The Dark Side of Winsock

By Jonathan Levin

ReCON 2005, Montreal

Introduction & Nomenclature

You probably already know this but…

IP communications are implemented using the socket API.

A socket is a transport endpoint, used to send/receive data.

The application reads from/writes to the socket, much as it would to any other file descriptor

The OS transparently fragments/encapsulates the data.

This talk assumes you've seen sockets in action before. Be it in Stevens' legendary tomes (TCP/IP Illustrated, UNIX Network Programming..) or elsewhere.

(more) Introduction & Nomenclature

You probably already know this too, but…

In UNIX, sockets follow the Berkeley (BSD) model closely

Windows adapted the BSD socket API into WinSock:

Winsock 1.x was a close adaptation of the BSD API
Winsock 2.x added new features

- Asynchronous calls & callbacks
- Overlapped I/O
- The layered service provider (LSP) architecture

(more) Introduction & Nomenclature

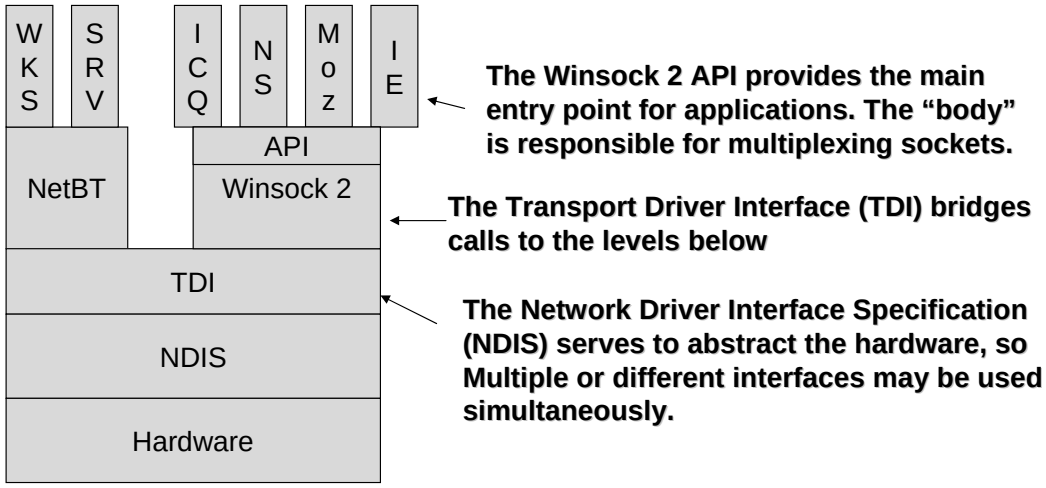
However, not too many people know that…

Winsock's Layered Service Provider architecture provides powerful hooking functionality enabling interception, eavesdropping or rerouting of almost all IP based traffic in windows platforms.

This talk will focus on the LSP, presenting it's useful (legitimate) applications, and even more useful (but less legitimate) ones.

Winsock 2 Architecture

Windows is designed in a scalable, multi-layered architecture:



NetBT (The NetBIOS over TCP/IP interface) is "reserved", and is used by windows' Workstation and Server services (file and print sharing) to bypass "traditional" winsock calls (and is out of our scope anyway).

All other (user mode) applications use winsock to communicate over the network.

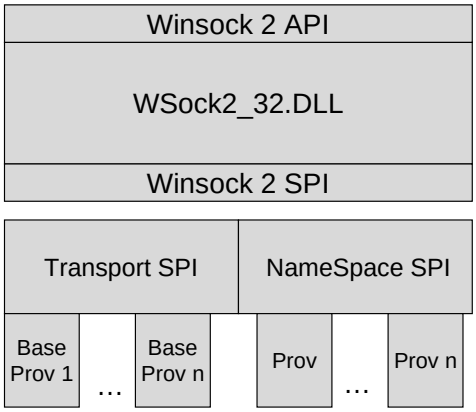
Winsock SPI

Winsock 2 Architecture

While exporting the API, Winsock itself is a client of the SPI , or service provider interface, exported to it by the miscellaneous service providers installed below it.

Providers may be classified as either:

-TRANSPORT
- NAMESPACE



The Winsock DLL itself serves as a multiplexer for two types of providers:

- **Transport Providers**: Protocol stacks, that setup connections, and transfer data on the network, possibly supplying features such as QoS, error handling, etc.

Windows 2000 ships with two transports:

rsvpsp.dll – implementing RSVP QoS

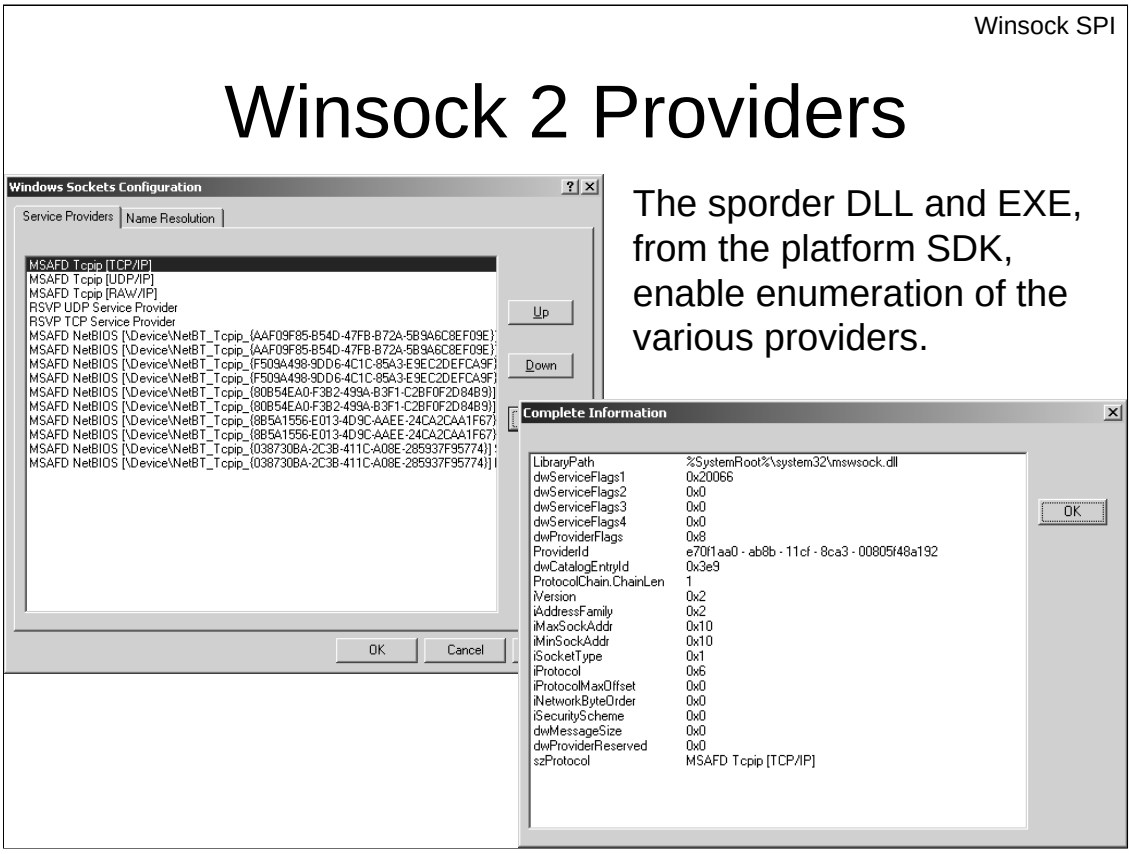
mswsock.dll – implementing the Winsock core.

The provider is chosen upon socket creation, by the parameters to the Socket() (or WSASocket()) call.

- **NameSpace Providers**: Naming services – suppliers of name resolution mechanisms (e.g. implementations of getXXXbyYYY functions).

Winsock 2000 supports the TCP/IP, NT DS and NLA namespaces.

There can be more than one provider of any type. Winsock accesses the providers by their interface, which is the Service Provider Interface, or **SPI**.

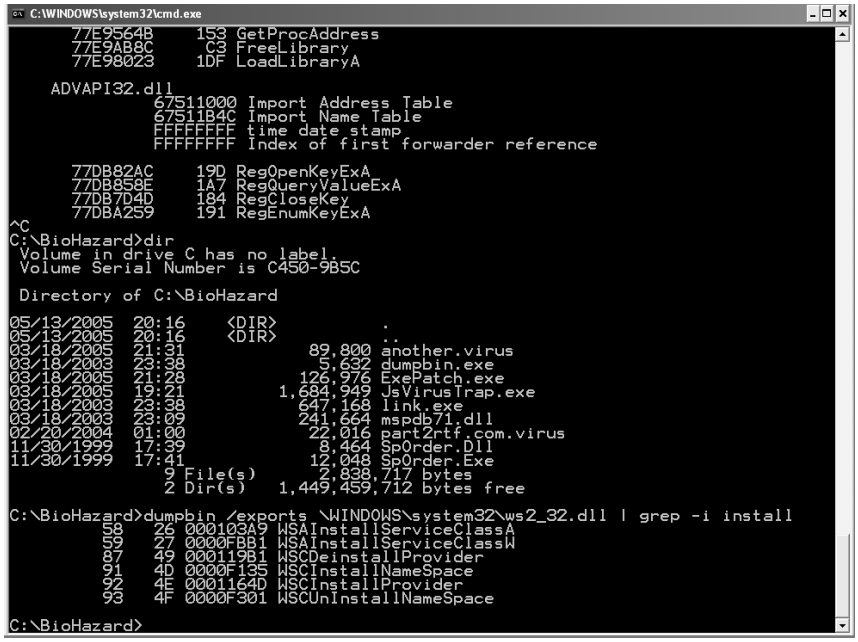


The above is a screen shot of the “SPOrder.EXE”, provided as part of the platform SDK. This small utility displays the service providers registered under winsock. Note both classes – “Service Providers” (i.e. Transport Service Providers) and “Name Resolution” (Namespace Service Providers).

Note each provider structure is quite detailed. The one shown here is for the AF_INET (0x02) address family protocol # 0x06 – better known as TCP.

Winsock SPI

Export Goods (ws2_32.dll)



Winsock provides a potent API for installing custom providers, both namespace and transport. In ws2spi.h:

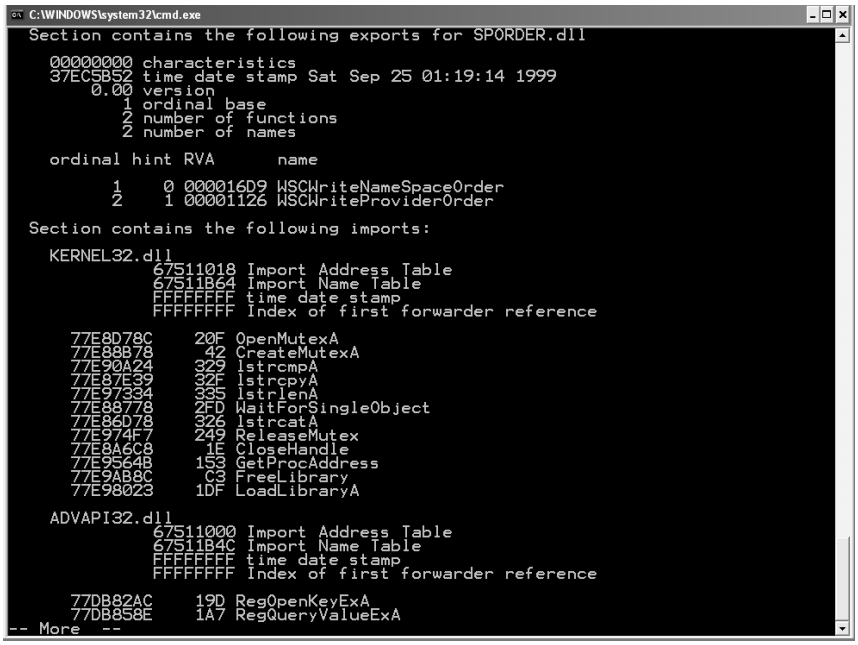
```
int WSPAPI WSCInstallProvider(
    IN LPGUID lpProviderId,
    IN const WCHAR FAR * lpszProviderDllPath,
    IN const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
    IN DWORD dwNumberOfEntries,
    OUT LPINT lpErrno
    );

int WSPAPI WSCDeinstallProvider(
    IN LPGUID lpProviderId,
    OUT LPINT lpErrno
    );
```

And the namespace ones:

```
INT WSPAPI WSCInstallNameSpace (
    IN LPWSTR lpszIdentifier,
    IN LPWSTR lpszPathName,
    IN DWORD dwNameSpace,
    IN DWORD dwVersion,
    IN LPGUID lpProviderId
    );


INT WSPAPI WSCUnInstallNameSpace (
    IN LPGUID lpProviderId
    );
```

The different header definitions (int vs. INT, and “Deinstall” vs. “Uninstall”) are like that in the original ws2spi.h.

The chaos of Sporder.dll



SPOrder.dll is a small DLL with insidious capabilities – it allows the reordering of service providers, by exporting two functions: ***WSCWriteNameSpaceOrder***, and ***WSCWriteProviderOrder***. And, as one can deduce by the names – these rewrite the order of the layered service providers – namespace and transport, respectively. A further look at the import table sheds some light as to how that's done – using the familiar ADVAPI32.DLL registry functions.

One needn't look hard to understand how to use these functions - These functions are part of the Platform SDK, and are defined in sporder.h:

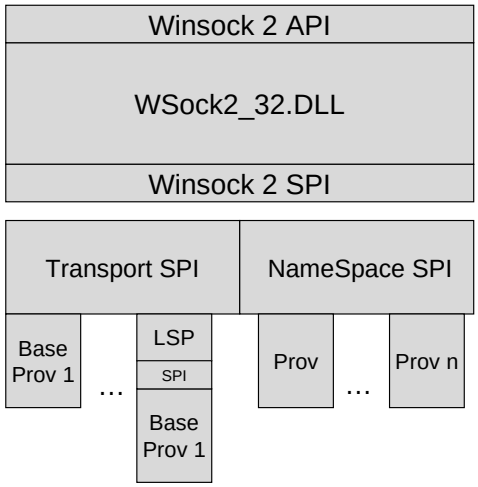
```
int
WSPAPI
WSCWriteProviderOrder (
    IN LPDWORD lpwdCatalogEntryId,
    IN DWORD dwNumberOfEntries
    );

int
WSPAPI
WSCWriteNameSpaceOrder (
    IN LPGUID lpProviderId,
    IN DWORD dwNumberOfEntries
    );
```

Winsock 2 Providers

Winsock providers may be enhanced by LAYERING additional providers, and chaining them.

The base service provider still handles the actual implementation (i.e. sending data, etc.) but layered SPs may be used for QoS, encryption, security, etc.

So long as all providers in a chain support SPI , any number of providers may be chained.



Winsock SPI

Winsock 2 Providers

Enumerating providers

ws2spi.h

```
int WSPAPI WSCEnumProtocols (
        IN      LPINT                lpiProtocols,
        OUT     LPWSAPROTOCOL_INFOW  lpProtocolBuffer,
        IN OUT  LPDWORD              lpdwBufferLength,
        OUT     LPINT                lpErrno
);
```

Usage: Retrieve information about available transport protocols.

Parameters:

lpiProtocols – NULL term. Array of iProtocols to enum, or NULL.
lpProtocolBuffer – buffer of WSAPROTOCOL_INFOW structs
lpdwBufferLength – in/out parameter specifying sizeof..
lpErrNo – Out parameter, holding error code, if any.

Returns: Number of enumerated protocols.

The following example demonstrates enumeration of the layered service providers, and the WSAPROTOCOL_INFOW structs. Essentially, this is a CLI version of sporder.exe from the platform SDK.

Note: it gets the job done. It's not an example of pretty or "right" coding.

```
/**
 * Winsock 2 API Protocol Enumerator -  By JL@HisOwn.com
 * (Standards disclaimers apply)
 */

#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

#define WINSOCK_API_LINKAGE

#include <winsock2.h>
#include <ws2spi.h>
#include <wtypes.h>
#include <assert.h>
#include <winnt.h>
#include <stdlib.h>
#include <stdio.h>
```

```
char *ExpandServiceFlags(DWORD serviceFlags)
{
  /* A little utility function to make sense of all those bit flags */
  /* The following code leaks. Yeah, I know.. Go find Buffer 0v3rfl0w$ :-) */

  char *serviceFlagsText = (char *) malloc (2048);
  memset (serviceFlagsText, '\0', 2048);
  char *strip_comma;
   /* Hey - it's only for printing and demo purposes.. */
  if (serviceFlags & XP1_CONNECTIONLESS)
   {
     strcat (serviceFlagsText, "Connectionless, ");
   }
  if (serviceFlags & XP1_GUARANTEED_ORDER)
   {
     strcat (serviceFlagsText, "Guaranteed Order, ");
   }
  if (serviceFlags & XP1_GUARANTEED_DELIVERY)
   {
     strcat (serviceFlagsText, "Message Oriented, ");
   }
  if (serviceFlags & XP1_MESSAGE_ORIENTED)
   {
     strcat (serviceFlagsText, "Message Oriented, ");
   }
  if (serviceFlags & XP1_CONNECT_DATA )
   {
     strcat (serviceFlagsText, "Connect Data, ");
   }
  if (serviceFlags & XP1_DISCONNECT_DATA )
   {
     strcat (serviceFlagsText, "Disconnect Data, ");
   }
  if (serviceFlags & XP1_SUPPORT_BROADCAST )
   {
     strcat (serviceFlagsText, "Broadcast Supported, ");
   }
  if (serviceFlags & XP1_EXPEDITED_DATA )
   {
     strcat (serviceFlagsText, "Urgent Data, ");
   }
  if (serviceFlags & XP1_QOS_SUPPORTED )
   {
     strcat (serviceFlagsText, "QoS supported, ");
   }
 /*
  * While we're quick and dirty, let's get as dirty as possible..
  */
  strip_comma = strrchr(serviceFlagsText,',');

  if (strip_comma)
     *strip_comma = '\0';

  return (serviceFlagsText);
}
```

```
void PrintProtocolInfo (LPWSAPROTOCOL_INFOW prot)
{

    wprintf (L"Protocol Name: %s\n",prot->szProtocol); /* #%^@$! UNICODE...*/
    printf ("\tServiceFlags1:  %d (%s)\n",
              prot->dwServiceFlags1,
              ExpandServiceFlags(prot->dwServiceFlags1));
    printf ("\tProvider Flags: %d\n",prot->dwProviderFlags);
    printf ("\tNetwork Byte Order: %s\n",
    (prot->iNetworkByteOrder == BIGENDIAN) ? "Big Endian" : "Little Endian");
    printf ("\tVersion: %d\n", prot->iVersion);
    printf ("\tAddress Family: %d\n", prot->iAddressFamily);
    printf ("\tSocket Type: ");
    switch (prot->iSocketType)
    {
      case SOCK_STREAM:
         printf ("STREAM\n");
         break;
      case SOCK_DGRAM:
         printf ("DGRAM\n");
         break;
      case SOCK_RAW:
         printf ("RAW\n");
         break;
      default:
          printf (" Some other type\n");
   }
    printf ("\tProtocol: ");

    switch (prot->iProtocol)
    {
      case IPPROTO_TCP:
         printf ("TCP/IP\n");
         break;
      case IPPROTO_UDP:
         printf ("UDP/IP\n");
         break;
      default:
          printf ("some other protocol\n");

    }
}
```

And finally, the main:

```
int _cdecl main( int argc, char** argv)
{
    LPWSAPROTOCOL_INFOW  bufProtocolInfo = NULL;
    DWORD                dwSize = 0;
    INT                  dwError;
    INT                  iNumProt;

    /*
     * Enum Protocols - First, obtain size required
     */

     printf("Sample program to enumerate Protocols\n");
     WSCEnumProtocols(NULL,                     // lpiProtocols
                      bufProtocolInfo,          // lpProtocolBuffer
                      & dwSize,                 // lpdwBufferLength
                      & dwError);               // lpErrno

     bufProtocolInfo = (LPWSAPROTOCOL_INFOW) malloc(dwSize);

     if (!bufProtocolInfo){
         fprintf (stderr,"SHOOT! Can't MALLOC!!\n");
         exit(1);
     }

     /* Now, Enum */
     iNumProt = WSCEnumProtocols(
                    NULL,                   // lpiProtocols
                    bufProtocolInfo,        // lpProtocolBuffer
                    &dwSize,                // lpdwBufferLength
                    &dwError);

     if (SOCKET_ERROR == iNumProt)
     {
           fprintf(stderr,"Darn! Can't Enum!!\n");
           exit(1);
     }

     printf("%d Protocols detected:\n", iNumProt);
     for (int i=0;
          i < iNumProt;
          i++)
     {
        PrintProtocolInfo(&bufProtocolInfo[i]);
        printf ("-------\n");
      }

      printf("Done");
        return(0);
}
```

Winsock SPI

Winsock 2 Providers

So you want to build your own LSP?

Start by implementing WSPStartup()

```
int WSPStartup (IN  WORD              wVersionRequested,
                OUT LPWSPDATAW        lpWSPData,
                IN  LPWSAPROTOCOL_INFOW lpProtocolInfo,
                IN WSPUPCALLTABLE      UpcallTable,
                OUT LPWSPPROC_TABLE     lpProcTable );
```

Usage: Initialize layered service provider

Parameters:

wVersionRequested – q.v. WSAStartup.
lpWSPData – layered service provider data, you should populate
lpProtocolInfo – protocol details. Useful if hooking multiple protocols
UpcallTable – dispatch table for winsock calls
lpProcTable - our implemented calls.

Returns: No error, hopefully..

Implementing a Layered Service Provider isn't as hard as it might seem. Basically, all it takes is to adhere to a set API, and manipulate some function pointers. Winsock Layered service providers are implemented as standard DLLs, exporting the WSPStartup() function:

The WSPStartup() is expected to:

- Set the Version info:

  (i.e. lpWSPData->wVersion = MAKEWORD(2,2);
  lpWSPData->wHighVersion = MAKEWORD(2,2);
  wcscpy(lpWSPData->szProtocol, L"My Name");)

- Save the UpCallTable: for future use

- Populate the lpProcTable to the addresses of the local WSP functions

  (e.g. - *lpProcTable->lpWSPAccept = WSPAccept;*
  lpProcTable->lpWSPConnect = WSPConnect;
  lpProcTable->lpWSPSend = WSPSend; …)

- Return NO_ERROR

Winsock SPI

Winsock 2 Providers

API->SPI Mapping

Most Winsock2 API functions are mapped to corresponding SPI functions, with the simple rule of WSA* → WSP*.

Once a WSA* function is called, Winsock 2 will call the corresponding WSP function, from the provider chain, in order.

ALL functionality can be hijacked – getpeerbyname, setsockopt.. AddresstoString, etc.

Call Upcall table function to enable passthrough.

Functions NOT implemented in the SPI:

Event Handling Functions:

WSACreateEvent,
WSACloseEvent,
WSASetEvent,
WSAResetEvent
WSAWaitForMultipleEvents

Naming Services functions:

GetXXXByYYY and their WSAAsync counterparts.
ntohs, ntohl, htonl, htons
inet_XtoY, inet_addr, ...

As well as:

WSAEnumProtocols – Enumerating service providers
WSAIsBlocking,
WSASetBlockingHook,
WSAUnhookBlockingHook

Winsock 2 Providers

Installing Providers

Finally, call our old friend, WSCInstallProvider():

ws2spi.h

```
int WSPAPI WSCInstallProvider(
    IN LPGUID lpProviderId,
    IN const WCHAR FAR * lpszProviderDllPath,
    IN const LPWSAPROTOCOL_INFOW lpProtocolInfoList,
    IN DWORD dwNumberOfEntries,
    OUT LPINT lpErrno
    );
```

Reorder using WSCWriteProviderOrder()

Finally, when your service provider is done, install it by an external .exe, like so:

```
INT InstallProvider(OUT PDWORD CatalogId)
{
    WSAPROTOCOL_INFOW  proto_info;
    int               rc, errno;

   GUID someGUID = { 0x10241975,  0x0000, 0x0000, 0x0000,  0x1234567890 };

    /* populate PROTOCOL_INFO */
    memset(&proto_info , '\0', sizeof(proto_info)); /* Tabula Rasa */
    proto_info.dwProviderFlags = PFL_HIDDEN; /* :-) */
    proto_info.ProviderId      = someGUID;
    proto_info.ProtocolChain.ChainLen = LAYERED_PROTOCOL;
    proto_info.iAddressFamily = AF_INET;
    proto_info.iSocketType = SOCK_STREAM;
    proto_info.iProtocol = IPPROTO_TCP;
    proto_info.iMaxSockAddr = proto_info.iMinSockAddr = 16;
    proto_info.iNetworkByteOrder = BIGENDIAN;
    proto_info.iSecurityScheme=SECURITY_PROTOCOL_NONE; /* Security? THIS?! HA! */
    wcscpy(proto_info.szProtocol, L"Incognito");
    rc = WSCInstallProvider(&LayeredProviderGuid,
                            L"trojan.dll", // lpszProviderDllPath
                            &proto_info,   // lpProtocolInfoList
                            1,             // dwNumberOfEntries (1 too many..)
                            &errno);       // lpErrno
    /* Pass this back to our caller – for reordering.. */
    *CatalogId = proto_info.dwCatalogEntryId;
    return(rc);
}
```

Winsock 2 SPI

Demo

The demo shown is a nearly unmodified version of the INTC/MSFT source code provided in the platform SDK.

Winsock 2 SPI

The Security Issue

Lessons to be learned:

No matter how you code your application – if you use Winsock, you're subject to socket hijacking.

Whether you use server or client sockets, an attacker can intercept your calls and redirect your connections to where ever he (or she) pleases.

Winsock 2 SPI

Good Vs. Bad

Possible (lawful goody-goody) uses include:

- Implement a user-mode application layer firewall
(rather than work at TDI/NDIS, be socket-aware)

- transparently add encryption to applications
(but then, there's IPSec)

- Enforce QoS
(s/Q/D)

- Support new protocols
(IPv8, anyone?)

Winsock 2 SPI

Let's just stick with the BAD

But the MUCH better (chaotic evil) uses include:

- **Intercepting name service provider calls**
  (for spyware, statistical purposes, etc..)
- **Eavesdropping (non SSL) connections**
  (all socket based communication (inc. raw))
- **Rerouting connections (i.e. socket hijacking)**

The Dark Side has never been so tempting before..

The End…

(or perhaps, the beginning?)

Questions/Comments Welcome:
SPI@HisOwn.Com