Accesso all'I/O in Windows e Linux

Quest'articolo descrive le tecniche per accedere in user-space alle porte di I/O sui sistemi IA-32 in ambienti Linux e Windows XP/2000/NT. È descritta l'implementazione di un driver per Windows che fa uso di funzioni dell'API del kernel non ufficialmente documentate.

di Antonino Calderone

I sistemi con architettura Intel a 32 Bit (IA-32), oltre a supportare dispositivi memory-mapped, consentono uno specifico indirizzamento attraverso quelle che sono denominate porte di I/O. Le istruzioni del microprocessore usate per accedere alle porte di I/O sono considerate privilegiate e sottoposte per questo al controllo del sistema. Sia Linux che Windows (XP, 2000 o NT) adottano questo meccanismo di protezione.

Solo due dei quattro livelli di privilegio, detti anche anelli (ring) dei processori con architettura IA-32 (x86), sono utilizzati nei due sistemi operativi: ring 0 per il "kernel space" e ring 3 per lo "user space". In kernel space è possibile utilizzare qualunque istruzione privilegiata e accedere a qualunque porzione di I/O senza restrizioni. In user space normalmente questo è possibile se il sistema operativo lo permette.

Anche se le applicazioni utente normalmente non hanno interesse ad accedere direttamente alle periferiche, esiste una particolare classe di applicazioni che viola questa norma. Questo tipo di applicazione rientra nella famiglia degli User Mode Driver (UMD). Gli UMD che accedono direttamente all'I/O (come, per esempio XFree86) sono normali applicazioni utente, almeno dal punto di vista della software factory, delle librerie di sistema e delle tecniche di debugging. Quello che per un UMD rappresenta un punto di forza, in genere rappresenta una mancanza per un Kernel Mode Driver. Scrivere un KMD per Windows o Linux richiede strumenti e conoscenze ben diverse da quelle usate per le comuni applicazioni utente. Un UMD può rappresentare una soluzione di ripiego, eventualmente temporanea, oppure un banco di prova propedeutico alla realizzazione di un vero e proprio KMD.

È altresì vero che consentire ad un applicativo (sia pure targato come UMD) di accedere a risorse hardware può rappresentare una palese violazione dei principi che hanno ispirato l'architettura dei moderni sistemi operativi. La tecnica dell'accesso diretto ha come possibile alternativa l'utilizzo di un KMD come "proxy" verso l'I/O. Questa ultima soluzione salva la forma, ma non altera la sostanza. È quanto meno prestazionalmente non vantaggiosa e in qualche caso inapplicabile. Semmai la vera obiezione dovrebbe essere che un UMD come surrogato di un KMD ha limiti non superabili. Tanto per citare un esempio, non può essere impiegato per dispositivi che facciano richieste di interruzione.

In effetti non crediamo di incoraggiare questa pratica per il solo fatto di descriverla. Il lettore saprà scegliere.

Antoninio Calderone acalderone@infomedia.it

Lavora come software engineer presso Marconi Communications SpA nella divisione Optical Networks. Inoltre collabora con la DigiFacta SW Engineering S.r.l. Si occupa principalmente di sistemi e protocolli per le telecomunicazioni, firmware, device driver, internals di sistemi operativi.

I/O permission level e I/O permission bit map

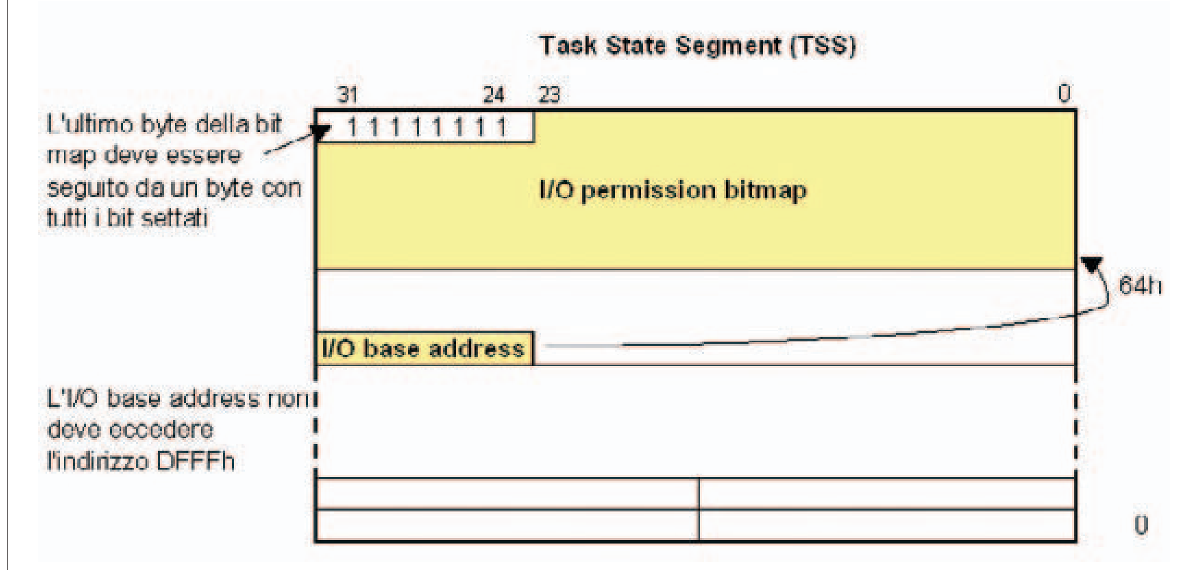
I processori x86 usano un algoritmo per validare l'accesso a una porta di I/O basato su due distinti meccanismi di "permission check" (è possibile trovarne una trattazione dettagliata in [1]):

- Check del campo I/O Privilege Level (IOPL) del registro EFLAGS.
- Check dell'I/O permission bit map (IOPM) del task state segment (TSS) di un processo.

In caso di dispositivi memory-mapped, vengono utilizzati i meccanismi di protezione propri del gestore di memoria, che non approfondiremo oltre in questa sede (chi fosse interessato può trovare informazioni dettagliate consultando [2]).

Il registro (a 32-bit) EFLAGS contiene i flag di stato, di controllo e di sistema. Lo stato del registro

FIGURA 1 L'I/O permission bit map, e il suo base–address, all'interno del TSS



EFLAGS viene salvato nel TSS alla sospensione di un task e viene rimpiazzato con il valore prelevato dal TSS del nuovo task da mandare in esecuzione. Il campo IOPL di questo registro controlla l'accesso allo spazio di indirizzamento delle porte di I/O, restringendo l'utilizzo delle istruzioni macchina che operano su tali porte. Le istruzioni macchina IN, INS, OUT, OUTS possono essere eseguite se il Current Privilege Level (CPL) del processo in esecuzione è minore o uguale all'IOPL del registro EFLAGS. Queste istruzioni (alle quali si aggiungono le istruzioni che manipolano all'attivazione dell'interrupt-enable flag, STI e CLI) si dicono *I/O sensitive*. Qualunque tentativo di violare la restrizione all'uso di tali istruzioni, da parte di un processo non privilegiato, ha come risultato il sollevamento da parte del microprocessore di una eccezione di protezione generale. Dato che ciascun processo ha una propria copia del registro EFLAGS, processi differenti possono avere differenti livelli di privilegio. I processi utente (CPL=3) non possono comunque modificare direttamente l'IOPL, ma devono giovarsi dei servizi del sistema operativo. Linux fornisce una chiamata di sistema denominata *iopl*. Questa funzione consente a un processo utente che abbia i privilegi di root o acquisisca la "capability" *CAP_SYS_RAWIO*, di modificare il campo IOPL, e ottenere l'accesso diretto a tutto lo spazio di indirizzamento dell'I/O. È possibile trovare l'implementazione della syscall *iopl* (sys_iopl) e della syscall *ioperm* (sys_ioperm) - della quale parleremo tra poco - consultando i sorgenti di Linux (visionabili anche all'URL [8]).

L'I/O permission bit map si trova nel TSS. L'indirizzo del primo byte (base-address) e la sua locazione (posti anch'essi nel TSS) possono comunque variare. Extra byte settati a 1, dopo la bit map, sono necessari per prevenire che accessi non allineati al limite dello spazio di indirizzamento provochino una eccezione. Il base address dell'I/O permission bit map non può essere in ogni caso maggiore o uguale al limite del TSS: il processore interpreterebbe questo come mancanza della bit map stessa; il base address della bit map deve avere offset minore o uguale a DFFFh.

Agendo sull'I/O permission bit map (**Figura 1**) è possibile abilitare l'accesso alle porte di I/O anche per programmi o processi meno privilegiati (CPL>=IOPL) oppure eseguiti nella modalità virtual-8086. La bit map può coprire l'intero spazio di indirizzamento delle porte di I/O (o un sottoinsieme di questo spazio). Ogni bit di questa mappa corrisponde a un byte di indirizzo di I/O. Per esempio, la porta con "size" di un byte all'indirizzo 0x31 corrisponde al bit in posizione 1 del settimo byte della bit map. Un processo non privilegiato può accedere a un indirizzo di I/O, purché il corrispondente bit nella mappa valga zero.

In Linux, utilizzando la primitiva *ioperm* è possibile alterare i bit dell'I/O permission bit map delle prime 0x3FF porte (per indirizzi superiori a questo è necessario usare invece la syscall iopl).

Ke386IoSetAccessProcess e Ke386SetIoAccessMap

Su Windows non esistono chiamate di sistema analoghe alle system call ioperm e iopl di Linux, ma è possibile rimediare implementando un Kernel Mode Driver (KMD) che si faccia carico di modificare l'IOPL oppure - ed è la soluzione prescelta - l'I/O permission bit map per conto di un processo utente. Scrivere un siffatto KMD non presenta particolari difficoltà. Ne presentiamo una possibile implementazione nel

LISTATO 1 Codice sorgente del KMD per Windows (continua...)

```
#include <ntddk.h>
#include "ioperm_devcntl.h"

#define IO_BITMAP_BITS  65536
#define IO_BITMAP_BYTES (IO_BITMAP_BITS/8)
#define IOPM_BITMAP_SIZE IO_BITMAP_BYTES
#define PRIVATE static

PRIVATE WCHAR DEVICE_NAME[] = L"\\Device\\ioperm";
PRIVATE WCHAR DEVICE_DOS_NAME[] = L"\\DosDevices\\ioperm";
PRIVATE UNICODE_STRING device_dos_name, device_name;

typedef unsigned char* iopm_bitmap_t;
PRIVATE iopm_bitmap_t iopm_bitmap_copy_ptr = 0;

void Ke386IoSetAccessProcess(PEPROCESS, int);
void Ke386SetIoAccessMap(int, iopm_bitmap_t);

NTSTATUS ioperm_devcntrl(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp );

NTSTATUS ioperm_create(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp) {
  Irp->IoStatus.Status = STATUS_SUCCESS;
  Irp->IoStatus.Information=0;

  IoCompleteRequest(Irp, IO_NO_INCREMENT);

  return(STATUS_SUCCESS);
}

NTSTATUS ioperm_close(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp) {
  Irp->IoStatus.Status = STATUS_SUCCESS;
  Irp->IoStatus.Information=0;

  IoCompleteRequest(Irp, IO_NO_INCREMENT);

  return(STATUS_SUCCESS);
}

VOID ioperm_unload(IN PDRIVER_OBJECT DriverObject) {
  if (iopm_bitmap_copy_ptr) {
    MmFreeNonCachedMemory(iopm_bitmap_copy_ptr, IOPM_BITMAP_SIZE);
  }

  IoDeleteSymbolicLink (&device_dos_name);
  IoDeleteDevice(DriverObject->DeviceObject);
}

NTSTATUS DriverEntry( IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath ) {
  PDEVICE_OBJECT deviceObject = 0;
  NTSTATUS status = STATUS_SUCCESS;

  iopm_bitmap_copy_ptr = MmAllocateNonCachedMemory(IOPM_BITMAP_SIZE);

  if (! iopm_bitmap_copy_ptr)  {
    status = STATUS_INSUFFICIENT_RESOURCES;
    goto error_handler;
  }

  RtlInitUnicodeString(&device_name, DEVICE_NAME );
  RtlInitUnicodeString(&device_dos_name, DEVICE_DOS_NAME );

status = IoCreateDevice(DriverObject, 0,
                          &device_name,
```

LISTATO 1 (...fine)

```
                        FILE_DEVICE_UNKNOWN, 0, FALSE,
                      &deviceObject);

  if (! NT_SUCCESS(status) ) {
    goto error_handler;
  }

  if (! NT_SUCCESS(status = IoCreateSymbolicLink (&device_dos_name, &device_name))) {
    goto error_handler;
  }

  DriverObject->MajorFunction[IRP_MJ_CREATE] = ioperm_create;
  DriverObject->MajorFunction[IRP_MJ_CLOSE] = ioperm_close;
  DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = ioperm_devcntrl;
  DriverObject->DriverUnload = ioperm_unload;

error_handler:
  return status;
}

NTSTATUS
ioperm_devcntrl( IN PDEVICE_OBJECT DeviceObject, IN PIRP pIrp ) {
  NTSTATUS status = STATUS_SUCCESS;
  pIrp->IoStatus.Information = 0;

  switch (IoGetCurrentIrpStackLocation(pIrp)->Parameters.DeviceIoControl.IoControlCode) {
    case IOCTL_IOPM_ENABLE_IO:
      RtlZeroMemory(iopm_bitmap_copy_ptr, IOPM_BITMAP_SIZE);
      break;

    case IOCTL_IOPM_DISABLE_IO:
      RtlFillMemory(iopm_bitmap_copy_ptr, IOPM_BITMAP_SIZE, -1);
      break;

    default:
      status = STATUS_UNSUCCESSFUL;
      break;
  }

  if ( NT_SUCCESS( status ) ) {
    PEPROCESS current_process = IoGetCurrentProcess();

    if (current_process) {
      Ke386SetIoAccessMap(1, iopm_bitmap_copy_ptr);
      Ke386IoSetAccessProcess(current_process, 1);
    }
  }

  pIrp->IoStatus.Status = status;
  IoCompleteRequest( pIrp, IO_NO_INCREMENT );

  return status;
}
```

Listato 1, che discuteremo più in dettaglio nel successivo paragrafo. L'unica vera peculiarità del KMD realizzato è l'impiego delle funzioni "undocumented" *Ke386IoSetAccessProcess* e *Ke386SetIoAccessMap*. Una documentazione "non ufficiale" delle primitive può essere trovata in [5] e in [6]; per comodità la riassumiamo nel **Riquadro 1**. Prima di procedere nella descrizione del driver, illustriamo un frammento di codice, che mostra come queste due funzioni possano essere congiuntamente impiegate per aggiornare l'I/O permission bit map del processo chiamante, abilitando per esso l'accesso a tutto lo spazio di indirizzamento delle porte di I/O.

```
char * pIOPM = NULL;
//...
```

```
pIOPM = MmAllocateNonCachedMemory(65536
                                  / 8);
RtlZeroMemory(pIOPM, 65536 / 8);
Ke386IoSetAccessProcess(IoGetCurrentPro
cess(), 1);
Ke386SetIoAccessMap(1, pIOPM);
```

Nell'esempio viene allocata una nuova bit map di 64 Kbit e ne viene azzerato il contenuto. La nuova bit map viene usata per rimpiazzare quella originale, in particolare la funzione Ke386IoSetAccessMap sostituisce la bit map, operando "nel contesto" selezionato dalla funzione Ke386IoSet AccessProcess.

Il Kernel Mode Driver

Il ruolo tipico di un device driver è quello di affiancarsi al kernel del sistema operativo per pilotare specifiche porzioni di hardware del sistema. Un driver, in un certo senso, fornisce l'implementazione dell'interfaccia del sistema di I/O per i dispositivi da esso pilotati.

Dal punto di vista dell'utente, i kernel mode driver possono essere acceduti come speciali file, attraverso funzioni del sottosistema *Win32* (quali per esempio *CreateFile*, *ReadFile*) oppure attraverso la funzione *DeviceIoControl*.

Dal punto di vista dello sviluppatore sono un insieme di routine che, invocate dello *I/O Manager* di Windows, processano le varie fasi di una richiesta di input o di output del dispositivo che controllano.

Tipicamente un KMD di Windows è composto da:

- la funzione *DriverEntry*: invocata dallo I/O Manager non appena il driver viene caricato;
- le routine per lo smistamento di richieste (*dipatch entry point*): invocate su richieste di I/O attraverso il trasferimento di *I/O Request Packet* (*IRP*);
- *Interrupt Service Routine* (*ISR*): alle quali viene trasferito il controllo in presenza di interrupt sollevate da parte del dispositivo controllato;
- *DeferredProcedure Call* (*DPC*): speciali procedure tipicamente invocate all'interno di ISR per completare, in particolari circostanze, il compito delle ISR stesse.

LISTATO 2 Codice sorgente del programma di prova per Windows/Linux (continua...)

```
#if defined (_MSC_VER)
#define PER_WIN32
# include <windows.h>
# include <winioctl.h>
# include "ioperm_devcntl.h"
# include <conio.h>
# include <stdio.h>
#elif defined (__GNUC__)
# include <sys/io.h>
# define PER_LINUX
# include <stdio.h>
# include <stdlib.h>
#define _inp inb_p
#define _outp(_PORT_,_VALUE_) outb_p(_VALUE_,_PORT_)
#else
# error Please use GNU C++ for Linux or MS Visual C++ compiler for Windows
#endif

// Usual parallel port addresses
unsigned short port_addr[] = {0x378, 0x278, 0x3BC};

#define MAGIC_NUMBER 0xAC

int main()
{
#ifdef PER_WIN32
  DWORD dwBytesReturned = 0;
  #define        MAX_ERR_STR 256
  char szError[MAX_ERR_STR] = {0};

  //CreateFile opens the device driver and returns a handle that will be used
  //to access the device driver
  HANDLE h = CreateFile("\\\\.\\ioperm",                    // file name
                      GENERIC_READ | GENERIC_WRITE,         // access mode
                      FILE_SHARE_READ | FILE_SHARE_WRITE,   // share mode
                      0,                                    // Security Attri-
                                                                        butes
                      OPEN_EXISTING,              // how to open
                      0,                          // file attributes: normal
                      0);                         // handle to template file:
                                                                         NONE


  //Check for open errors
  if (h == INVALID_HANDLE_VALUE) {
    sprintf(szError, "Error %ld", GetLastError());
    MessageBox(0, "IoPerm Error", szError, MB_OK|MB_ICONERROR);
    return -1;
  }

  // The DeviceIoControl function sends the IOCTL_IOPM_ENABLE_IO
  // control code directly to a ioperm device driver,
  // causing the device to perform the ioperm operation enabling or
  // disabling direct I/O access for calling process
  if (! DeviceIoControl(h,                    // handle to device
                      IOCTL_IOPM_ENABLE_IO, // operation
                      NULL,                 // no input data buffer
                      0,                    // size of input data buffer is 0
                      NULL,                 // no output data buffer
                      0,                    // size of output data buffer is 0
                      &dwBytesReturned,     // byte count
                      NULL))                // overlapped information
  {
```

LISTATO 2 (...fine)

```
   sprintf(szError, "Error %ld", GetLastError());
    MessageBox(0, "IoPerm Error", szError, MB_OK|MB_ICONERROR);
    CloseHandle(h);

 return -1;
  }

  //We need to re-schedule
  Sleep(1);

#elif defined(PER_LINUX)
  if (iopl(3)) {
    perror("iopl");
    exit(-1);
  }
#endif


  // Ok, device opened
  for (unsigned int i=0;
       i<sizeof(port_addr)/sizeof(port_addr[0]);
       ++i)
  {
    _outp(port_addr[i], MAGIC_NUMBER);

    if ( MAGIC_NUMBER == _inp(port_addr[i]) ) {
      printf("Parallel port found at address 0x%X ", port_addr[i]);
      printf("(Status Info 0x%02X - Control Info 0x%02X)\n",
             _inp(port_addr[i] + 1),
             _inp(port_addr[i] + 2));
    }
  }

#ifdef PER_WIN32
  CloseHandle(h);
#endif

  return 0;
}
```

Il device driver del **Listato 1** ovviamente non richiede la gestione di ISR o l'uso DPC (per una completa trattazione dell'argomento si consiglia quindi di consultare [3] e [4]). Peraltro, questo driver necessita di essere pilotato per abilitare o disabilitare l'I/O permission check. Operazione che verrà svolta in risposta a una richiesta di «ioctl». L'»entry point» nel driver è la funzione DriverEntry. Il prototipo della funzione *DriverEntry* è il seguente:

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

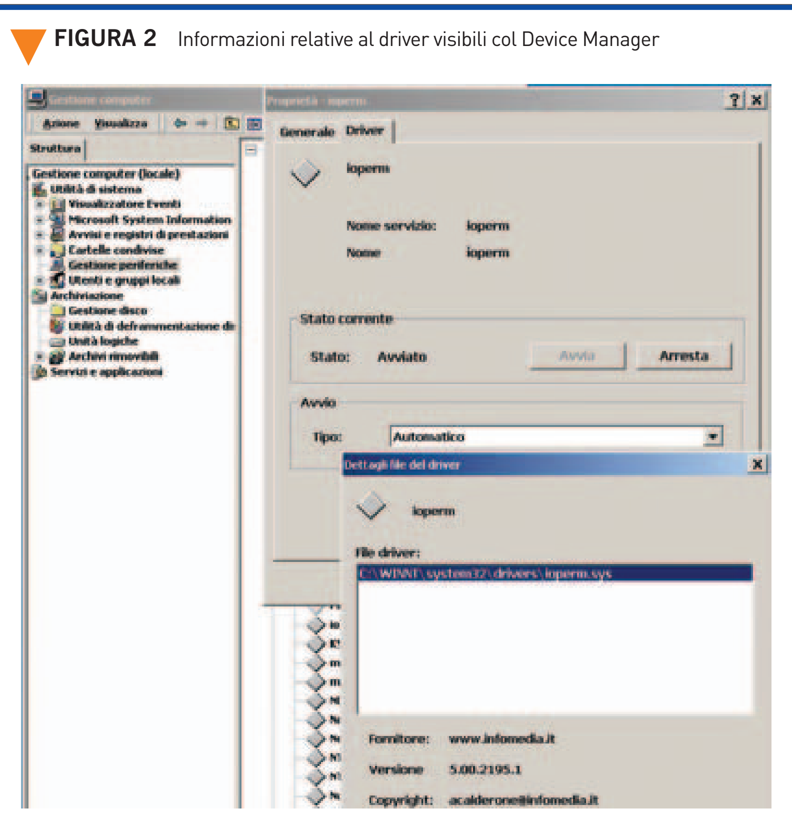
Questa funzione accetta il puntatore a una struttura di tipo *DRIVER_OBJECT*. Il *driver object* è costruito dal sistema su richiesta di attivazione dello stesso, ed è unico per il driver. Il *RegistryPath* è un puntatore a una stringa unicode contenente il registry path name, che naturalmente si riferisce a una chiave del registro di configurazione di Windows. Il valore di ritorno della funzione *DriverEntry* è usato dallo I/O Manager: un valore diverso da *STATUS_SUCCESS* causa l'immediata terminazione del driver che viene rimosso dalla memoria. La funzione *DriverEntry* crea un *device object* (almeno uno) associando a questo un nome ed un (eventuale) link simbolico; in ultimo registra le routine per il dispatching delle richieste (che per il nostro driver sono *ioperm_create*, *ioperm_close*, *ioperm_devcntrl*). Queste routine prendono il nome di *Dispatch entry point*. Lo smistamento delle richieste di I/O viene fatto processando un *I/O Request Packet* o IRP. Il sistema di I/O di Windows è infatti packet-driven. L'I/O Manager costruisce un IRP come risultato di una richiesta di uno dei servizi di I/O di sistema. Ciascun IRP è una struttura avente una parte fissa (*header*) e una o più parti strettamente correlate alle particolari attività del driver chiamate *I/O stack location*. Il nostro device esporta il solo "metodo" per la gestione di due particolari richieste di I/O control: l'abilitazione o la disabilitazione dell'accesso diretto all'I/O. In Linux non è possibile ottenere lo stesso risultato usando ioperm ma, come già spiegato, impiegando la primi-

Funzione	Descrizione
void Ke386IoSetAccessProcess (PEPROCESS, int);	Questa funzione informa il kernel che un certo processo intende accedere all'I/O permission bit map. Il secondo argomento presumibilmente, abilita (1) oppure disabilita (0) l'accesso all'IOPM.
void Ke386SetIoAccessMap (int, IOPM *);	Questa funzione consente di associare una nuova bit map al processo. Se il primo parametro è 1, il vecchio IOPM viene sovrascritto col nuovo; il puntatore alla nuova bit map è passato come secondo argomento.
void Ke386QueryIoAccessMap (int, IOPM *);	Questa funzione restituisce l'IOPM corrente. Il significato del primo argomento non è chiaro, ma perché la funzione restituisca la bit map correttamente è necessario che questo sia 1.

RIQUADRO 1 Documentazione "non ufficiale" delle primitive dell'API nativa di Windows usate nel Driver

tiva iopl. Infatti la syscall ioperm, per ragioni storiche, può modificare solo i permessi per le prime 0x3ff porte. Per generare il modulo binario del driver abbiamo utilizzato il Driver Development Kit (DDK) di Microsoft [7], al quale rimandiamo per approfondire gli aspetti legati alla software-factory. Una volta installato, il DDK mette a disposizione un ambiente completo di tutti gli strumenti necessari per la generazione del driver vero e proprio. Per compilare il driver è sufficiente lanciare l'utility *build* a partire dalla directory nella quale sia stato copiato il codice sorgente. Il risultato della compilazione è un file con estensione .sys (nella fattispecie ioperm.sys). L'installazione del device può essere fatta popolando opportunamente il registro di sistema e copiando il file .sys prodotto nell'area *system32\drivers* a partire dalla directory di installazione di Windows. Riportiamo di seguito un file .REG impiegabile per la registrazione del device driver:

FIGURA 2 Informazioni relative al driver visibili col Device Manager




```
Windows Registry Editor Version
5.00

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\
                                    Services\ioperm]
"Type"=dword:00000001
"ErrorControl"=dword:00000001
"Start"=dword:00000002
"DisplayName"="ioperm"
"ImagePath"="System32\\DRIVERS\\ioperm.sys"
```

Le corrispondenti imposta-zioni sono visibili e configurabili usando il Device Manager di Windows (come illustrato in **Figura 2**). Il banco di prova che proponiamo è un piccolo programma che effettua il probing della porta parallela.

Questo dispositivo in genere è mappato a partire da uno dei seguenti indirizzi: 0x278, 0x378, 0x3BC. Il registro dati (offset 0) è un data latch a 8 bit. Sfruttando questo fatto, il programma prova a scrivere e poi rileggere un byte nel registro dati. A patto di non scegliere 0 o 0xFF come valore (tipico valore di pull-down o pull-up) si può essere ragionevolmente sicuri, in caso di presenza della porta parallela, di rileggere quanto scritto. Il **Listato 2** riporta il codice del programma che è stato compilato e testato usando il compilatore C++ dell'ambiente Visual Studio 6.0 di Microsoft e il GNU C++ 3.4.x in ambiente GNU/Linux.

Conclusioni

Andando a ritroso nelle versioni di Linux, è possibile verificare che le syscall ioperm e iopl sono state inserite sin dagli albori del kernel. Non ci stupisce questo fatto, e nemmeno la scelta diametralmente opposta operata da Microsoft.

Bibliografia

[1] Intel - "*Intel Architecture Software Developer's Manual - Volume 1:Basic Architecture*", Intel, 1999
[2] Intel - "*Intel Architecture Software Developer's Manual, Volume 3*" Intel, 1999
[3] P.G. Viscarola & W.A. Mason - "*Windows NT - Device Driver Development*", MTP, 1999
[4] David A. Salomon - "*Inside Microsoft Windows NT - 2nd Edition*", Microsoft Press, 1998

Riferimenti

[5] http://www.ddj.com/articles/1996/9605/
[6] http://www.beyondlogic.org/porttalk/porttalk.htm
[7] http://www.microsoft.com/whdc/devtools/ddk/default.mspx
[8] http://lxr.linux.no/source/arch/i386/kernel/ioport.c

CODICE ALLEGATO

ftp.infomedia.it **Driver**