I file system filter driver

L'articolo introduce l'architettura dei file system filter e dei minifilter driver per Windows XP e Vista, discutendo i principali aspetti implementativi.

Introduzione

di Antonino Calderone

Esistono numerose applicazioni di utilizzo più o meno comune che fanno un uso avanzato del file system, in particolare i software usati per la salvaguardia della sicurezza e dell'integrità dei dati. Questi, infatti ne estendono le funzionalità per monitorare gli accessi al disco (gli antivirus), per effettuare backup automatici (i tool di recovery), per cifrare file (i driver di crittografia dei dati), oppure, meno nobilmente, per occultare porzioni di file system (i rootkit).

E tutte queste applicazioni utilizzano degli speciali device driver denominati *file system filter*.

Prima di sviscerarne i misteri ci preme dispensare un consiglio: chi sospetti che sul proprio computer possa essere stato installato un rootkit ([1]) a propria insaputa, farebbe meglio a non farsi prendere dal panico, ma a continuare nella lettura di quest'articolo. Quello che diremo potrebbe essergli utile per l'eventuale rimozione dell'ospite non desiderato.

Chi invece non ha motivo di sospettare nulla di simile, farebbe meglio a leggere quest'articolo per le stesse ragioni, ma con più attenzione.

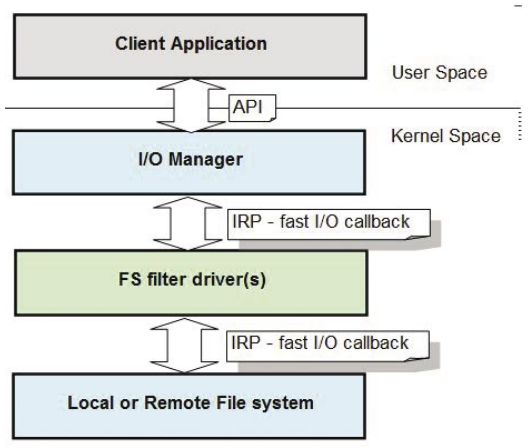
Antonino Calderone acalderone@infomedia.it

Lavora in Ericsson con la qualifica di Technical Development Engineer occupandosi principalmente di networking e le problematiche di interprocess communication per i sistemi embedded e real-time.
Progetta e sviluppa software, device driver e firmware per i sistemi per le telecomunicazioni e sistemi di controllo e automazione. Sono tra i suoi principali interessi gli aspetti architetturali e gli internals dei sistemi operativi e la programmazione nello spazio di nucleo.

Anatomia di un file system filter driver

Un file system filter è un *Kernel Mode Driver* (avevamo introdotto i KMD nell'articolo [2]), posizionato logicamente al di sopra dei driver del file system (da solo o impilato con altri filtri), e assolve la funzione di monitorarne ed eventualmente modificarne il comportamento, per supportare funzioni aggiuntive rispetto a quelle comunemente offerte dal driver nativo sotteso (come schematizzato in **Figura 1**).

FIGURA 1 Collocazione logica dei file system filter driver e percorso logico degli IRP



LISTATO 1 Implementazione della routine filter_pass_through

```
NTSTATUS filter_pass_through (
  __in PDEVICE_OBJECT DeviceObject,
  __in PIRP Irp )
{
 IoSkipCurrentIrpStackLocation( Irp );

 //  Call the file system driver with the IRP
 return IoCallDriver(
   ((MY_DEVICE_EXTENSION) DeviceObject->DeviceExtension)->
      AttachedToDeviceObject, Irp );
}
```

Un file system filter non è dunque un driver convenzionale, ma opera in simbiosi con il driver del file system nella gestione delle operazioni quali la creazione, l'apertura e la chiusura dei file, la gestione delle directory, le informazioni sul volume e quella del flusso di dati letti o scritti sui file stessi, le operazioni di caching, locking, quota disco, sicurezza e molto altro.

La sua struttura implementativa ha molti elementi del driver convenzionale: ha ovviamente un entry point, supporta le dispatch e I/O completion routine; gestisce e può generare I/O Request Packet (IRP); può gestire le richieste di IOCTL in aggiunta alle richieste di file system control (FSCTL).

Il suo compito primario è quello di agganciare le richieste verso il driver del file system, che sappiamo processa gli IRP costruiti dall'*I/O Manager dell'Executive*, in risposta alle richieste di accesso ai dati salvati in modo strutturato su disco.

Il filtro estende le funzionalità del driver sotteso collocandosi al di sopra di esso nella struttura logica conosciuta come *driver stack*. Più precisamente, l'I/O Manager consente a un driver di collegare un proprio *device object* a quello di un altro driver: un filtro può realizzare questo collegamento usando la primitiva *IoAttachDeviceToDeviceStackSafe* ([3]).

Invocando questa routine, un driver si assicura che tutte le successive richieste inviate al device object del file system driver sotteso siano prima inoltrate ad esso, che può quindi processarle, e a propria volta mandarle avanti verso il driver successivo della pila, mediante la primitiva *IoCallDriver* ([4]).

Il collegamento va effettuato in una *notification routine*, implementata nel filtro e invocata ogni volta che il file system diviene attivo (e cessare quando smette di esserlo). Questa routine può essere registrata usando la primitiva *IoRegisterFsRegistrationChange* ([5]).

Solitamente questa registrazione avviene nel contesto di esecuzione della routine *DriverEntry* del filtro, dove vengono anche registrate le callback function per la gestione degli IRP e delle *fast I/O request*, congiuntamente a tutte le altre inizializzazioni funzionali per lo stesso driver del filtro.

Ciascuna richiesta processata in una *dispatch routine* del filtro è pre-processata o post-processata (o entrambe le cose) oppure passata in modo trasparente al driver sottostante.

Nel **Listato 1** abbiamo riportato come esempio una routine denominata *filter_pass_through* che un filtro potrebbe associare agli IRP che non debbano essere processati. Nell'esempio si assume che il campo della struttura *DeviceExtension* denominato *AttachedToDeviceObject*, punti al device object al quale il filtro è collegato.

Quando è necessario post-processare una richie-

LISTATO 2 Esempio di sincronizzazione tramite I/O completion routine

```
KEVENT waitEvent;

KeInitializeEvent( &waitEvent, NotificationEvent, FALSE );

IoCopyCurrentIrpStackLocationToNext( Irp );

IoSetCompletionRoutine( Irp,
                        my_completion_routine,
                        &waitEvent,     //context parameter
                        TRUE, TRUE, TRUE );

status = IoCallDriver(
  ((MY_DEVICE_EXTENSION) DeviceObject->DeviceExtension)->
     AttachedToDeviceObject, Irp );

if (STATUS_PENDING == status) {
  status = KeWaitForSingleObject(&waitEvent, Executive,
                                 KernelMode, FALSE, NULL );
}
```

LISTATO 3 Implementazione della completion routine

```
NTSTATUS
my_completion_routine (
  __in PDEVICE_OBJECT DeviceObject,
  __in PIRP Irp,
  __in PVOID Context
)
{
  KeSetEvent((PKEVENT)Context, IO_NO_INCREMENT, FALSE);

  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

sta, dopo la chiamata alla funzione *IoCallDriver*, il filtro deve effettuare una copia dei parametri dell'IRP dalla propria *I/O stack location* a quella del driver del file system, chiamando la routine *IoCopyCurrentIrpStackLocationToNext*, quindi attendere che questi completi il proprio lavoro. Tipicamente questo è ottenuto impostando una *I/O completion routine* (tramite la funzione *IoSetCompletionRoutine*) prima della chiamata alla funzione *IoCallDriver*, come illustrato nell'esempio del **Listato 2**.

La completion routine dell'esempio segnala l'evento atteso dopo la chiamata alla routine *IoCallDriver* e la sua implementazione si riduce essenzialmente a quella del **Listato 3**.

La gestione delle fast I/O routine è più semplicemente realizzata chiamando dalla callback function agganciata a quella corrispondente del driver sotteso (tipicamente quella file system driver), eventualmente preceduta o seguita dal codice che effettua rispettivamente le pre-elaborazioni o le post-elaborazioni.

Pur tratteggiato, il contorno di un file system filter è quello delineato. Aggiungiamo che con il WDK di Microsoft ([6]) sono distribuiti alcuni esempi completi che possono essere usati come punto di partenza per una propria implementazione. Tuttavia prima di cimentarsi in tale attività suggeriamo anche al più impaziente dei lettori di proseguire con la lettura dell'articolo, perché stiamo per analizzare un componente di sistema che può semplificare di molto la creazione dei filtri.

Questo componente è un file system filter creato da Microsoft, disponibile nelle recenti versioni di Windows e installato con i *service pack* delle versioni meno recenti, denominato *Filter Manager*.

Il Filter Manager e i minifilter

Scrivere un file system filter può riservare molte insidie: le performance oppure la resilienza dei dati del file system "filtrato" possono essere compromessi pericolosamente nel caso di malfunzionamenti.

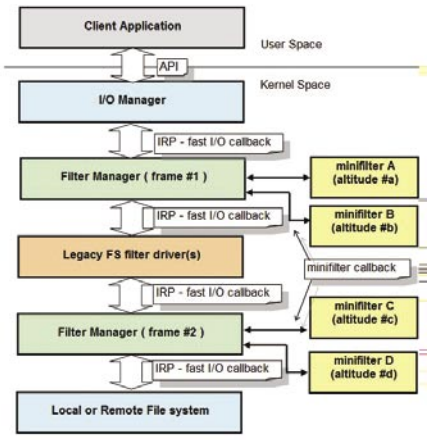
Usando il Filter Manager si riducono questi rischi: i file system filter vengono rimpiazzati da driver denominati *minifilter*, all'interno dei quali è possibile implementare le sole funzioni per le routine del driver del file system che si intenda estendere, senza doversi preoccupare degli aspetti implementativi di contorno (che spesso però nascondono problematiche che rendono difficile e critica la creazione di un filtro convenzionale).

L'introduzione dei tool per sviluppare minifilter è avvenuta con il rilascio dell'IFS Kit ([7]). Da allora, Microsoft ha classificato legacy i file system filter convenzionali, consigliando agli sviluppatori di adottare il nuovo modello. Tra i vantaggi derivanti, la disponibilità di uno specifico set di funzioni fornite come supporto di libreria pensate per semplificare la scrittura delle tipiche operazioni implementate all'interno di un filtro.

Il Filter Manager è parte integrante del sistema operativo, ma diventa attivo solo quando un minifilter viene caricato e collegato allo stack dei driver del file system.

Quando coesistono più minifilter e con il Filter Manager più filtri legacy, l'ordine di collegamento

FIGURA 2 Schematizzazione di Filter Manager e minifilter



dipende da un identificatore numerico chiamato *altitude* (che traduciamo come altitudine).

La collocazione di un minifilter a una certa altitudine è definita *instance* (istanza). Un minifilter può essere multi-istanziato a diverse altitudini.

L'attribuzione dell'altitudine assicura che ciascun filtro sia collocato nello stack dei driver in una posizione nota e prestabilita. I valori di altitudine sono suddivisi in classi e assegnati da Microsoft, che amministra e pubblica un elenco aggiornato dei software e relativi produttori che abbiano richiesto e ottenuto l'attribuzione di un'altitudine (che viene dunque riservata).

Un minifilter driver può filtrare sia le operazioni basate sugli IRP che quelle sulle fast I/O routine. Per ciascuna di esse possono essere associate delle *preoperation* e/o delle *postoperation callback routine*, invocate dal manager per ogni minifilter che le abbia registrate, in base all'ordine stabilito dalla propria altitudine. Per inter-operare con i filtri legacy, il manager può essere agganciato in locazioni differenti rispetto quest'ultimi. Ciascuna di queste istanze è denominata *frame*.

Al frame è associato un ben preciso intervallo di altitudini che assicurano a un minifilter una giusta collocazione anche rispetto i filtri legacy (come schematizzato in **Figura 2**).

Un minifilter può essere caricato come un qualunque altro driver, quindi al boot o dinamicamente. Il suo entry point è la routine *DriverEntry* all'interno della quale devono essere effettuate le operazioni di registrazione delle routine di callback. Tale registrazione avviene mediante l'uso della funzione *FltRegisterFilter*.

Quando le strutture del filtro sono inizializzate in modo consistente è possibile notificare al Filter Manager che il filtro è operativo chiamando la funzione *FltStartFiltering*.

L'installazione di un minifilter avviene definendo un file *.INF*. In questo possono essere definite le opzioni di start-up per più istanze del driver, e per ognuna di esse possono essere specificati nome, altitudine e una serie di flag di configurazione.

Il manager è responsabile della notifica sulla disponibilità di un volume quando questo viene montato, invocando la funzione di callback *InstanceSetupCallback* del minifilter. Questa chiamata viene fatta durante l'esecuzione della routine *FltStartFiltering*, ma può inoltre essere sollecitata dalla chiamata esplicita alle funzioni *FltAttachVolume* oppure *FilterAttach*.

L'istanza di un minifilter è rimossa quando il driver viene scaricato, e questo causa il detach dei collegamenti al volume. Esiste una specifica callback che un minifilter registra denominata *InstanceQueryTeardownCallback*, che dà la possibilità al minifilter di avere notifica del tentativo di smontaggio un volume. Il minifilter può così effettuare le azioni per completare le operazioni pendenti ed effettuare quelle necessarie per liberare e ripristinare lo stato delle risorse riservate. Dopo l'avvenuta notifica il manager libera le risorse allocate e rimuove l'istanza del driver.

Un filtro può essere rimosso anche su richiesta esplicita se chiamata la funzione *FltUnloadFilter* o la funzione in user-space *FilterUnload*. Durante questa fase, il filtro ottiene un'ultima notifica tramite la funzione *FilterUnloadCallback*. La registrazione di questa callback è facoltativa, tuttavia un filtro che non registri la callback non viene rimosso dal filter manager.

Avevamo visto come per un filtro legacy il processing delle IRP implicasse una corretta sincronizzazione se necessario usando delle I/O completion routine, oppure la copia dei parametri nelle stack location dei driver sottesi per la propagazione degli IRP. Con i minifilter non è necessario preoccuparsi di questi aspetti, il Filter Manager nasconde questa complessità e in più fornisce un nutrito numero di strumenti per semplificare tutti gli aspetti della

FIGURA 3 Stampe di debugging tracciate dall'utility DbgView

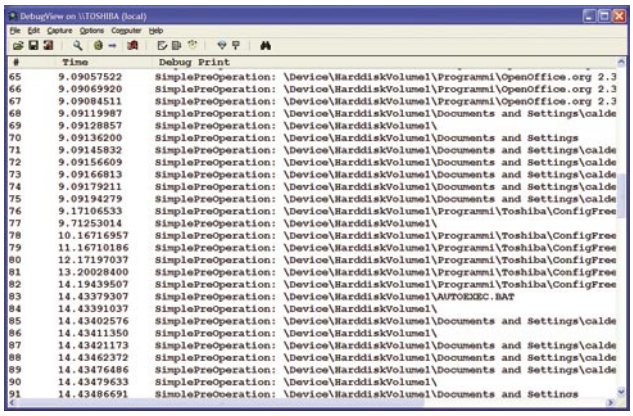
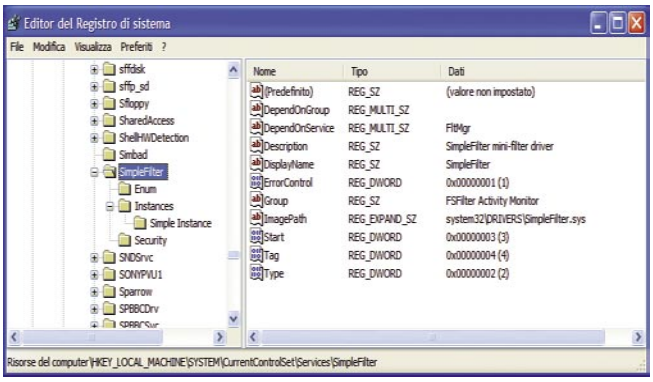


FIGURA 4 Valori del registry del minifilter SimpleFilter dopo l'installazione



gestione delle richieste, compresa la possibilità di differire le richieste; fornisce inoltre gli strumenti per la gestione di code e supporta meccanismi di locking. E ancora fornisce il supporto per la gestione dei parametri, l'accesso ai buffer, la gestione dei nomi dei file, la comunicazione tra kernel e user space (si veda per maggiori dettagli [8]). L'interazione con questi strumenti avviene mediante una libreria di funzioni (*FltXxx*) utilizzabile (solo) all'interno dei minifilter, e una libreria di funzioni che esporta in user-space i metodi per interagire con questi.

Un esempio concreto: SimpleFilter

Dopo quanto detto riteniamo utile proporre l'implementazione completa di un minifilter che abbiamo denominato *SimpleFilter*.

L'obiettivo è quello di fornire un esempio sufficientemente completo, ma al contempo semplice. Il lettore potrà dilettarsi ad aggiungere funzionalità e complessità a suo piacimento.

Il minifilter (**Listato 4**) è stato concepito per intercettare la richiesta di *IRP_MJ_CREATE* verso il file system driver e inviare una stampa di debugging per ogni file object associato a tale richiesta, usando la primitiva *DbgPrint* ([9]).

Queste stampe, tra l'altro, possono essere visualizzate mediante l'utility *DbgView*, uno dei tool che fanno parte della *Sysinternals Suite* (si possono ottenere maggiori dettagli al riguardo all'URL [10]).

L'I/O Manager trasmette una richiesta di tipo *IRP_MJ_CREATE* al driver del file system, quando un nuovo file o una nuova directory stanno per essere creati o analogamente quando un file, un dispositivo, una directory o il volume logico esistenti stanno per essere aperti.

Di norma tale IRP è inoltrato a fronte della chiamata Win32 *CreateFile* o direttamente a fronte dell'invocazione di una corrispondente API nativa del kernel.

Il file system driver riceve e processa la richiesta, restituendo l'esito di tale operazione all'I/O Manager che la propaga a propria volta verso i moduli di livello più alto.

La struttura del codice dell'esempio essenzialmente ricalca quanto discusso nel precedente paragrafo, annotiamo che il lavoro principale è svolto nella routine *SimplePreOperation*, chiamata prima che venga processata la richiesta contenuta nell'IRP. Questa controlla se la richiesta riguardi un FileObject, in caso affermativo ottiene la risoluzione del nome dello stesso tramite la funzione *FltGetFileNameInformation*. Il campo *Name* della struttura restituita da tale funzione conterrà di norma il nome completo dell'oggetto file, che viene dunque tracciato mediante la funzione *DbgPrint*.

In **Figura 3** è mostrato un esempio dell'output prodotto dal minifilter e catturato dall'utility *DbgView*. L'esempio completo può essere scaricato dal sito ftp di Infomedia e compilato mediante gli strumenti forniti con il WDK (vi rimandiamo alla documentazione completa di questo per i dettagli sulla software factory).

Per creare il package di installazione si possono copiare il file *simplefilter.sys* (il risultato della compilazione del driver) e il file *simplefilter.inf* nella medesima directory (chiamiamola di installazione).

Successivamente da Explorer si può chiedere di installare il driver scegliendo l'opzione "installa" dal menu contestuale associato alla pressione del tasto destro del mouse sul file *simplefilter.inf*. Oppure è possibile ottenere lo stesso risultato lanciando dalla directory di installazione il comando:

```
RUNDLL32.EXE SETUPAPI.DLL,InstallHinfSection
DefaultInstall 132 .\simplefilter.inf
```

(ulteriori dettagli sull'installazione dei minifilter possono essere trovati all'URL [11]). L'installazione, in ultimo, consiste nella copia del file .sys nella directory *%systemroot%\system32\drivers* e nella

creazione della chiave di registro

```
HKEY_LOCAL_MACHINE\SYSTEM\
CurrentControlSet\Services\
SimpleFilter
```

e i relativi campi (come illustrato in **Figura 4**) necessari al Service Control Manager per poter caricare il device driver. Dopo la sua installazione, il filtro può essere dunque caricato dinamicamente mediante il comando

```
net start simplefilter
```

Questo però non basta per attivarlo. Per poterlo vedere in azione è necessario anche collegarlo a un volume logico usando la primitiva user-space *FilterAttach*. E analogamente è possibile scollegarlo usando la funzione duale *FilterDetach*. Nell'esempio proposto abbiamo scritto una *console application* (**Listato 5**) denominata *simple*, che può essere usata per "collegare a" e se necessario "scollegare da" un volume logico il filtro. La sintassi associata alla riga di comando dell'applicazione *simple* è la seguente:

```
simple [unit[:]] [/d]
```

Per esempio, per agganciare le chiamate verso il volume C: è sufficiente eseguire il comando

```
simple c:
```

e analogamente per il detaching dello stesso, il comando

```
simple c: /d
```

Il nostro minifilter è realmente semplice ma anche poco utile, tuttavia può essere usato come punto di partenza per una propria implementazione, in aggiunta ad altri esempi più complessi e interessanti già distribuiti con il WDK di Microsoft.

LISTATO 5 Sorgente della console application *simple*

```
//simple (c) 2008 - Antonino Calderone - acalderone@infomedia.it

#include <stdlib.h>
#include <stdio.h>
#include <windows.h>
#include <assert.h>
#include <strsafe.h>
#include <fltUser.h>

#define SUCCESS     0
#define FILTER_NAME L"SimpleFilter"

int _cdecl main ( __in int argc, __in_ecount(argc) char *argv[])
{
  HRESULT hResult = S_OK;

  WCHAR unit_str[] = L"C:";

  if (argc<2) {
    fprintf(stderr, "usage:%s [unit[:]] [/d]\n", argv[0]);
    return 1;
  }

  ((char*)unit_str)[0] = argv[1][0];

  if (argc>2 && strcmp(argv[2], "/d")==0) {
    hResult = FilterDetach( FILTER_NAME,
                            (PWSTR)unit_str,
                            (PWSTR)L"Simple Instance" );
  }
  else {
    hResult = FilterAttach( FILTER_NAME,
                            (PWSTR)unit_str,
                            NULL, // instance name
                            0, 0 );
  }

  if ( !hResult ) {
    printf("Operation success\n");
  }
  else {
    fprintf(stderr, "Operation failed with error code 0x%08x\n",
hResult);
      return 1;
  }

  return 0;
}
```

Conclusioni

La nostra introduzione ai filtri per i file system driver si conclude con le seguenti riflessioni.

La prima: i filtri (o minifilter che siano) sono uno

strumento estremamente potente, e consentono di aggiungere alle proprie applicazioni funzionalità avanzate. Ne abbiamo citate alcune di uso più o meno comune nel paragrafo introduttivo. La ragione è (o crediamo dovrebbe essere) ovvia: il ruolo del file system è centrale per ogni sistema operativo moderno e Windows non costituisce un'eccezione.

La seconda: scrivere un filtro costa parecchio quando si comincia a fare sul serio, come per qualunque altro modulo che funzioni in kernel space. Tuttavia i nuovi strumenti forniti da Microsoft semplificano quest'attività, abbassando la probabilità di instabilità per le applicazioni che ne facciano uso. Cosa che ci fa ben sperare circa i conflitti e le idiosincrasie di molti software che turbano (e adesso ne dovrebbe essere meno oscura la ragione) il funzionamento del sistema.

L'ultima: tra gli esempi citati abbiamo menzionato i rootkit: ci preme fornire un'ultima "dritta" al riguardo. Nella già citata *Sysinternals Suite* sono presenti molti strumenti di analisi e monitoring del sistema, non ultima un'applicazione denominata *RootkitRevealer*. Inutile sottolinearne l'utilità.

CODICE ALLEGATO

ftp.infomedia.it *FSFDriver*

Sorgenti del Minifilter Driver "simplefilter" per Windows XP e Vista e della console application simple. Include sorgenti e binari.

BIBLIOGRAFIA & RIFERIMENTI

[1] http://it.wikipedia.org/wiki/Rootkit
[2] A. Calderone - "Accesso all'I/O in Windows e Linux", Computer Programming N.156 - Gennaio 2006
[3] http://msdn2.microsoft.com/en-us/library/ms795450.aspx
[4] http://msdn2.microsoft.com/en-us/library/aa490633.aspx
[5] http://msdn2.microsoft.com/en-us/library/ms795466.aspx
[6] http://www.microsoft.com/whdc/devtools/wdk/WDKpkg.mspx
[7] http://www.microsoft.com/whdc/DevTools/IFSKit/default.mspx
[8] http://msdn2.microsoft.com/en-us/library/aa488210.aspx
[9] http://msdn2.microsoft.com/en-us/library/ms792790.aspx
[10] http://technet.microsoft.com/it-it/sysinternals/default(en-us).aspx
[11] http://msdn2.microsoft.com/en-us/library/aa488214.aspx