Table of Contents

I.	Introduction	3
II.	Compiling & Debugging Drivers	19
III.	Kernel Survival Guide	30
IV.	Memory Management	56
V.	Creating a Device Driver	65
VI.	Handling Devices	77
VII.	I/O in Device Drivers	93
VIII.	Kernel Network Architecture	130
IX.	NDIS Device Driver Basics	145
X.	NDIS Miniport Drivers	153
XI.	NDIS Protocol Drivers	164
XII.	Vista/2008 – Windows Filtering Platform	177
XIII.	Vista/2008 – Winsock Kernel Drivers	192
XIV.	Kernel Hacking	213

Appendices

I. x86 architecture and Debugging Cheat Sheet

Recommended Reading

Jonathan Levin specializes in training and consulting services. This, and many other training materials, are created and constantly updated to reflect the ever changing environment of the IT industry.

To report errata, provide feedback, or for more details, feel free to email JL@HisOwn.com

© Copyright 版權聲明

This material is protected under copyright laws. Unauthorized reproduction, alteration, use in part or in whole is prohibited, without express permission from the author.
I put a LOT of effort into my work (and I hope it shows). Respect that.

Printed on 100% recycled paper. I hope you like the course and keep the handout.. Else – Recycle!

Windows Filtering Platform

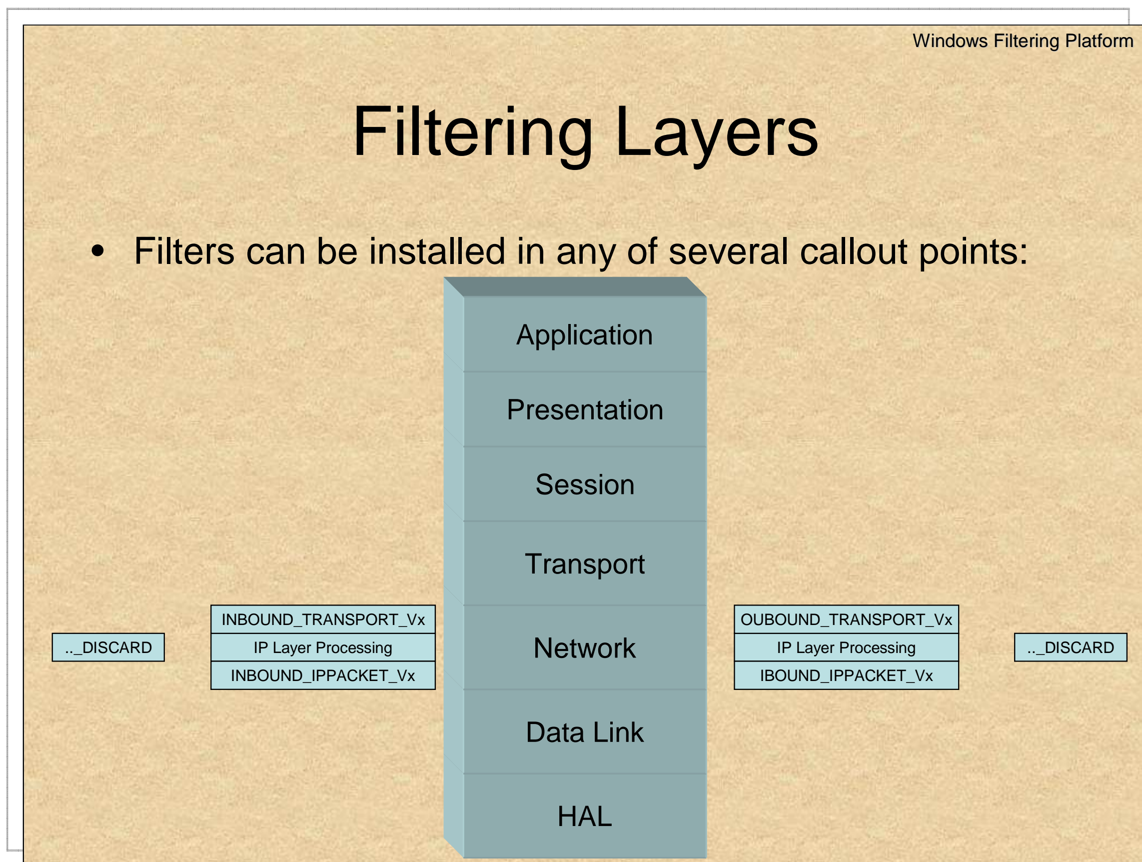
Windows Filtering Platform

Windows Filtering Platform

- New API in Vista and Server 2008
  - Deprecate TDI Filter Drivers
  - Preferred over NDIS filters
- Quick & efficient method for packet inspection:
  - Filtering platform already integrated in TCP/IP Stack
  - Drivers operate by setting callouts (=hooks) platform calls
- Mostly Kernel mode, but allow User Mode interaction
  - Filter configuration/installation/monitoring in User Mode
  - Actual filter processing in Kernel Mode

Windows Filtering Platform (WFP) is the latest addition to Windows Vista and Server 2008. This is Microsoft's answer to the very popular (and potent) "NetFilter/IPTables" architecture of Linux.

Microsoft reference page for WFP:
http://www.microsoft.com/whdc/device/network/WFP.mspx



Much like NetFilter Hooks, WFP Filters can be installed in any of several predefined points on the packet receive (incoming) and send (outgoing) paths.

Filter Layer (FWPM_LAYER_)	Purpose
INBOUND_IPPACKET_V4 INBOUND_IPPACKET_V6	Earliest point on receive path filter may be installed. Prior to IP/IPSec processing. Packet may be fragmented.
INBOUND_TRANSPORT_V4 INBOUND_TRANSPORT_V6	Post Layer III (IP) but prior to Layer IV (TCP or UDP) processing
INBOUND_IPPACKET_V4_DISCARD INBOUND_IPPACKET_V4_DISCARD	Post Layer III (IP) for packets that will never make it to layer IV

For the outbound, similar filter layers apply:

Filter Layer (FWPS_LAYER_)	Purpose
OUTBOUND_IPPACKET_V4 OUTBOUND_IPPACKET_V6	Latest point on send path filter may be installed. Prior to fragmentation but post all other processing.
OUTBOUND_TRANSPORT_V4 OUTBOUND_TRANSPORT_V6	Post Layer IV (TCP or UDP) but prior to Layer III (IP) processing
OUTBOUND_IPPACKET_V4_DISCARD OUTBOUND_IPPACKET_V6_DISCARD	Post Layer IV (TCP or UDP) processing, reseved for packets that have been discarded by Layer III

(C) 2009 JL@HisOwn.com

Creating a Custom Kernel Filter

- WFP filters are a special case of Kernel Drivers

```
NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject,
                     PUNICODE_STRING pRegistryPath);
```

- Driver is required to:
  - Populate pDriverObject, particularly specify an unload function
  - Create a device
  - Register callouts with WFP Filter Engine

Interfacing with the Engine

- WFP can be accessed from User or Kernel mode

```
NTSTATUS NTAPI
  FwpmEngineOpen0(
    IN const wchar_t  *serverName OPTIONAL,
    IN UINT32  authnService,
    IN SEC_WINNT_AUTH_IDENTITY_W  *authIdentity OPTIONAL,
    IN const FWPM_SESSION0  *session OPTIONAL,
    OUT HANDLE  *engineHandle
    );
```

```
NTSTATUS NTAPI FwpmEngineClose0(IN HANDLE engineHandle);
```

To access the Windows Filtering Platform Engine, call **FwpmEngineOpen0**. This function is accessible from both User space (via *Fwpmu.h* and *Fwpuclnt.lib*) and Kernel space (*Fwpmk.h*). The function is similarly defined in User space:

```
DWORD FwpmEngineOpen0(IN CONST wchar_t *serverName,
                      IN UINT32 authnService,
             OPTIONAL IN SEC_WINNT_AUTH_IDENTITY_W *authIdentity,
             OPTIONAL IN CONST FWPM_SESSION0 *session,
                      OUT HANDLE *engineHandle );
```

From User Mode, ServerName may be the name of a remote host – but Drivers can only work on the localhost, and must therefore leave this parameter as NULL.

The 2nd and 3rd parameters are for authentication - Drivers use RPC_C_AUTHN_WINNT or RPC_C_AUTHN_DEFAULT for the “Auth” service, and can leave authIdentity as NULL.

The 4th parameter is a pointer to an **FWPM_SESSION0** strucuture, defined as follows:

```
typedef struct FWPM_SESSION0_ {
  GUID                 sessionKey;
  FWPM_DISPLAY_DATA0   displayData;
  UINT32               flags;
  UINT32               txnWaitTimeoutInMSec;
  DWORD                processId;
  SID                 *sid;
  wchar_t             *username;
  BOOL                 kernelMode;
} FWPM_SESSION0;
```

Most fields in this structure are either unused, or automatically assigned – so it's common practice to memset to \0, then assign only the "flags" field – in which only one flag is defined: FWPM_SESSION_FLAG_DYNAMIC – which states that any filters added in the session context (= with the open Engine Handle) will be removed when the session is closed (= Handle is closed).

The following sample illustrates how to open a handle to the Engine:

```
HANDLE g_EngineHandle = NULL;
FWPM_SESSION0 session = {0};
NTSTATUS status;

session.flags = FWPM_SESSION_FLAG_DYNAMIC;

status = FwpmEngineOpen0(NULL,
                  RPC_C_AUTHN_WINNT,
                  NULL,
                  &session,
                  &g_EngineHandle);

..
/* Start Transactions, register callouts..
*/
..

FwpmEngineClose0(g_EngineHandle);
```

Listing 1: WFP Engine

Windows Filtering Platform

Transactions

- Operations with WFP are transaction-based:

```
NTSTATUS FwpmTransactionBegin0(IN HANDLE engineHandle,
                               IN ULONG  flags );
```

- Transactions are enforced when commited

```
NTSTATUS FwpmTransactionCommit0(IN HANDLE engineHandle);
```

- Transactions can also be rolled back, or aborted:

```
NTSTATUS FwpmTransactionAbort0(IN HANDLE engineHandle);
```

All operations in WFP are performed in a transactional context. Using the API is straightforward: Start a transaction with **FwpmTransactionBegin0**, then end it with **FwpmTransactionCommit0**, or roll back with **FwpmTransactionAbort0**. The only "flag" defined for **FwpmTransactionBegin0** is **FWPM_TXN_READ_ONLY** – for read only transactions.

```
HANDLE g_EngineHandle = NULL;
FWPM_SESSION0 session = {0};
NTSTATUS status;

session.flags = FWPM_SESSION_FLAG_DYNAMIC;

status = FwpmEngineOpen0(NULL,
                  RPC_C_AUTHN_WINNT,
                  NULL,
                  &session,
                  &g_EngineHandle);

status = FwpmTransactionBegin0(gEngineHandle, 0);
..
/* register callouts.. */
..

FwpmTransactionCommit0(g_EngineHandle);
FwpmEngineClose0(g_EngineHandle);
```

Listing 2: WFP Transactions

Windows Filtering Platform

Callouts

- Register Callouts using FwpsCalloutRegister0

```
NTSTATUS NTAPI FwpsCalloutRegister0(IN OUT void          *devObject,
                                    IN const FWPS_CALLOUT0 *callout,
                                    OUT OPTIONAL UINT32    *calloutId);
```

- Callout specifies three callback functions:
  - Classification function: On data event handler
  - Notification function: Miscellanaeous (non-data) event handler
  - Flow Deletion notification function: Data flow termination handler
- Add Callouts using FwpmCalloutAdd0

```
NTSTATUS NTAPI FwpmCalloutAdd0(IN  HANDLE               engineHandle,
                         CONST IN  FWPM_CALLOUT0        *callout,
                      OPTIONAL IN  PSECURITY_DESCRIPTOR sd,
                      OPTIONAL OUT UINT32               *id);
```

To register which callouts the Filter can process, it calls **FwpsCalloutRegister0**. This function, shown above, accepts three parameters:

- **The Device Object:** created by the Driver upon its DriverEntry.
- **Callout Data:** This is a struct of the following format:

```
typedef struct FWPS_CALLOUT0_ {
  GUID                                calloutKey;
  UINT32                              flags;
  FWPS_CALLOUT_CLASSIFY_FN0           classifyFn;
  FWPS_CALLOUT_NOTIFY_FN0             notifyFn;
  FWPS_CALLOUT_FLOW_DELETE_NOTIFY_FN0 flowDeleteFn;
} FWPS_CALLOUT0;
```

- **Callout Id**: If not NULL, this specifies a 32-bit ID that may be used to selectively unregister the callout, associate transport layer flows with it, or edit it by packet injection.

Each callout is uniquely identified by a 128 bit GUID. This is set by the driver, and should use the same GUID prefix the driver uses. The filter is expected to define up to three callback functions, as we will see next.

Only two flags are currently defined: **FWP_CALLOUT_FLAG_CONDITIONAL_ON_FLOW** – if the callout is to be associated with a data flow (discussed later) and **FWP_CALLOUT_FLAG ALLOW_OFFLOAD** to specify the filter can work with TCP offloading, when it can. This is particularly important, since without this flag TCP offloading will be disabled for traffic matching the filter, noticably impacting performance.

The callout is added with **fwpmCalloutAdd0**, which takes in a pointer to an FWPM_CALLOUT0 structure, defined as follows:

```
typedef struct FWPM_CALLOUT0_ {
  GUID                    calloutKey;
  FWPM_DISPLAY_DATA0      displayData;
  UINT32                  flags;
  GUID                   *providerKey;
  FWP_BYTE_BLOB           providerData;
  GUID                    applicableLayer;
  UINT32                  calloutId;
} FWPM_CALLOUT0;
```

Putting it all together, we have:

```
DEFINE_GUID (CALLOUT_KEY, …,…);

UINT32 calloutId
FWPS_CALLOUT0 fwpsCallout = {0};

FWPM_FILTER0 filter = {0};
FWPM_FILTER_CONDITION0 filterConditions[1] = {0};

FWPM_CALLOUT0 fwpmCallout = {0};

fwpsCallout.calloutKey = CALLOUT_KEY;

fwpsCallout.classifyFn = myClassificationFunction;
fwpsCallout.notifyFn   = myNotificationFunction;

status = FwpsCalloutRegister0(pdo,
                              &sCallout,
                              &calloutId);

/* Callout is registered - now add */


fwpmCallout.calloutKey = *calloutKey;
fwpmCallout.displayData.name = L"Callout Name";
fwpmcallout.displayData.description = L"Callout Desc";

/* Choose filtering layer identifier .. */
fwpmCallout.applicableLayer = &FWPM_LAYER_STREAM_V4;

status = FwpmCalloutAdd0(
               gEngineHandle,
               &mCallout,
               NULL,
               NULL
               );
```

Listing 3: Registering and adding a callout

Callouts

- Classification function implements:

```
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);
```

- Data to caller specified in classifyOut structure
  - actionType specifies processing rule

Action	Description
FWP_ACTION_BLOCK	Data blocked terminating send or recv path
FWP_ACTION_PERMIT	Data allowed
FWP_ACTION_CONTINUE	Decision deferred to next filter in chain

The classification function is the heart of the filter. It implements the logic that either permits or blocks the transaction.

To inspect the actual data, one would first need to obtain it from the layer data by typecasting it to an FWPS_STREAM_CALLOUT_IO_PACKET0:

```
typedef struct FWPS_STREAM_CALLOUT_IO_PACKET0_ {
  IN FWPS_STREAM_DATA0          *dataStream;
  IN SIZE_T                      missedBytes;
  OUT UINT32                     countBytesRequired;
  OUT SIZE_T                     countBytesEnforced;
  OUT FWPS_STREAM_ACTION_TYPE    streamAction;
} FWPS_STREAM_CALLOUT_IO_PACKET0;
```

And the data is then obtainable by the FWPS_STREAM_DATA pointer:

```
typedef struct FWPS_STREAM_DATA0_ {
  UINT32                      flags;
  FWPS_STREAM_DATA_OFFSET0    dataOffset;
  SIZE_T                      dataLength;
  NET_BUFFER_LIST            *netBufferListChain;
} FWPS_STREAM_DATA0;
```

Seeing as WFP is only supported on Vista and later OSes, it uses the NDIS 6 NET_BUFFER structures in netBufferListChain. The Flags are a bitwise or (|) of the following:

Flag	Purpose
FWPS_STREAM_FLAG_RECEIVE	This is an inbound stream
FWPS_STREAM_FLAG_SEND	This is an outbound stream
FWPS_STREAM_FLAG_{XXX}_EXPEDITED	URG flag was set in packet’s TCP header
FWPS_STREAM_FLAG_{XXX}_DISCONNECT	FIN flag was set in packet’s TCP header
FWPS_STREAM_FLAG_{XXX}_ABORT	RST flag was set in packet’s TCP header
FWPS_STREAM_FLAG_SEND_NODELAY	Socket option of NoDelay was set

(C) 2009 JL@HisOwn.com

Windows Filtering Platform

Filters

- Register Filters using FwpmFilterAdd0

```
NTSTATUS NTAPI FwpmFilterAdd0(IN   HANDLE                  engineHandle,
                        CONST IN   FWPM_FILTER0           *filter,
                     OPTIONAL IN   PSECURITY_DESCRIPTOR    sd,
                     OPTIONAL OUT UINT64 *id);
```

- To associate a Filter with a callout:
  - Set action.calloutKey field to callout GUID
  - Set action.actionType to TERMINATING or INSPECT
- Amazing variety of filter keys to choose from:
  - http://msdn.microsoft.com/en-us/library/aa504850.aspx

The last step in creating a callout is to associate with a filter. For this, FwpmFilterAdd0 must be called with a pointer to an FWPM_FILTER0 structure:

```
typedef struct FWPM_FILTER0_ {
GUID                     filterKey;
FWPM_DISPLAY_DATA0       displayData;
UINT32                   flags;
GUID                    *providerKey;
FWP_BYTE_BLOB            providerData;
GUID                     layerKey;
GUID                     subLayerKey;
FWP_VALUE0               weight;
UINT32                   numFilterConditions;
FWPM_FILTER_CONDITION0  *filterCondition;
FWPM_ACTION0             action;
union {
                  UINT64 rawContext;
                    GUID providerContextKey;
      } ;
GUID                     *reserved;
UINT64                   filterId;
FWP_VALUE0               effectiveWeight;
} FWPM_FILTER0;
```

Filters have many complex features, like weight, which are not discussed in our scope. The two most important fields for our discussion, however, are filterCondition and action.

The filterCondition points to an array of numFilterConditions FWPM_FILTER_CONDITION0:

```
typedef struct FWPM_FILTER_CONDITION0_ {
GUID fieldKey;
FWP_MATCH_TYPE matchType;
FWP_CONDITION_VALUE conditionValue; }
FWPM_FILTER_CONDITION0;
```

With FWPM_MATCH_TYPE being the operators, and FWPM_CONDITION_VALUE0 being a union identified by a type field. To filter against UDP, we'd do something like this:

```
FWPM_FILTER_CONDITION0 fwpcConditions[2];

// UDP protocol filter condition
fwpcConditions[0].fieldKey             = FWPM_CONDITION_IP_PROTOCOL;
fwpcConditions[0].matchType            = FWP_MATCH_EQUAL;
fwpcConditions[0].conditionValue.type  = FWP_UINT8;
fwpcConditions[0].conditionValue.uint8 = IPPROTO_UDP; /* 0x11 */

// DNS protocol filter condition - destined to port 53
fwpcConditions[1].fieldKey             = FWPM_CONDITION_IP_REMOTE_PORT;
fwpcConditions[1].matchType            = FWP_MATCH_EQUAL;
fwpcConditions[1].conditionValue.type  = FWP_UINT8;
fwpcConditions[1].conditionValue.uint8 = 0x035; /* That's 53 ☺ */

fwpFilter.numFilterConditions = 2;
fwpFilter.filterConditions = fwpcConditions;

/* Defer action of BLOCK or PERMIT to callout */
fwpfilter.action.type = FWP_ACTION_CALLOUT_TERMINATING;
fwpfilter.action.calloutKey = *calloutKey;


 status = FwpmFilterAdd0(
               gEngineHandle,
               &filter,
               NULL,
               NULL);
```

Listing 4: Adding a filter

…If you liked this course, consider…

Networking:

Networking Protocols – OSI Layers 2-4:
Focusing on - Ethernet, Wi-Fi, IPv4, IPv6, TCP, UDP and SCTP

Application Protocols – OSI Layers 5-7:
Including - DNS, FTP, SMTP, IMAP/POP3, HTTP and SSL

VoIP:
In depth discussion of H.323, SCCP, SIP and RTP/RTCP, down to the packet level.

Windows Networking Internals:

NetBIOS/SMB, CIFS, DCE/RPC, Kerberos, NTLM, and networking architecture

---

Linux:

Linux Survival and Basic Skills:

Graceful introduction into the wonderful world of Linux for the non-command line oriented user. Basic skills and commands, work in shells, redirection, pipes, filters and scripting

Linux Administration:

Follow up to the Basic course, focusing on advanced subjects such as user administration, software management, network service control, performance monitoring and tuning.

Linux User Mode Programming:

Programming POSIX and UNIX APIs in Linux, including processes, threads, IPC mechanisms and networking. Linux User experience required.

Linux Kernel Programming:

Guided tour of the Linux Kernel, 2.4 and 2.6, focusing on design, architecture, writing device drivers (character, block), performance and network devices

Embedded Linux Kernel Programming:

Similar to the Linux Kernel programming course, but with a strong emphasis on development on non-intel and/or tightly constrained embedded platforms

---

Windows:

Windows Programming:

Windows Application Development, focusing on Processes, Threads, DLLs, Memory Management, and Winsock

Windows Kernel Programming (this course):

Windows Kernel Architecture and Device Driver development – focusing on Network Device Drivers (in particular, NDIS) and the Windows Driver Model. Updated to include NDIS 6 and Winsock Kernel

---

Security:

Cryptography:

From Basics to implementations in 5 days: foundations, Symmetric Algorithms, Asymmetric Algorithms, Hashes, and protocols. Design, Logic and implementation

Application Security

Writing secure code – Dealing with Buffer Overflows, Code, SQL and command Injection, and other bugs… before they become vulnerabilities that hackers can exploit.

(C) 2009 JL@HisOwn.com