_ void layerData,

\_In\_opt\_ const void classifyContext, \_In\_ const FWPS\_FILTER2 filter, \_In\_ UINT64 flowContext, \_Inout\_ FWPS\_CLASSIFY\_OUT0 classifyOut)

{

// 数据包的方向,取值 FWP\_DIRECTION\_INBOUND = 1 或 FWP\_DIRECTION\_OUTBOUND = 0

WORD wDirection = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_FLOW\_ESTABLISHED\_V4\_DIRECTION].value.int8;

// 定义本机地址与本机端口

ULONG ulLocalIp = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_LOCAL\_ADDRESS].value.uint32; UINT16 uLocalPort = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_LOCAL\_PORT].value.uint16;

// 定义对端地址与对端端口

ULONG ulRemoteIp = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_REMOTE\_ADDRESS].value.uint32; UINT16 uRemotePort = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_REMOTE\_PORT].value.uint16;

// 获取当前进程IRQ

KIRQL kCurrentIrql = KeGetCurrentIrql();

// 获取进程ID

ULONG64 processId = inMetaValues->processId; UCHAR szProcessPath[256] = { 0 };

CHAR szProtocalName[256] = { 0 };

RtlZeroMemory(szProcessPath, 256);

// 获取进程路径

for (ULONG i = 0; i < inMetaValues->processPath->size; i++)

{

// 里面是宽字符存储的

szProcessPath[i] = inMetaValues->processPath->data[i];

}

// 获取当前协议类型

ProtocalIdToName(inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_PROTOCOL].value.uint16, szProtocalName);

// 设置默认规则 允许连接

classifyOut->actionType = FWP\_ACTION\_PERMIT;

// 禁止指定进程网络连接

if (NULL != wcsstr((PWCHAR)szProcessPath, L"iexplore.exe"))

{

// 设置拒绝规则 拒绝连接

classifyOut->actionType = FWP\_ACTION\_BLOCK;

classifyOut->rights = classifyOut->rights & (~FWPS\_RIGHT\_ACTION\_WRITE); classifyOut->flags = classifyOut->flags | FWPS\_CLASSIFY\_OUT\_FLAG\_ABSORB;

DbgPrint("[LyShark.com] 拦截IE网络链接请求... \n");

}

// 输出对端地址字符串 并阻断链接

char szRemoteAddress[256] = { 0 }; char szRemotePort[128] = { 0 };

char szLocalAddress[256] = { 0 }; char szLocalPort[128] = { 0 };

sprintf(szRemoteAddress, "%u.%u.%u.%u", (ulRemoteIp >> 24) & 0xFF, (ulRemoteIp >> 16) & 0xFF, (ulRemoteIp >> 8) & 0xFF, (ulRemoteIp)& 0xFF);

sprintf(szRemotePort, "%d", uRemotePort);

sprintf(szLocalAddress, "%u.%u.%u.%u", (ulLocalIp >> 24) & 0xFF, (ulLocalIp

>> 16) & 0xFF, (ulLocalIp >> 8) & 0xFF, (ulLocalIp)& 0xFF); sprintf(szLocalPort, "%d", uLocalPort);

// DbgPrint("本端: %s : %s --> 对端: %s : %s \n", szLocalAddress, szLocalPort, szRemoteAddress, szRemotePort);

// 如果对端地址是 8.141.58.64 且对端端口是 443 则拒绝连接

if (strcmp(szRemoteAddress, "8.141.58.64") == 0 && strcmp(szRemotePort, "443") == 0)

{

DbgPrint("[LyShark.com] 拦截网站访问请求 --> %s : %s \n", szRemoteAddress, szRemotePort);

// 设置拒绝规则 拒绝连接

classifyOut->actionType = FWP\_ACTION\_BLOCK;

classifyOut->rights = classifyOut->rights & (~FWPS\_RIGHT\_ACTION\_WRITE); classifyOut->flags = classifyOut->flags | FWPS\_CLASSIFY\_OUT\_FLAG\_ABSORB;



}

else if (strcmp(szRemotePort, "0") == 0)

{

DbgPrint("[LyShark.com] 拦截Ping访问请求 --> %s \n", szRemoteAddress);

// 设置拒绝规则 拒绝连接

classifyOut->actionType = FWP\_ACTION\_BLOCK;

classifyOut->rights = classifyOut->rights & (~FWPS\_RIGHT\_ACTION\_WRITE); classifyOut->flags = classifyOut->flags | FWPS\_CLASSIFY\_OUT\_FLAG\_ABSORB;

}

// 显示

DbgPrint("[LyShark.com] 方向: %d -> 协议类型: %s -> 本端地址: %u.%u.%u.%u:%d ->

对端地址: %u.%u.%u.%u:%d -> IRQL: %d -> 进程ID: %I64d -> 路径: %S \n", wDirection,

szProtocalName, (ulLocalIp >> 24) & 0xFF, (ulLocalIp >> 16) & 0xFF, (ulLocalIp >> 8) & 0xFF, (ulLocalIp)& 0xFF, uLocalPort,

(ulRemoteIp >> 24) & 0xFF, (ulRemoteIp >> 16) & 0xFF, (ulRemoteIp >> 8) & 0xFF, (ulRemoteIp)& 0xFF, uRemotePort,

kCurrentIrql, processId, (PWCHAR)szProcessPath);

}

// Callout函数 notifyFn 事后回调函数

NTSTATUS NTAPI notifyFn(\_In\_ FWPS\_CALLOUT\_NOTIFY\_TYPE notifyType, \_In\_ const GUID filterKey, \_Inout\_ FWPS\_FILTER2 filter)

{

NTSTATUS status = STATUS\_SUCCESS;

return status;

}

// Callout函数 flowDeleteFn 事后回调函数

VOID NTAPI flowDeleteFn(\_In\_ UINT16 layerId, \_In\_ UINT32 calloutId, \_In\_ UINT64 flowContext)

{

return;

}

// 默认派遣函数

NTSTATUS DriverDefaultHandle(PDEVICE\_OBJECT pDevObj, PIRP pIrp)

{

NTSTATUS status = STATUS\_SUCCESS;

pIrp->IoStatus.Status = status; pIrp->IoStatus.Information = 0;

IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);

return status;

}

// 创建设备

NTSTATUS CreateDevice(PDRIVER\_OBJECT pDriverObject)

{

NTSTATUS status = STATUS\_SUCCESS; PDEVICE\_OBJECT pDevObj = NULL;

UNICODE\_STRING ustrDevName, ustrSymName;

RtlInitUnicodeString(&ustrDevName, DEV\_NAME); RtlInitUnicodeString(&ustrSymName, SYM\_NAME);

status = IoCreateDevice(pDriverObject, 0, &ustrDevName, FILE\_DEVICE\_NETWORK, 0, FALSE, &pDevObj);

if (!NT\_SUCCESS(status))

{

return status;

}

status = IoCreateSymbolicLink(&ustrSymName, &ustrDevName); if (!NT\_SUCCESS(status))

{

return status;

}

return status;

}

// 卸载驱动

VOID UnDriver(PDRIVER\_OBJECT driver)

{

// 删除回调函数和过滤器,关闭引擎

WfpUnload();

UNICODE\_STRING ustrSymName; RtlInitUnicodeString(&ustrSymName, SYM\_NAME); IoDeleteSymbolicLink(&ustrSymName);

if (driver->DeviceObject)

{

IoDeleteDevice(driver->DeviceObject);

}

}

// 驱动入口

NTSTATUS DriverEntry(IN PDRIVER\_OBJECT Driver, PUNICODE\_STRING RegistryPath)

{

NTSTATUS status = STATUS\_SUCCESS;

Driver->DriverUnload = UnDriver;

for (ULONG i = 0; i < IRP\_MJ\_MAXIMUM\_FUNCTION; i++)

{

Driver->MajorFunction[i] = DriverDefaultHandle;

}

// 创建设备

CreateDevice(Driver);

// 启动WFP

WfpLoad(Driver->DeviceObject);

Driver->DriverUnload = UnDriver; return STATUS\_SUCCESS;

}

# 上方代码是一个最基本的WFP过滤框架头部函数，声明部分来源于微软的定义此处不做解释，需要注意

GUID\_ALE\_AUTH\_CONNECT\_CALLOUT\_V4 代表的是一个随机 UUID 值，该值可以任意定义只要不一致即可，驱动程序运行后会率先执行 WfpLoad() 这个函数，该函数内部通过 RegisterCalloutForLayer() 注册了一个过滤点，此处我们必须要注意三个回调函数，classifyFn, notifyFn, flowDeleteFn 他们分别的功能时，事前回调，事后回调，事后回调，而WFP框架中我们最需要注意的也就是对这三个函数进行重 定义，也就是需要重写函数来实现我们特定的功能。



NTSTATUS RegisterCalloutForLayer (

IN const GUID layerKey, IN const GUID calloutKey,

IN FWPS\_CALLOUT\_CLASSIFY\_FN classifyFn, IN FWPS\_CALLOUT\_NOTIFY\_FN notifyFn,

IN FWPS\_CALLOUT\_FLOW\_DELETE\_NOTIFY\_FN flowDeleteNotifyFn, OUT UINT32 calloutId,

OUT UINT64 filterId

}

# 既然是防火墙那么必然 classifyFn 事前更重要一些，如果需要监控网络流量则需要在事前函数中做处理，而如果是监视则可以在事后做处理，既然要在事前进行处理，那么我们就来看看事前是如何处理的 流量。

// Callout函数 classifyFn 事前回调函数



VOID NTAPI classifyFn(\_In\_ const FWPS\_INCOMING\_VALUES0 inFixedValues, \_In\_ const FWPS\_INCOMING\_METADATA\_VALUES0 inMetaValues, \_Inout\_opt\_ void layerData,

\_In\_opt\_ const void classifyContext, \_In\_ const FWPS\_FILTER2 filter, \_In\_ UINT64 flowContext, \_Inout\_ FWPS\_CLASSIFY\_OUT0 classifyOut)

{

// 数据包的方向,取值 FWP\_DIRECTION\_INBOUND = 1 或 FWP\_DIRECTION\_OUTBOUND = 0

WORD wDirection = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_FLOW\_ESTABLISHED\_V4\_DIRECTION].value.int8;

// 定义本机地址与本机端口

ULONG ulLocalIp = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_LOCAL\_ADDRESS].value.uint32; UINT16 uLocalPort = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_LOCAL\_PORT].value.uint16;

// 定义对端地址与对端端口

ULONG ulRemoteIp = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_REMOTE\_ADDRESS].value.uint32; UINT16 uRemotePort = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_REMOTE\_PORT].value.uint16;

// 获取当前进程IRQ

KIRQL kCurrentIrql = KeGetCurrentIrql();

// 获取进程ID

ULONG64 processId = inMetaValues->processId; UCHAR szProcessPath[256] = { 0 };

CHAR szProtocalName[256] = { 0 }; RtlZeroMemory(szProcessPath, 256);

// 获取进程路径

for (ULONG i = 0; i < inMetaValues->processPath->size; i++)

{

// 里面是宽字符存储的

szProcessPath[i] = inMetaValues->processPath->data[i];

}

// 获取当前协议类型

ProtocalIdToName(inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_PROTOCOL].value.uint16, szProtocalName);

// 设置默认规则 允许连接

classifyOut->actionType = FWP\_ACTION\_PERMIT;

// 禁止指定进程网络连接

if (NULL != wcsstr((PWCHAR)szProcessPath, L"qq.exe"))

{

// 设置拒绝规则 拒绝连接

classifyOut->actionType = FWP\_ACTION\_BLOCK;

classifyOut->rights = classifyOut->rights & (~FWPS\_RIGHT\_ACTION\_WRITE); classifyOut->flags = classifyOut->flags | FWPS\_CLASSIFY\_OUT\_FLAG\_ABSORB;

}

// 输出对端地址字符串 并阻断链接

char szRemoteAddress[256] = { 0 }; char szRemotePort[128] = { 0 };

char szLocalAddress[256] = { 0 }; char szLocalPort[128] = { 0 };

sprintf(szRemoteAddress, "%u.%u.%u.%u", (ulRemoteIp >> 24) & 0xFF, (ulRemoteIp >> 16) & 0xFF, (ulRemoteIp >> 8) & 0xFF, (ulRemoteIp)& 0xFF);

sprintf(szRemotePort, "%d", uRemotePort);

sprintf(szLocalAddress, "%u.%u.%u.%u", (ulLocalIp >> 24) & 0xFF, (ulLocalIp

>> 16) & 0xFF, (ulLocalIp >> 8) & 0xFF, (ulLocalIp)& 0xFF); sprintf(szLocalPort, "%d", uLocalPort);

// DbgPrint("本端: %s : %s --> 对端: %s : %s \n", szLocalAddress, szLocalPort, szRemoteAddress, szRemotePort);

// 如果对端地址是 8.141.58.64 且对端端口是 443 则拒绝连接

if (strcmp(szRemoteAddress, "8.141.58.64") == 0 && strcmp(szRemotePort, "443") == 0)

{

DbgPrint("拦截网站访问请求 --> %s : %s \n", szRemoteAddress, szRemotePort);

// 设置拒绝规则 拒绝连接

classifyOut->actionType = FWP\_ACTION\_BLOCK;

classifyOut->rights = classifyOut->rights & (~FWPS\_RIGHT\_ACTION\_WRITE);



classifyOut->flags = classifyOut->flags | FWPS\_CLASSIFY\_OUT\_FLAG\_ABSORB;

}

else if (strcmp(szRemotePort, "0") == 0)

{

DbgPrint("拦截Ping访问请求 --> %s \n", szRemoteAddress);

// 设置拒绝规则 拒绝连接

classifyOut->actionType = FWP\_ACTION\_BLOCK;

classifyOut->rights = classifyOut->rights & (~FWPS\_RIGHT\_ACTION\_WRITE); classifyOut->flags = classifyOut->flags | FWPS\_CLASSIFY\_OUT\_FLAG\_ABSORB;

}

/

// 显示

DbgPrint("方向: %d -> 协议类型: %s -> 本端地址: %u.%u.%u.%u:%d -> 对端地址:

%u.%u.%u.%u:%d -> IRQL: %d -> 进程ID: %I64d -> 路径: %S \n",

wDirection, szProtocalName, (ulLocalIp >> 24) & 0xFF, (ulLocalIp >> 16) & 0xFF, (ulLocalIp >> 8) & 0xFF, (ulLocalIp)& 0xFF, uLocalPort,

(ulRemoteIp >> 24) & 0xFF, (ulRemoteIp >> 16) & 0xFF, (ulRemoteIp >> 8) & 0xFF, (ulRemoteIp)& 0xFF, uRemotePort,

kCurrentIrql, processId, (PWCHAR)szProcessPath);

/

}

# 当有新的网络数据包路由到事前函数时，程序中会通过如下案例直接得到我们所需要的数据包头，

ProtocalIdToName 函数则是一个将特定类型数字转为字符串的转换函数。

// 数据包的方向,取值 FWP\_DIRECTION\_INBOUND = 1 或 FWP\_DIRECTION\_OUTBOUND = 0

WORD wDirection = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_FLOW\_ESTABLISHED\_V4\_DIRECTION].value.int8;

// 定义本机地址与本机端口

ULONG ulLocalIp = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_LOCAL\_ADDRESS].value.uint32; UINT16 uLocalPort = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_LOCAL\_PORT].value.uint16;

// 定义对端地址与对端端口

ULONG ulRemoteIp = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_REMOTE\_ADDRESS].value.uint32; UINT16 uRemotePort = inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_REMOTE\_PORT].value.uint16;

// 获取当前进程IRQ

KIRQL kCurrentIrql = KeGetCurrentIrql();

// 获取进程ID

ULONG64 processId = inMetaValues->processId; UCHAR szProcessPath[256] = { 0 };

CHAR szProtocalName[256] = { 0 }; RtlZeroMemory(szProcessPath, 256);

// 获取进程路径

for (ULONG i = 0; i < inMetaValues->processPath->size; i++)

{

// 里面是宽字符存储的

szProcessPath[i] = inMetaValues->processPath->data[i];

}

// 获取当前协议类型

ProtocalIdToName(inFixedValues-

>incomingValue[FWPS\_FIELD\_ALE\_AUTH\_CONNECT\_V4\_IP\_PROTOCOL].value.uint16, szProtocalName);

**拦截浏览器上网：** 防火墙的默认规则我们将其改为放行所有 classifyOut->actionType = FWP\_ACTION\_PERMIT; ，当我们需要拦截特定进程上网时则只需要判断调用原，如果时特定进程则直接设置拒绝网络访问。

// 设置默认规则 允许连接

classifyOut->actionType = FWP\_ACTION\_PERMIT;

// 禁止指定进程网络连接

if (NULL != wcsstr((PWCHAR)szProcessPath, L"iexplore.exe"))

{

// 设置拒绝规则 拒绝连接

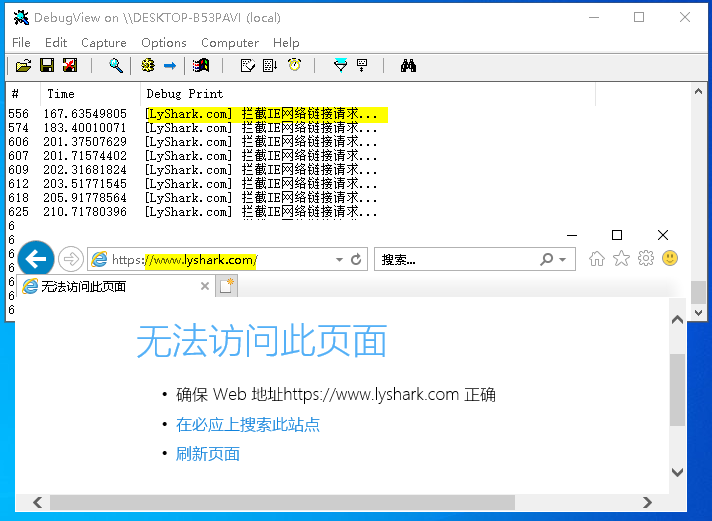
classifyOut->actionType = FWP\_ACTION\_BLOCK;

classifyOut->rights = classifyOut->rights & (~FWPS\_RIGHT\_ACTION\_WRITE); classifyOut->flags = classifyOut->flags | FWPS\_CLASSIFY\_OUT\_FLAG\_ABSORB;

DbgPrint("[LyShark.com] 拦截IE网络链接请求... \n");

}

# 当这段驱动程序被加载后，则用户使用IE访问任何页面都将提示无法访问。



**拦截指定IP地址：** 防火墙的另一个重要功能就是拦截主机自身访问特定网段，此功能只需要增加过滤条件即可实现，如下当用户访问 8.141.58.64 这个IP地址是则会被拦截，如果监测到用户时Ping请求则也会被拦截。

// 如果对端地址是 8.141.58.64 且对端端口是 443 则拒绝连接

if (strcmp(szRemoteAddress, "8.141.58.64") == 0 && strcmp(szRemotePort, "443") ==

0)

{

DbgPrint("拦截网站访问请求 --> %s : %s \n", szRemoteAddress, szRemotePort);

// 设置拒绝规则 拒绝连接

classifyOut->actionType = FWP\_ACTION\_BLOCK;

classifyOut->rights = classifyOut->rights & (~FWPS\_RIGHT\_ACTION\_WRITE); classifyOut->flags = classifyOut->flags | FWPS\_CLASSIFY\_OUT\_FLAG\_ABSORB;

}

else if (strcmp(szRemotePort, "0") == 0)

{

DbgPrint("拦截Ping访问请求 --> %s \n", szRemoteAddress);

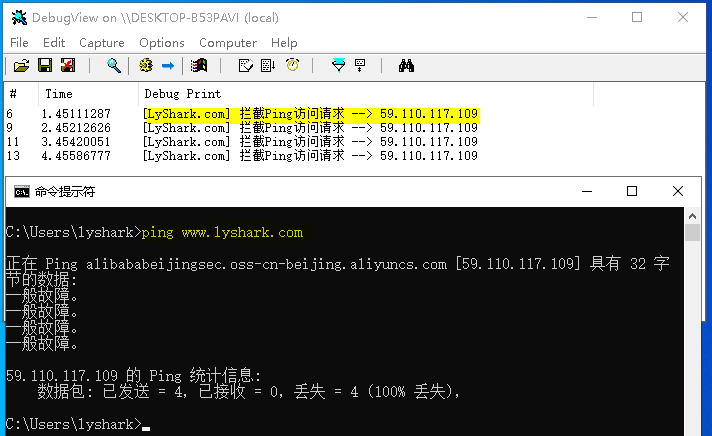
// 设置拒绝规则 拒绝连接

classifyOut->actionType = FWP\_ACTION\_BLOCK;

classifyOut->rights = classifyOut->rights & (~FWPS\_RIGHT\_ACTION\_WRITE); classifyOut->flags = classifyOut->flags | FWPS\_CLASSIFY\_OUT\_FLAG\_ABSORB;

}

当这段驱动程序被加载后，则用户主机无法访问 8.141.58.64 且无法使用ping命令。



# **抓取底层数据包：** 如果仅仅只是想要输出流经自身主机的数据包，则只需要对特定数据包进行解码即可得到原始数据。

// 输出对端地址字符串 并阻断链接

char szRemoteAddress[256] = { 0 }; char szRemotePort[128] = { 0 };

char szLocalAddress[256] = { 0 }; char szLocalPort[128] = { 0 };

sprintf(szRemoteAddress, "%u.%u.%u.%u", (ulRemoteIp >> 24) & 0xFF, (ulRemoteIp >>

16) & 0xFF, (ulRemoteIp >> 8) & 0xFF, (ulRemoteIp)& 0xFF); sprintf(szRemotePort, "%d", uRemotePort);

sprintf(szLocalAddress, "%u.%u.%u.%u", (ulLocalIp >> 24) & 0xFF, (ulLocalIp >>

16) & 0xFF, (ulLocalIp >> 8) & 0xFF, (ulLocalIp)& 0xFF); sprintf(szLocalPort, "%d", uLocalPort);

// 显示

DbgPrint("[LyShark.com] 方向: %d -> 协议类型: %s -> 本端地址: %u.%u.%u.%u:%d -> 对端地址: %u.%u.%u.%u:%d -> IRQL: %d -> 进程ID: %I64d -> 路径: %S \n",

wDirection, szProtocalName, (ulLocalIp >> 24) & 0xFF, (ulLocalIp >> 16) & 0xFF, (ulLocalIp >> 8) & 0xFF, (ulLocalIp)& 0xFF, uLocalPort,

(ulRemoteIp >> 24) & 0xFF, (ulRemoteIp >> 16) & 0xFF, (ulRemoteIp >> 8) & 0xFF, (ulRemoteIp)& 0xFF, uRemotePort,

kCurrentIrql, processId,

(PWCHAR)szProcessPath);

当这段驱动程序被加载后，则用户可看到流经本机的所有数据包。

