Il Nostro Primo Driver

Infos

Author:	Quequero
Email:	quequero@bitchx.it
Website:	http://quequero.org

Date:	20/04/2008 (dd/mm/yyyy)
Level:	
Language:	Italian 
Comments:	Blow Jobs, Hand Jobs, Steve Jobs!

Introduzione

Oggi impareremo a scrivere un driver per Windows Xp/Vista ed impareremo ad utilizzare le API per consentire il caricamento del driver dall'userspace. Scopriremo quindi come è possibile far comunicare un componente che gira ad userspace con il nostro driver, ed in barba a quanto avevamo imparato la scorsa volta, non useremo il template di Visual Studio. C'e' una ragione non vi preoccupate, e la vedremo più in la'. Per ora... Procuratevi i tools necessari!

Tools

- Windows Driver Kit ma se non ce l'avete anche il DDK va bene.
- QDriverLoader, o il driver loader che preferite.
- DbgView
- Notepad++ o l'editor che preferite.

Sorgenti

- Driver Base, driver base, potete utilizzarlo anche come template di partenza per nuovi driver.
- Driver Completo, il driver completo come visto nel tutorial.
- Interfaccia userspace, il programma in userspace che comunica con il driver.

Essay

L'ultima volta avevamo visto come era possibile compilare un driver con VisualStudio, tuttavia il template che viene utilizzato per default è già completo di codice per l'inizializzazione del driver e la creazione di un device. Mi è sembrato quindi doveroso spiegare cosa fosse quel codice ed ecco il perché di questo tutorial :), detto cio'... Diamoci dentro! :).

L'userspace

Prima di creare il nostro driver dobbiamo imparare a caricarlo dall'userspace, altrimenti non saremo in grado di utilizzare il *.sys* che creeremo a breve. La procedura è semplice ed è bene conoscerla poiché prima o poi avremo bisogno di caricare un driver :). Fortunatamente esistono delle API create per tale proposito e sono:

- *OpenSCManager()*
- *CreateService()*
- *OpenService()*
- *StartService()*
- *QueryService()*

Dai nomi è chiaro che dovremo trattare un driver come se fosse a tutti gli effetti un servizio. Vediamo quindi, passo dopo passo, come si fa a creare/avviare questo servizio. La prima cosa da fare è aprire il *Service Control Manager* (SCM), si tratta di un server RPC che gestisce il database dei servizi: può aggiungerli, rimuoverli, richiederne lo stato, stopparli, scaricarli etc... Ed è quel componente di Windows che utilizziamo quando scegliamo i servizi da avviare al boot. Quindi per caricare un driver, che come abbiamo appena detto è un servizio, dovremo parlare col *Service Control Manager*, per farlo il primo passo sarà quello di stabilire una connessione al database tramite l'API *OpenSCManager()* che prende i seguenti parametri:

```
SC_HANDLE WINAPI OpenSCManager(
  __in_opt  LPCTSTR lpMachineName,
  __in_opt  LPCTSTR lpDatabaseName,
  __in      DWORD dwDesiredAccess
);
```

Che sono, nell'ordine:

1. Il nome della macchina
2. Il nome del database
3. Il tipo di operazione che vogliamo eseguire

Possiamo richiamare la funzione così:

```
SC_HANDLE hService = OpenSCManager(NULL, NULL,
SC_MANAGER_CREATE_SERVICE);
```

In questo modo chiediamo l'apertura sulla macchina locale del database *SERVICES_ACTIVE_DATABASE*, richiedendo i permessi per la creazione di un servizio. Dopo di cio' dovremo creare effettivamente il servizio utilizzando la *CreateService()*:

```
SC_HANDLE WINAPI CreateService(
  __in       SC_HANDLE hSCManager,
  __in       LPCTSTR lpServiceName,
  __in_opt   LPCTSTR lpDisplayName,
  __in       DWORD dwDesiredAccess,
```

```
  __in       DWORD dwServiceType,
  __in       DWORD dwStartType,
  __in       DWORD dwErrorControl,
  __in_opt   LPCTSTR lpBinaryPathName,
  __in_opt   LPCTSTR lpLoadOrderGroup,
  __out_opt  LPDWORD lpdwTagId,
  __in_opt   LPCTSTR lpDependencies,
  __in_opt   LPCTSTR lpServiceStartName,
  __in_opt   LPCTSTR lpPassword
);
```

si tratta di un'API che prende molti parametri, eccoli spiegati rapidamente:

1. *hSCManager* è l'handle al control manager ottenuto con *OpenSCManager()*.
2. *lpServiceName* è una stringa che specifica il nome da dare al servizio.
3. *dwDesiredAccess* è il tipo di accesso che richiediamo.
4. *dwServiceType* è il tipo di servizio che vogliamo creare.
5. *dwStartType* è il tipo di avvio (automatico, al boot, manuale etc...).
6. *dwErrorControl* specifica cosa fare in caso di errore in fase di loading.
7. *lpBinaryPathName* è il path dove si trova il file del driver.
8. *lpLoadOrderGroup* il gruppo al quale vogliamo associare il driver.
9. *lpdwTagId* è una variabile che riceve il tag del gruppo.
10. *lpDependencies* stabilisce le dipendenze, (ad esempio il firewall avrà bisogno del servizio di networking etc...).
11. *lpServiceStartName* è l'account sotto il quale girerà il servizio.
12. *lpPassword* la password dell'account, per i driver non serve.

Nonostante i parametri siano molti non tutti sono necessari, ecco quindi come possiamo chiamare questa funzione:

```
SC_HANDLE hDriver = CreateService(hService, "My Driver", "This is my
first windows driver!",
                                  SERVICE_ALL_ACCESS,
SERVICE_KERNEL_DRIVER,
                                  SERVICE_DEMAND_START,
SERVICE_ERROR_NORMAL,
                                  "c:\\driver.sys", NULL, NULL, NULL,
NULL, NULL);
```

E quindi dopo aver creato il servizio dovremo anche avviarlo, lo faremo utilizzando *StartService()*:

```
BOOL WINAPI StartService(
  __in      SC_HANDLE hService,
  __in      DWORD dwNumServiceArgs,
  __in_opt  LPCTSTR* lpServiceArgVectors
);
```

Dove:

1. *hService* è l'handle ottenuto da *CreateService()* o *OpenService()*.
2. *dwNumServiceArgs* è il numero di argomenti da passare al service.
3. *lpServiceArgVectors* è la lista dei parametri.

Anche stavolta l'utilizzo dell'API è semplice poiché non dobbiamo passare nulla al driver:

```
StartService(hDriver, 0, NULL);
```

Fatto cio', il nostro driver verrà avviato, posto il fatto di essere amministratori ovviamente. Per fermare il driver dovremo invece aprire il SCM con *OpenSCManager()*, aprire il servizio con *OpenService()*, fermarlo con *ControlService()* e quindi cancellarlo con *DeleteService()*, una buona pratica è anche quella di querare lo stato del servizio per vedere se effettivamente si è fermato, oppure se non è già in fase di stop, comunque in breve ecco la procedura di stop e delete di un servizio:

```
SERVICE_STATUS ss;
SC_HANDLE hService, hDriver;

hService = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
hDriver = OpenService(hService, DRIVER_DESC, SERVICE_ALL_ACCESS);
ControlService(hDriver, SERVICE_CONTROL_STOP, &ss));
DeleteService(hDriver);
CloseServiceHandle(hDriver);
CloseServiceHandle(hService);
```

Non vi preoccupate, ho codato il QDriverLoader proprio per evitare a voi lo stesso calvario ;p. Comunque ora che sappiamo come avviare un driver... È giunto il momento di crearlo! :).

Il Driver

Magari siete curiosi e quindi non voglio farvi aspettare, passiamo direttamente alla creazione del driver! Per prima cosa installiamo il WDK/DDK (in una directory che **non** contenga spazi nel path), in questo modo avremo tutti gli header e librerie necessarie per utilizzare le funzioni di cui avremo bisogno. Fatto cio' creeremo i file necessari alla compilazione e poi scriveremo il codice, ma lo faremo in due step: nel primo creeremo soltanto le funzioni base di inizializzazione ed uscita, nel secondo aggiungeremo la gestione delle IOCTL e la creazione del device.

Directory Tree

Creiamo una nuova cartella, in un path che non contenga spazi, e creiamo al suo interno il *tree* di file di cui avremo bisogno, ovvero:

- sources (conterrà le direttive per il Makefile).
- Makefile (conerrà le direttive per la compilazione).
- driver.c (il sorgente del driver).

Apriamo con *notepad++* il file *sources* e riempiamolo come di seguito:

```
TARGETNAME = Driver
TARGETPATH = obj
TARGETTYPE = DRIVER

INCLUDES = %BUILD%\inc
LIBS = %BUILD%\lib

SOURCES = driver.c
```

Le direttive sono abbastanza esplicative, l'importante è che la variabile *SOURCES* punti al nome di tutti i file *.c* che è necessario compilare, il nome finale del driver sarà quello specificato dalla direttiva *TARGETNAME*, a cui verrà logicamente accodata l'estensione *.sys*. Apriamo ora il file *Makefile* e riempiamolo come segue:

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

Non serve altro, in questo modo impostiamo l'ambiente di default con le definizioni standard che sono già presenti nel WDK/DDK. Ed ora non ci resta che scrivere il codice nel driver.

Driver Source

Iniziamo con la versione base del driver: dovremo definire una funzione di inizializzazione ed una di uscita, vediamole:

```
// driver.c
#include <ntddk.h>

void DriverUnload(PDRIVER_OBJECT pDriverObject)
{
     DbgPrint("Que - Driver unloading\n");
}

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING
RegistryPath)
{
    // Stampa un messaggino
    DbgPrint("Que - Hello, World\n");

    // Setta il puntatore alla funzione di unload
    DriverObject->DriverUnload = DriverUnload;

    return STATUS_SUCCESS;
}
```

La prima funzione che incontriamo è *DriverUnload* che viene invocata quando decidiamo di scaricare il driver dalla memoria, non si tratta di una funzione necessaria quindi potremmo ometterla, ma così facendo non saremo in grado di scaricare il driver senza rebootare la macchina. La seconda funzione è invece la *DriverEntry* che è a tutti gli effetti la *main()* di un driver, possiamo rinominarla come desideriamo, ma avendo scelto di utilizzare l'ambiente di *make* di default, dovremo lasciarla così. Prima di proseguire alla compilazione spieghiamo cosa fanno:

DriverEntry() & DriverUnload()

Dal sorgente vediamo che questa funzione prende in ingresso due parametri: un puntatore ad una struttura di tipo *DRIVER_OBJECT* ed un puntatore ad una stringa UNICODE. La struttura *DRIVER_OBJECT* è molto interessante ed è definita nel file *wdm.h*, diamole uno sguardo:

```
typedef struct _DRIVER_OBJECT {
    CSHORT Type;
    CSHORT Size;
    PDEVICE_OBJECT DeviceObject;
    ULONG Flags;
    PVOID DriverStart;
    ULONG DriverSize;
    PVOID DriverSection;
    PDRIVER_EXTENSION DriverExtension;
    UNICODE_STRING DriverName;
    PUNICODE_STRING HardwareDatabase;
    PFAST_IO_DISPATCH FastIoDispatch;
    PDRIVER_INITIALIZE DriverInit;
```

```
    PDRIVER_STARTIO DriverStartIo;
    PDRIVER_UNLOAD DriverUnload;
    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT;
```

È una struttura decisamente ampia, ma non dovremo utilizzarla tutta quindi per il momento non vi preoccupate. Durante il nostro primo step inizializzeremo soltanto il membro *DriverUnload*, ma per completezza illustreremo anche la funzione di alcuni membri della *DRIVER_OBJECT*:

- *DeviceObject* contiene la lista di tutti i device che man mano vengono creati dal driver.
- *DriverStart* contiene l'indirizzo dove è stato caricato il driver.
- *DriverSize* contiene la dimensione del driver in memoria.
- *HardwareDatabase* è una stringa UNICODE che punta alla chiave di registro *\Registry\Machine\Hardware*, e può essere utilizzata dal driver per ottenere informazioni di configurazione delle varie componenti presenti sul computer.
- *DriverInit* viene inizializzato dall'I/O Manager e punta alla funzione di inizializzazione, la *DriverInit()* appunto.
- *DriverStartIo* deve essere inizializzato per puntare alla routine di *StartIo* che viene utilizzata per iniziare un'operazione di I/O su un dispositivo fisico.
- *DriverUnload* deve essere inizializzato per puntare alla routine di unload del driver.
- *MajorFunction[]* è una dispatch table che verrà usata per inserire gli hook alle funzioni di dispatch che ci interessa dirottare.

Il secondo parametro della *DriverEntry()* è *RegistryPath*, un puntatore ad una stringa UNICODE che viene inizializzata dall'I/O Manager (e liberata all'uscita della *DriverEntry*) e contiene il path di una chiave nel registro che, generalmente, viene generata da una chiamata alla *CreateService()* fatta dall'userspace quando creiamo il servizio. Tale chiave può essere utilizzata dal driver per memorizzare flag o informazioni di altro genere che possono tornare utili nelle esecuzioni successive, normalmente il path è *\Registry\Machine\System\CurrentControlSet\Services**NomeDriver***, dove *\Registry\Machine* è l'equivalente a kernel space di *HKEY_LOCAL_MACHNE*. La prima chiamata a funzione che incontriamo nel sorgente è:

```
DbgPrint("Que - Hello, World\n");
```

Che evidentemente stampa una stringa... Ma dove? Un driver non ha un contesto di output associato, quindi non potremo scrivere all'interno di un terminale, come faremmo normalmente con una *printf()*, e quindi dove finisce la stringa? Semplicemente viene inviata al kernel debugger presente sulla macchina, in questo caso non utilizzeremo *WinDBG*, ma ci faremo bastare il più comodo *DbgView*. Non dimenticate di accodare sempre un *\n* alla stringa da stampare altrimenti *DbgView* non visualizzerà niente dal momento che opera in maniera buffered, e quindi stampa solo quando incontra un a capo. La seconda riga di codice non fa altro che inizializzare il membro *DriverUnload* che risiede all'interno della struttura *DriverObject* per farlo puntare alla funzione di unload.

```
DriverObject->DriverUnload = DriverUnload;
```

Dopo di che il driver torna dalla funzione di inizializzazione con un bel *STATUS_SUCCESS* che sta ad indicare che va tutto bene. La funzione di unload è composta come segue:

```
void DriverUnload(PDRIVER_OBJECT pDriverObject)
{
     DbgPrint("Que - Driver unloading\n");
}
```

L'unico parametro è un puntatore ad una *DRIVER_OBJECT*, per il resto la funzione non fa proprio niente, a parte stampare un messaggino sul debugger per notificare l'utente che il driver è stato scaricato. Questa funzione non andrà mai invocata direttamente da noi, ci penserà l'I/O Manager a scaricare il driver

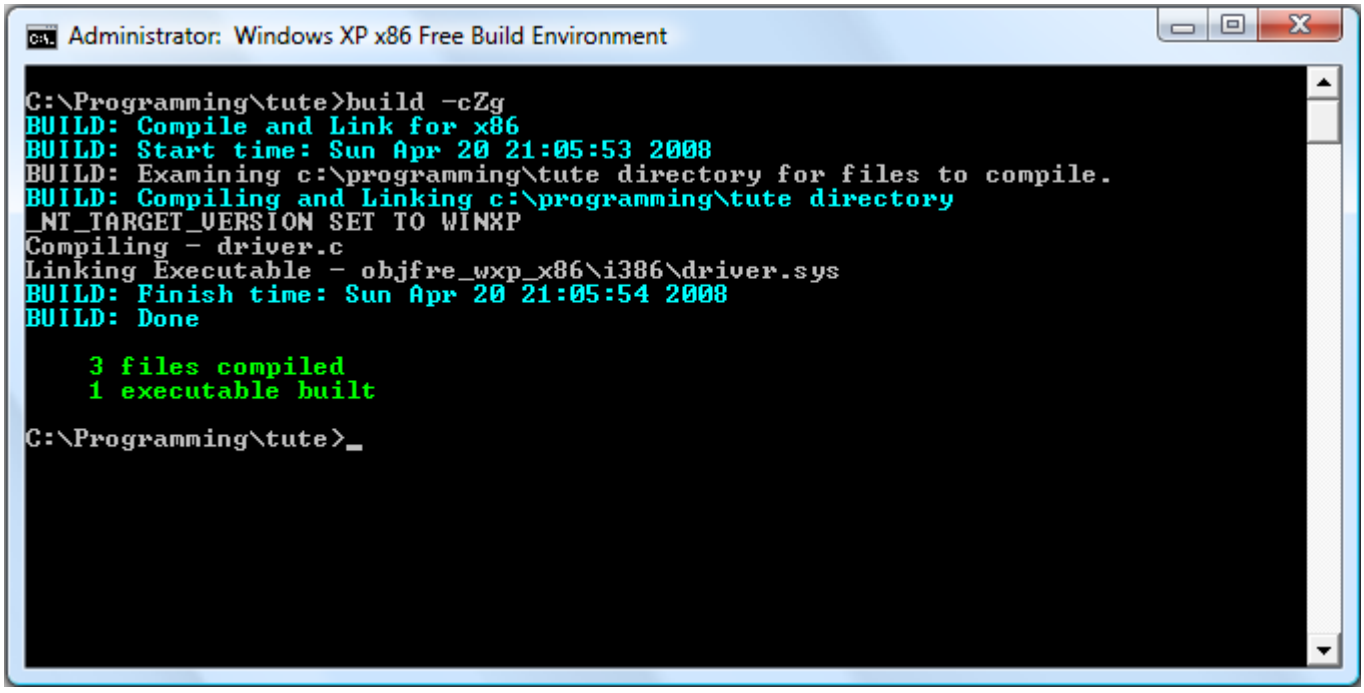
quando il *Service Manager* ne farà richiesta, o quando qualcuno chiamerà esplicitamente la *ZwUnloadDriver*. Fatto cio', passiamo alla compilazione e poi al caricamento del nostro primo driverino :).

Compile it!

Se abbiamo installato il WDK andiamo sul menu *Start* | *Windows Driver Kit* | *WDK 6000* | *Build Environments* (un path simile viene utilizzato dal DDK) e scegliamo l'ambiente su cui dovrà girare il nostro driver. Onde evitare di dover riscrivere questo tutorial da 0, faro' girare il driver all'interno di una macchina virtuale su cui è installato Windows XP, quindi scegliero': *Windows XP x86 Free Build*, voi potete scegliere l'os su cui vi trovate, in modo da poter testare il driver, sta a voi scegliere l'ambiente *Checked* o *Free*: se avete un'installazione standard di Windows dovete scegliere *Free*, se avete la versione di debug potete scegliere *Checked*. Questo prompt servirà per compilare il nostro sorgente, quindi trasferiamoci nella directory dove si trova il sorgente (il DDK non supporta le directory che contengono spazi nel path, tenetelo a mente) e digitiamo:

build -cZg

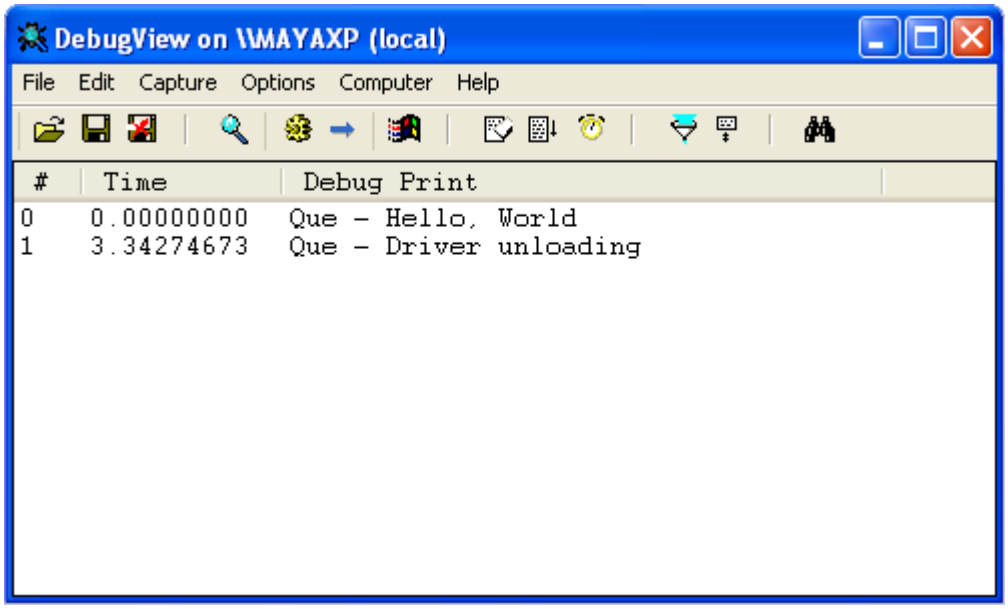
Se tutto va per il meglio vedremo qualcosa del genere:



Troveremo il nostro *driver.sys* all'interno di una sottodirectory creata dal *make* all'interno della directory dove sono i sorgenti, se invece abbiamo sbagliato qualcosa vedremo tante scritte in un bel rosso cangiante, simbolo universale di errore.

Load it!

Quando lavorate con i driver è cosa buona e giusta utilizzare una macchina virtuale, visto che ogni minimo errore può condurvi ad un triste BSOD con conseguente reboot/lockup della macchina. Detto cio' avviamo *DbgView* e nel menu *Capture* spuntiamo la riga *Capture Kernel*, in modo da intercettare i messaggi del kernel. Avviamo quindi QDriverLoader per caricare il driver, e tramite il *browse button* selezioniamo il nostro bel *driver.sys*, premiamo prima *Load Driver* per caricare il driver e poi *Unload Driver* per scaricarlo, se tutto è andato per il verso giusto, e quindi state ancora leggendo questa pagina, vedrete qualcosa di simile:



Orbene, per il momento abbiamo creato, compilato, caricato e scaricato un driver che non fa veramente niente, e nonostante questo il lavoro è stato notevole, ma almeno abbiamo creato una sorta di template che possiamo riusare per ogni nostro driver futuro. Adesso però vediamo di rendere funzionale il driver, o almeno proviamo a farlo comunicare con l'userspace.

I Device

Un driver per comunicare con l'userspace ha bisogno di stabilire un canale di comunicazione, tale canale è il *device object* che dall'userspace potremo trattare come se fosse un comune file, quindi potremo scrivere al suo interno, leggere dal device ed inviare dei messaggi di controllo. Ma questa non è l'unica *raison d'être* dei device object, infatti un driver dovrà necessariamente creare il proprio device object se desidera gestire le richieste di I/O di un determinato dispositivo. Infatti affinché un driver possa ricevere un **IRP** è necessario che sia presente un device, questa regola ha un'unica eccezione nei minidriver che sono associati ad un class driver. In ogni caso la creazione di un device è semplice e si effettua in tre passi, vediamone il codice:

```
const WCHAR kernLink[] =  L"\\Device\\Que";
const WCHAR userLink[]  = L"\\DosDevices\\Que";

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING
RegistryPath)
{
        NTSTATUS status;
        PDEVICE_OBJECT pDevice = NULL;
        UNICODE_STRING uDeviceName, uUserspaceName;

        RtlInitUnicodeString(&uDeviceName, kernLink);
        RtlInitUnicodeString(&uUserspaceName, userLink);

        status = IoCreateDevice(DriverObject, 0, &uDeviceName,
FILE_DEVICE_UNKNOWN,
                FILE_DEVICE_SECURE_OPEN, FALSE, &pDevice);

        if (!NT_SUCCESS(status))
        {
                DbgPrint(("Que - Couldn't create the device
object\n"));
```

```
                return status;
        }

        status = IoCreateSymbolicLink(&uUserspaceName, &uDeviceName);

        if(!NT_SUCCESS(status)) {
                DbgPrint(("Que - Couldn't create symlink to
device.\n"));
                return status;
        }
```

Per prima cosa dobbiamo decidere il nome del device nel kernel space ed il nome del device nell'userspace, dal momento che le nomenclature da utilizzare sono diverse, e dobbiamo inserire questi nomi all'interno di una stringa UNICODE, niente di complicato visto che esistono delle funzioni già pronte:

```
RtlInitUnicodeString(&uDeviceName, kernLink); // Il nome per il kernel
space
        RtlInitUnicodeString(&uUserspaceName, userLink); // Il nome
per l'userspace
```

Poi dobbiamo creare il device utilizzando la *IoCreateDevice()*:

```
NTSTATUS
  IoCreateDevice(
    IN PDRIVER_OBJECT  DriverObject,
    IN ULONG  DeviceExtensionSize,
    IN PUNICODE_STRING  DeviceName  OPTIONAL,
    IN DEVICE_TYPE  DeviceType,
    IN ULONG  DeviceCharacteristics,
    IN BOOLEAN  Exclusive,
    OUT PDEVICE_OBJECT  *DeviceObject
    );
```

Spieghiamone i parametri:

1. *DriverObject* è il driver object che possiamo prendere direttamente dalla *DriverEntry()*.
2. *DeviceExtensionSize* è il numero di byte da riservare alla device extension.
3. *DeviceName* è ovviamente il nome del device (dal kernel space).
4. *DeviceType* è il tipo di device.
5. *DeviceCharacteristics* stabilisce le caratteristiche del device.
6. *Exclusive* definisce se il device è di tipo esclusivo.
7. *DeviceObject' è la variabile che riceve il puntatore al device creato.*

Nel nostro caso creeremo un device generico, quindi non avremo bisogno di una alcuna device extension, useremo il tipo *FILE_DEVICE_UNKNOWN* e stabiliremo come caratteristica soltanto la *FILE_DEVICE_SECURE_OPEN* e non renderemo il device di tipo esclusivo, il codice che ne risulta è il seguente:

```
status = IoCreateDevice(DriverObject, 0, &uDeviceName,
FILE_DEVICE_UNKNOWN,
                        FILE_DEVICE_SECURE_OPEN, FALSE, &pDevice);
```

Dopo aver creato il device è necessario collegare l'oggetto generato nel kernel con quello dell'userspace, e lo faremo tramite un vero e proprio link simbolico:

```
status = IoCreateSymbolicLink(&uUserspaceName, &uDeviceName);
```

Ora il nostro device è pronto e possiamo utilizzarlo. Purtroppo però non è in grado di gestire nessun IRP e quindi anche aprendolo non potremo fare davvero nulla, percio' spieghiamo cos'e' un IRP ed ampliamo il driver in modo da aggiungere un handler.

Let's talk about IRP

Gli **IRP** (I/O Request Packet) sono strutture dati che vengono utilizzate dai driver per gestire e scambiare le richieste di I/O. Invece di passare a tutti gli strati una serie di dati, che sarebbe decisamente scomodo, viene riempita una struttura in grado di descrivere la richiesta di I/O che deve essere processata. I driver comunque non inviano l'IRP direttamente agli altri driver, ma lo fanno tramite l'I/O manager che, a sua volta, si prende carico del delivery ed invia l'IRP soltanto ai driver che possono gestire quella determinata richiesta. Se la richiesta di I/O non può essere completata immediatamente allora viene messa in una coda, al termine del processing il controllo può tornare di nuovo all'I/O manager se il driver richiama la *IoCompleteRequest()*.

IRP (I/O Request Packet)

È chiaro che gli IRP sono una parte fondamentale del driver coding su Windows, ed è quindi importante conoscerli bene per poterli utilizzare. La struttura, parzialmente opaca purtroppo, è la seguente:

```
typedef struct _IRP {
  .
  .
  PMDL  MdlAddress;
  ULONG  Flags;
  union {
    struct _IRP  *MasterIrp;
    .
    .
    PVOID  SystemBuffer;
  } AssociatedIrp;
  .
  .
  IO_STATUS_BLOCK  IoStatus;
  KPROCESSOR_MODE  RequestorMode;
  BOOLEAN PendingReturned;
  .
  .
  BOOLEAN  Cancel;
  KIRQL  CancelIrql;
  .
  .
  PDRIVER_CANCEL  CancelRoutine;
  PVOID UserBuffer;
  union {
    struct {
    .
    .
    union {
      KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
      struct {
        PVOID  DriverContext[4];
```

```
      };
    };
    .
    .
    PETHREAD  Thread;
    .
    .
    LIST_ENTRY  ListEntry;
    .
    .
    } Overlay;
  .
  .
  } Tail;
} IRP, *PIRP;
```

Questa invece è la struttura che viene utilizzata da Wine, contiene anche alcuni membri che non sono stati documentati ufficialmente da Microsoft:

```
typedef struct _IRP {
   CSHORT  Type;
   USHORT  Size;
   struct _MDL  *MdlAddress;
   ULONG  Flags;
   union {
     struct _IRP  *MasterIrp;
     LONG  IrpCount;
     PVOID  SystemBuffer;
   } AssociatedIrp;
   LIST_ENTRY  ThreadListEntry;
   IO_STATUS_BLOCK  IoStatus;
   KPROCESSOR_MODE  RequestorMode;
   BOOLEAN  PendingReturned;
   CHAR  StackCount;
   CHAR  CurrentLocation;
   BOOLEAN  Cancel;
   KIRQL  CancelIrql;
   CCHAR  ApcEnvironment;
   UCHAR  AllocationFlags;
   PIO_STATUS_BLOCK  UserIosb;
   PKEVENT  UserEvent;
   union {
     struct {
       PIO_APC_ROUTINE  UserApcRoutine;
       PVOID  UserApcContext;
     } AsynchronousParameters;
     LARGE_INTEGER  AllocationSize;
   } Overlay;
   PDRIVER_CANCEL  CancelRoutine;
   PVOID  UserBuffer;
   union {
     struct {
        union {
         KDEVICE_QUEUE_ENTRY  DeviceQueueEntry;
         struct {
           PVOID  DriverContext[4];
```

```
        } DUMMYSTRUCTNAME;
      } DUMMYUNIONNAME;
      PETHREAD  Thread;
      PCHAR  AuxiliaryBuffer;
      struct {
        LIST_ENTRY  ListEntry;
        union {
          struct _IO_STACK_LOCATION  *CurrentStackLocation;
          ULONG  PacketType;
        } DUMMYUNIONNAME;
      } DUMMYSTRUCTNAME;
      struct _FILE_OBJECT  *OriginalFileObject;
    } Overlay;
    KAPC  Apc;
    PVOID  CompletionKey;
  } Tail;
} IRP;
```

Gli IRP generati dall'I/O non sono comunque di dimensione fissa, la struttura esposta poco sopra è sempre la stessa, e quindi ha anche la stessa dimensione per tutti i request, ma ad ogni IRP vengono sempre associati uno o più I/O stack, o meglio: vengono accodati al pacchetto tanti I/O stack quanti sono i driver nella catena che possono gestire quel determinato I/O request. Vediamo ora la funzione dei membri della struttura che sono stati documentati da Microsoft:

- **MdlAddress**: MDL è l'acronimo di: *Memory Descriptor List* e rappresenta una struttura che descrive il buffer di memoria utilizzato dal richiedente.
- **Flags**: è un campo read-only che specifica alcune caratteristiche associate all'IRP.
- **AssociatedIrp.MasterIrp**: punta al master IRP se l'IRP è stato creato da un driver di più alto livello.
- **AssociatedIrp.SystemBuffer**: punta ad un buffer di sistema che può essere utilizzato per passare dei dati.
- **IoStatus**: rappresenta lo stato finale dell'IRP dopo esser stato processato.
- **RequestorMode**: indica la modalità di esecuzione del richiedente: UserMode o KernelMode.
- **PendingReturned**: se è settato a TRUE vuol dire che un driver ha marcato come pending questo IRP.
- **Cancel**: se settato a TRUE allora l'IRP è stato (o dovrebbe essere) cancellato.
- **CancelIrql**: contiene l'IRQL al quale gira il driver quando viene chiamata la *IoAcquireCancelSpinLock()*.
- **CancelRoutine**: punta alla *Cancel* routine, se è NULL l'IRP non può essere cancellato.
- **UserBuffer**: contiene l'indirizzo di un buffer di output.
- **Tail.Overlay.DeviceQueueEntry**: punta alla coda degli IRP del driver se l'IRP è stato accodato.
- **Tail.Overlay.DriverContext**: se l'IRP non è stato accodato, questo membro può contenere fino a 4 puntatori.
- **Tail.Overlay.Thread**: punta alla struttura ETHREAD del thread chiamante.
- **Tail.Overlay.ListEntry**: se il driver gestice una coda interna di IRP, questo campo collega gli IRP tra loro.

I/O Stack Location

Questo per quanto riguarda la parte fissa dell'IRP, per quanto concerne la coda degli stack abbiamo invece una pratica chiamata, la *IoGetCurrentIrpStackLocation()* che come dice il nome stesso ritorna un puntatore allo stack di I/O, come si può intuire il solo parametro necessario alla funzione è un puntatore all'IRP che stiamo processando. C'e' comunque da precisare che il dato tornato è un puntatore all'indirizzo di una struttura di tipo *IO_STACK_LOCATION*, che non studieremo per intero (e' decisamente grande) ma descriveremo 4 dei suoi membri più importanti:

- **MajorFunction**: indica il tipo di operazone di I/O da portare a termine, è **molto** importante come vedremo.
- **Parameters**: contiene 7 sotto-membri specifici di ogni *MajorFunction*.
- **DeviceObject**: contiene il puntatore ad un *device object* che è l'obiettivo di questa richiesta di I/O.
- **FileObject**: è il puntatore al *file object* associato a questa richiesta.

Adesso viene la parte interessante: abbiamo parlato di IRP che vengono inviati ai driver in grado di processarli, cio' vuol dire che possiamo inserire un driver nella coda per elaborare un determinato IRP, a questo punto è chiaro perché la programmazione a kernel space diventa interessante: perché possiamo davvero fare tutto. Immaginiamo infatti di poter intercettare e loggare/modificare tutte le richieste di lettura o scrittura che vengono fatte sul filesystem... Riuscite ad immaginarne utilizzi interessanti? Io no, e francamente preferisco la birra agli IRP :p. Ma facciamo finta di essere interessati all'argomento e rispondiamo professionalmente alla domanda: cerrrrrrrrrrto che si, potremmo modificare le richieste degli utenti, o magari nascondere dei file al fine di renderli realmente invisibili all'interno del sistema. Proviamo quindi ad inserirci nella coda degli IRP per gestire una richiesta di I/O proveniente dall'userspace.

IOCTL & Device Control

IOCTL significa: I/O Control, ed è quello che utilizzeremo tra poco per inviare un comando al driver dall'userspace. Per intercettare un certo tipo di IRP dobbiamo inserirci nella coda, e questo è decisamente semplice: nella *DriverEntry()* non dobbiamo far altro che inserire un puntatore alla nostra routine di dispatch per ogni tipo di richiesta di I/O che desideriamo gestire. Per comunicare tramite IOCTL dovremo intercettare le richieste di tipo *IRP_MJ_DEVICE_CONTROL* che vengono inviate quando il driver riceve un I/O Control. L'inserimento dell'hook nella coda si traduce in un semplicissimo:

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
DispatchDeviceControl;
```

Dove *DispatchDeviceControl* è il nome della funzione che verrà invocata quando il nostro driver riceverà un IRP di tipo *IRP_MJ_DEVICE_CONTROL*. Per dirla meglio: **tutti** gli IRP di quel tipo verranno processati dalla *DispatchDeviceControl*, quindi con una sola funzione saremo in grado di gestire tutti gli I/O Controls, ed ognuno potrà eseguire un'operazione differente. A questo punto dobbiamo creare i nostri *IOCTL*, in verità ne definiremo soltanto uno, perché per tutti gli altri il procedimento è identico. Uno IOCTL è un codice numerico che trasporta al suo interno alcune informazioni per i driver che lo gestiranno, non possiamo quindi definire un numero a caso, ad esempio:

```
// Questo NON va bene
#define NOSTRO_IOCTL 0x1122344
```

Ma dobbiamo utilizzare una comodissima macro, la *CTL_CODE()* che prende questi parametri:

- **DeviceType**: deve essere uguale al membro *DeviceType* della *DEVICE_OBJECT* associata al nostro driver. Tutti i valori da 0x0000 a 0x7FFF sono riservati per la Microsoft, gli altri li possiamo usare come vogliamo.
- **FunctionCode**: identifica la funzione che deve essere eseguita dal driver. Tutti i valori da 0x0000 a 0x07FF sono riservati per la Microsoft, gli altri li possiamo usare come vogliamo.
- **TransferType**: indica la modalità che il sistema utilizzerà per passari i dati dal chiamante al driver.
- **RequiredAccess**: indica il tipo di accesso che il chiamante deve richiedere quando apre il device. Se il chiamante non possiede i diritti richiesti l'I/O Manager non invia l'IRP al driver.

Come primo parametro utilizzeremo *FILE_DEVICE_UNKNOWN* che equivale a specificare un device generico, in questo caso potremmo anche utilizzare un valore qualunque superiore a 0x7FFF, in entrambi i casi ci identificheremo come device generico. Il secondo parametro specificherà l'identificativo del

nostro control code, possiamo specificare un qualunque numero superiore a 0x07FF, diciamo: 0x801. Il terzo parametro è la modalità di trasferimento dati che vogliamo utilizzare, nel nostro caso sarà *METHOD_BUFFERED*. E come quarto parametro specificheremo *FILE_ANY_ACCESS*, di modo che chiunque possa utilizzare il device object, il nostro IOCTL sarà dunque:

```
#define TEST_IOCTL CTL_CODE(FILE_DEVICE_UNKNOWN, 0x801,
METHOD_BUFFERED, FILE_ANY_ACCESS)
```

Fatto cio', prima di implementare la routine per la gestione dello IOCTL, facciamo una precisazione: abbiamo scelto la modalità *buffered*, in questo caso avremo accesso ad un buffer per la lettura/scrittura di dati che sarà puntato dal membro *AssociatedIrp.SystemBuffer* della struttura IRP. Per i dati in input la dimensione del buffer è specificata nel membro *Parameters.DeviceIoControl.InputBufferLength* della IO_STACK_LOCATION, mentre la dimensione del buffer di output (che non useremo) è specificata nel membro *Parameters.DeviceIoControl.OutputBufferLength*. Detto cio', passiamo al coding :).

DispatchDeviceControl()

Questa funzione gestirà solo le richieste di tipo IRP_MJ_DEVICE_CONTROL, quindi avrà bisogno di un puntatore ad una struttura di tipo IRP e poi un puntatore ad un *DeviceObject*. La funzione controllerà l'IRP, otterrà un puntatore alla *IO_STACK_LOCATION* e poi stamperà all'interno del debugger un numero che gli abbiamo inviato dall'userspace, ma diamoci dentro col codice:

```
NTSTATUS DispatchDeviceControl(PDEVICE_OBJECT pDeviceObject, PIRP
pIrp)
{
        PIO_STACK_LOCATION pIoStack;
        PUCHAR pSystemBuffer;
        NTSTATUS ntStatus = STATUS_SUCCESS;
        unsigned int uSystemBufferLen, uUserParam;

        pIoStack = IoGetCurrentIrpStackLocation(pIrp);

        pSystemBuffer = pIrp->AssociatedIrp.SystemBuffer;
        uSystemBufferLen = pIoStack-
>Parameters.DeviceIoControl.InputBufferLength;

        switch(pIoStack->Parameters.DeviceIoControl.IoControlCode) {
                case TEST_IOCTL:
                        if(uSystemBufferLen < sizeof(unsigned int)){
                                ntStatus = STATUS_SUCCESS;
                                break;
                        }

                        uUserParam = *(unsigned int
*)&pSystemBuffer[0];

                        DbgPrint("Que - Parameter received from
userspace: %d\n", uUserParam);
                        ntStatus = STATUS_SUCCESS;
                        break;

                default:
                        DbgPrint("Que - Unknown control code received:
0x%08x\n", pIoStack->Parameters.DeviceIoControl.IoControlCode);
                        ntStatus = STATUS_SUCCESS;
```

```
                        break;
        }

        if(ntStatus != STATUS_PENDING) {
                pIrp->IoStatus.Status = ntStatus;
                pIrp->IoStatus.Information = 0;
                IoCompleteRequest(pIrp, IO_NO_INCREMENT);
        }

        return ntStatus;
}
```

La prima cosa che facciamo è ottenere un puntatore allo stack dell'IRP tramite la seguente chiamata:

```
pIoStack = IoGetCurrentIrpStackLocation(pIrp);
```

Useremo questo stack per leggere sia il control code (IOCTL) inviato dall'userspace che la dimensione del buffer in cui sono contenuti i dati. Le successive due righe, vale a dire:

```
pSystemBuffer = pIrp->AssociatedIrp.SystemBuffer;
uSystemBufferLen = pIoStack-
>Parameters.DeviceIoControl.InputBufferLength;
```

servono per ottenere un puntatore al system buffer che contiene i dati inviatici dall'userspace e la lunghezza di questo stesso buffer. Fatto cio' possiamo controllare il control code ricevuto, così vedremo se siamo in grado di gestirlo:

```
switch(pIoStack->Parameters.DeviceIoControl.IoControlCode) {
        case TEST_IOCTL:
                ...
        default:
                ...
}
```

per effettuare il controllo è sufficiente uno switch sul parametro *Parameters.DeviceIoControl.IoControlCode*. Sarà nostra premura gestire gli IOCTL che conosciamo e passare la palla se non conosciamo quel determinato control code. Il codice del primo *case* è decisamente autoesplicativo visto che copia semplicemente i primi 4 byte del system buffer dentro un *unsigned int*. Traliasciamo la spiegazione del *default* case dello switch visto che mi sembra ovvia, e passiamo alla parte finale della routine che completa il percorso dell'IRP:

```
if(ntStatus != STATUS_PENDING) {
        pIrp->IoStatus.Status = ntStatus;
        pIrp->IoStatus.Information = 0;
        IoCompleteRequest(pIrp, IO_NO_INCREMENT);
}
```

In questo pezzetto di codice semplicemente richiamiamo la routine di completamento dell'IRP soltanto se il suo stato non è pending. Non possiamo richiamare la *IoCompleteRequest()* su un IRP marcato come pending, altrimenti apparirebbe un bello schermo blu, in quel caso è necessario richiamare la *IoMarkIrpPending()* per segnalare agli altri driver della catena che qualche componente ancora non ha terminato la propria *IoCompletionRoutine*, un'eventualità che comunque non può presentarsi sul nostro driver, ecco perché non c'è il codice per marcare l'IRP. La routine di gestione del control code è completa, diamo ora uno sguardo alla *DriverEntry()* perché è necessario aggiungere alcuni hook.

Major Functions

Per dire al driver che quando riceve una richiesta tramite *DeviceIoControl()* dall'userspace deve chiamare la nostra *DispatchDeviceControl()* è necessario installare un "hook" nell'array delle major functions. Come abbiamo visto prima si tratta di un array che contiene una serie di puntatori a funzione da richiamare per ogni determinato IRP, nel nostro caso dovremo inizializzare due puntatori:

- **IRP_MJ_CREATE**: viene richiamata quando apriamo il device con *CreateFile()*
- **IRP_MJ_DEVICE_CONTROL**: viene richiamata quando utilizziamo la *DeviceIoControl()*

La prima major function possiamo inizializzarla con una funzione *dummy* che non fa nulla, se non la inizializziamo non riusciremo ad aprire il device con la *CreateFile()*. La seconda major function è quella che dirotta l'esecuzione sulla chiamata vista poco sopra, in termini di codice significa semplicemente fare questo:

```
DriverObject->MajorFunction[IRP_MJ_CREATE] = Dummy;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
DispatchDeviceControl;
```

Il codice finale del driver, se non l'avete già scaricato, potete prenderlo qui.

DeviceIoControl()

Siamo praticamente arrivati alla fine, dopo aver compilato e caricato il driver dobbiamo semplimente dialogare con lui tramite... Cosa? In verità possiamo usare moltissimi modi per parlare con il driver, anche perché trovandosi a livello kernel lo stesso driver può intercettare qualunque tipo di request, ma nel nostro caso sarà sufficiente effettuare una chiamata a *DeviceIoControl()*:

```
BOOL WINAPI DeviceIoControl(
  __in          HANDLE hDevice,
  __in          DWORD dwIoControlCode,
  __in          LPVOID lpInBuffer,
  __in          DWORD nInBufferSize,
  __out         LPVOID lpOutBuffer,
  __in          DWORD nOutBufferSize,
  __out         LPDWORD lpBytesReturned,
  __in          LPOVERLAPPED lpOverlapped
);
```

Come al solito esaminiamoli tutti, non preoccupatevi che sono davvero immediati:

- *hDevice*: l'handle al device.
- *dwIoControlCode*: lo IOCTL.
- *lpInBuffer*: un puntatore al buffer che CONTIENE i dati da passare AL driver.
- *nInBufferSize*: la lunghezza del buffer precedente.
- *lpOutBuffer*: un puntatore al buffer che RICEVE i dati inviati DAL driver.
- *nOutBufferSize*: la lunghezza del buffer predecente.
- *lpBytesReturned*: un puntatore ad una variabile che conterrà il numero di byte ricevuti.
- *lpOverlapped*: un eventuale puntatore ad una struttura *OVERLAPPED*, usata per le operazioni asincrone.

I parametri sono abbastanza chiari, forse vi starete chiedendo: come otteniamo l'handle al device? In effetti abbiamo creato un device (tramite il driver) che abbiamo chiamato *\\DosDevices\\Que*, per aprirlo è sufficiente richiamare la *CreateFile()* in questo modo:

```
CreateFile("\\\\.\\Que", GENERIC_WRITE, FILE_SHARE_WRITE, NULL,
OPEN_EXISTING, NULL, NULL);
```

Memorizzate bene la forma di apertura del device perché sicuramente vi tornerà utile in futuro. L'ultimo parametro forse è un po' meno ovvio, ma possiamo settarlo a NULL visto che per il momento non siamo interessati ad operazioni asincrone. Il sorgente (con i vari check rimossi per ridurre lo spazio) del codice ad userspace è il seguente:

```
#define TEST_IOCTL CTL_CODE(FILE_DEVICE_UNKNOWN, 0x801,
METHOD_IN_DIRECT, FILE_ANY_ACCESS)

int main(int argc, char *argv[])
{
        HANDLE hFile = INVALID_HANDLE_VALUE;
        DWORD dwReturned;
        UINT value;

        value = atoi(argv[1]);
        hFile = CreateFileW(L"\\\\.\\Que", GENERIC_WRITE,
FILE_SHARE_WRITE, NULL, OPEN_EXISTING, NULL, NULL);
        DeviceIoControl(hFile, TEST_IOCTL, (LPVOID)&value,
sizeof(INT), NULL, 0, &dwReturned, NULL);
        CloseHandle(hFile))
        return 0;
}
```

Il pacchetto completo lo trovate invece qui. Il funzionamento ancora una volta è molto semplice, apriamo il device e tramite la *DeviceIoControl()* inviamo il nostro *TEST_IOCTL* ed un buffer di 4 byte che contiene un numero scelto dall'utente. Una volta compilato anche l'eseguibile dovremo soltanto avviarlo da console:

interface.exe 111

E verificare il messaggio stampato dal driver su *DbgView*... Tutto qui :). Buon divertimento a tutti e alla prossima!!!

Quequero

Note Finali

Abbiamo imparato a creare un driver e a gestire qualche IRP di base, le idee che possono partire da qui sono moltissime e c'e' molto con cui divertirsi se la programmazione a kernel space vi dovesse interessare. Ci vediamo al prossimo tutorial e nel frattempo divertitevi a creare il vostro driver e a dirottare/intercettare qualche IRP per fare qualcosa di divertente :).

Un salutone a tutto lo staff **UIC**, andreone, spark, rey, vinx, tutto il forum e ad olografix che ci supporta!!! E come al solito contattatemi se ci sono errori/orrori/inesattezze etc...

Disclaimer

I documenti qui pubblicati sono da considerarsi pubblici e liberamente distribuibili, a patto che se ne citi la fonte di provenienza. Tutti i documenti presenti su queste pagine sono stati scritti esclusivamente a scopo di ricerca, nessuna di queste analisi è stata fatta per fini commerciali, o dietro alcun tipo di compenso. I documenti pubblicati presentano delle analisi puramente teoriche della struttura di un programma, in nessun caso il software è stato realmente disassemblato o modificato; ogni corrispondenza presente tra i documenti pubblicati e le istruzioni del software oggetto dell'analisi, è da ritenersi puramente casuale. Tutti i documenti vengono inviati in forma anonima ed automaticamente pubblicati, i diritti di tali opere appartengono esclusivamente al firmatario del documento (se presente), in nessun caso il gestore di questo sito, o del server su cui risiede, può essere ritenuto responsabile dei contenuti qui presenti, oltretutto il gestore del sito non è in grado di risalire all'identità del mittente dei documenti. Tutti i documenti ed i file di questo sito non presentano alcun tipo di garanzia, pertanto ne è sconsigliata a tutti la lettura o l'esecuzione, lo staff non si assume alcuna responsabilità per quanto riguarda l'uso improprio di tali documenti e/o file, è doveroso aggiungere che ogni riferimento a fatti cose o persone è da considerarsi PURAMENTE casuale. Tutti coloro che potrebbero ritenersi moralmente offesi dai contenuti di queste pagine, sono tenuti ad uscire immediatamente da questo sito.

Vogliamo inoltre ricordare che il Reverse Engineering è uno strumento tecnologico di grande potenza ed importanza, senza di esso non sarebbe possibile creare antivirus, scoprire funzioni malevoli e non dichiarate all'interno di un programma di pubblico utilizzo. Non sarebbe possibile scoprire, in assenza di un sistema sicuro per il controllo dell'integrità, se il "tal" programma è realmente quello che l'utente ha scelto di installare ed eseguire, né sarebbe possibile continuare lo sviluppo di quei programmi (o l'utilizzo di quelle periferiche) ritenuti obsoleti e non più supportati dalle fonti ufficiali.