IBSec: Securing InterBase Network Traffic On The Fly

by Jani Järvinen

After the inclusion of the InterBase Express components in Delphi 5, InterBase has gained popularity in all kinds of database applications. Personally, I like to use InterBase because I find it fast, reliable and compact. Furthermore, it is a pure SQL database and can thus be operated with the same principles as products like Oracle and Microsoft SQL Server.

One problem with InterBase, however, is that all the network communication between the InterBase server and the client application is unencrypted (ie, it is plaintext). This means that someone with a network monitor could easily see the data the application and the server exchange.

Of course, this is not a problem when InterBase is operated locally, or in physically secured intranets, but in the public internet and in extranet systems the network transmissions must be encrypted. Because InterBase does not have an option to enable encryption of transmissions, we have to develop the required security features ourselves.

When developing these kinds of solutions, it is best to design them so that existing applications are not broken. As you might guess, this is more easily said than done, but in my opinion this is the only reasonable way to do it.

In this article, I will represent an application named IBSec, short for InterBase Security. It allows you to secure your InterBase TCP/IP network traffic without breaking existing applications. Before looking at the code, let's take a look at the design of IBSec.

Securing InterBase Traffic

To transparently secure InterBase network traffic, we must somehow 'capture' the normal plaintext InterBase network traffic, transmit it in encrypted form over the internet, then decrypt the data at the other end. Finally, the unencrypted data can be sent to the InterBase server.

Not counting the 'local' connection method, InterBase supports three communication protocols: NetBEUI, IPX/SPX and TCP/IP. When an InterBase client application, say InterBase Interactive SQL (WISQL), wants to connect to an InterBase server using TCP/IP, it opens up a TCP connection to a specific IP address and port.

By default, the client application assumes that the responding application is the InterBase server itself. However, it is easy to create a TCP server application that responds to connections on a certain port. If our application can transparently transfer all the bytes between the client application and the InterBase server, the client application will still think it is talking directly to an InterBase server.

Because our application sits between the client and the server, it can do anything it likes with the bytes transferred. Of course, in this case, our application will encrypt the data, but it could also compress or cache the data.

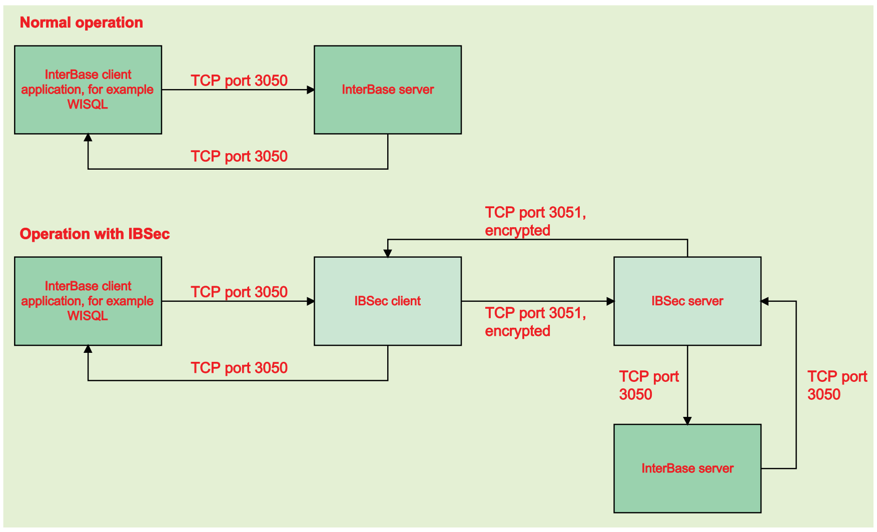
To illustrate this concept, I've drawn Figure 1. At the top, you can see how communication between a client application and the InterBase server normally occurs. Below that, you can see how IBSec fits into the view. On the left, the InterBase client application talks to our IBSec client, which in turn talks to the IBSec server. The IBSec server finally talks to the InterBase server. Only the communication between the IBSec applications is secured.

Socket Implementation

To keep IBSec simple, I've restricted IBSec to support only TCP/IP communications. Because of this, IBSec only needs to use Delphi's `TClientSocket` and `TServerSocket` components, which wrap up the WinSock APIs into easy-to-use entities. The `TClientSocket` component is used to connect to a TCP server and the `TServerSocket` is used to open up a TCP port for client connections.

In IBSec client, an instance of a `TServerSocket` is created at startup and set to listen to connections at the user's request. By default, InterBase client applications expect to communicate with the InterBase server at port 3050. To keep IBSec transparent to the

➤ *Figure 1: How IBSec fits in.*



client application, the TServerSocket is set to listen to this port.

Because it is normal practice to run IBSec server on the same computer as the InterBase server, we need a second port for communication between the two IBSec applications. I've chosen to use the port 3051, which is free on most Windows machines. If this is not the case with your setup, feel free to change this port by editing the value on the main form.

As I said, the IBSec server will communicate with the IBSec client at port 3051. Thus, the TServerSocket is set to listen to this port. Note that the IBSec application is written so that the exact same program can function either as a client or as a server. The user sets the operation mode and, when the user clicks the Start button, IBSec sets the server socket to listen to either port 3050 or 3051.

Listening to TCP connections as described previously is only half of the socket implementation, however. When IBSec client accepts a connection from an InterBase client application, it instantiates a client socket which is used to communicate with the IBSec server. Similarly, when an IBSec server accepts a connection from the IBSec client, it creates an instance of the TClientSocket class to communicate with the InterBase server.

Sockets And Messages

Delphi's TClientSocket and TServerSocket components can operate in either blocking or non-blocking modes. For simplicity, I've chosen to use the non-blocking mode, which is the default for these components. The idea behind these two modes is that in the non-blocking mode the application doesn't need to wait for data arrival, for instance. Instead, it can continue its normal operation until data actually arrives.

When a connection arrives to a server socket, the event handler for the event OnClientConnect is executed (see Listing 1). The event handler first adds an entry to the IBSec log (a simple memo field) and then instantiates a TClientSocket component. Because both the client and server sockets must be aware of each other, the event handler also associates pointers both ways. Why this must be done becomes evident when you look at the later code listings.

```
procedure TIBSecMainForm.ServerSocketClientConnect(Sender: TObject;
  Socket: TCustomWinSocket);
Var CS : TClientSocket;
begin
  LogMessage('Client '+Socket.RemoteAddress+' connect');
  Assert(Socket.Data = nil);
  CS := TClientSocket.Create(Self);
  CS.Socket.Data := Socket;
  Socket.Data := CS;
  PostMessage(Handle,wm_ConnectSocket,Integer(CS),Integer(Socket));
end;
```

➤ *Listing 1: Handling the OnClientConnect event on the server socket.*

```
BufLen := 16*1024; { 16k }
Buffer := StrAlloc(BufLen);
BufLen := Socket.ReceiveBuf(Buffer^,BufLen);
If CryptData.Checked Then Begin
  If (Operation.ItemIndex = Server) Then
    Decrypt(Buffer,BufLen)
  Else
    Encrypt(Buffer,BufLen);
End;
LogMessage('ServerSocket.ClientRead '+IntToStr(BufLen)+' bytes');
If ((Socket.Data = nil) Or
    (Not TClientSocket(Socket.Data).Socket.Connected)) Then Begin
  { save the buffer in a queue }
  New(DataRec);
  With DataRec^ do Begin
    ServerSocket := Socket;
    DataBuffer := Buffer;
    BufferLen := BufLen;
  End;
  SendQueue.Add(DataRec);
End;
```

➤ *Listing 2: Reading data from the server socket.*

On the last line of the event handler you can see a call to the PostMessage function. This Windows API function posts a message to the application's event queue and, after processing all other messages in the event queue, the application will finally process the message we posted.

The reason that we must use such custom messages is that Delphi's socket components can get confused if we attempt to operate multiple sockets inside socket event handlers. Note that I'm assuming here that you are familiar with Windows message processing and defining your own message handlers in Delphi. If this concept is new to you, please read the sidebar *Defining Custom Message Handlers*. Alternatively you could consult Delphi's help: you can find a good introduction by typing in the phrase 'message handlers'.

Sending And Receiving Data

Now back to the socket implementation. When an InterBase client application first contacts IBSec, it immediately sends several bytes after opening up the TCP connection to IBSec. Although we instantly create a client socket (a 'slave' socket), the client socket will not be immediately ready to be used for communications.

This is a problem that can be solved by using a data queue. That is, instead of sending data read from the server socket immediately to the client socket, we store the data in a queue, and only send the queue when the client socket is ready. The code that handles the queue can be seen in Listing 2, which shows part of the OnClientRead event handler of the server socket.

Although a queue is a good solution to this problem, it is not very effective performance-wise. To keep IBSec's performance at an acceptable level, I've chosen to use the queue only when a client socket is not ready for communications. If it is, I will instead send the data immediately through the client socket.

In Listing 3 you can see part of the code for the `WM_SendQueue` message handler. As I didn't want to use any custom threads in IBSec, I chose to use Windows messages to inform IBSec that there is data waiting in the queue to be sent.

Although not technically necessary, IBSec uses a `TThreadList` class to represent the data queue. `TThreadList` is, as it name implies, a thread-safe list. In its current implementation, all IBSec methods and event handlers get called in the main-thread context, as can be seen from the log entries (the `GetCurrentThreadID` function is used to verify this).

`TThreadList` is not mandatory in the current implementation of IBSec (we could have used `TList` instead), but I wanted to use it just in case you want to use IBSec's code in a multi-threaded application. This way you do not need to rewrite the data queue handling.

Handling Disconnections

When an InterBase client application wishes to disconnect from the IBSec client, we must gracefully disconnect all the slave connections too. Generally, the InterBase client application will commit any pending transactions just before disconnecting, so IBSec must be prepared to handle data transmissions even if disconnection occurs immediately after receiving the data.

When the client disconnects, the IBSec client receives the `OnClientDisconnect` event on the server socket (see Listing 4). Here, the client socket is disconnected by setting its `Active` property to `False`. Note that this does not force us to create a custom message handler to disconnect the socket.

The situation is different, however, when a disconnection occurs on the client socket level (`TClientSocket.OnDisconnect`). In this case we must again use a custom Windows message to disconnect the master connection. Although it is the InterBase client application that most often initiates the disconnection, we must be prepared to handle different cases as well.

Similarly, IBSec must be prepared to handle certain communication errors. For example, the TCP connection might spontaneously abort, maybe because of a hardware error. The server socket's `OnClientError` and the client socket's `OnError` event handlers handle such errors. For example, if a `WSAECONNABORTED` error (defined in WinSock.pas) occurs, IBSec knows to disconnect all the necessary connections.

Keeping A Secret

Up to this point, we have only discussed how IBSec communicates using the sockets. Of course, IBSec wouldn't be IBSec without cryptography. IBSec uses the Microsoft CryptoAPI for encryption and decryption of data.

The Microsoft CryptoAPI (or CAPI for short) was introduced in 1996 and has since been an integral part of Microsoft's operating systems. Windows 95 didn't originally include CAPI, but upgrading Internet Explorer brought CAPI to Windows 95 too. Users of Windows NT 4.0 and Windows 2000 do not need any additional components, as CAPI is installed on these systems by default.

CAPI is an easy-to-use interface that lets the application developer add all kinds of cryptography services to his or her application. For example, CAPI supports digital signatures, authentication, key exchange and, of course, data encryption and decryption.

The problem with CAPI for Delphi developers has been that the C language header files had not been translated into Object Pascal. I've done my own conversion of the header files: the results are in WinCrypt.pas on the disk.

CAPI supports a multitude of cryptographic algorithms, but for IBSec we need one that does not affect the number of bytes transferred. One such algorithm is RC4, which is supported by all CAPI versions. It is a streaming cipher developed by RSA Data Security Inc in 1987 (see the reference at the end of the article for details).

Using CryptoAPI

When IBSec is started, it initialises CryptoAPI. This is called 'acquiring a context', and the code for this

➤ *Listing 3: Handling the thread-safe data buffer queue.*

```
With SendQueue do Begin
  List := LockList;
  Try
    LogMessage('Sending queued buffers: '+IntToStr(List.Count));
    For I := 0 to List.Count-1 do Begin
      DataRec := List[I];
      With DataRec^ do Begin
        Client := ServerSocket.Data;
        If ((Client = nil) Or
            (Not Client.Socket.Connected)) Then
          Retry := True
        Else Begin
          Bytes := Client.Socket.SendBuf(DataBuffer^,BufferLen);
          LogMessage(IntToStr(Bytes)+' bytes sent');
          List[I] := nil;
          StrDispose(DataBuffer);
          Dispose(DataRec);
        End;
      End;
    End;
    List.Pack; { pack the queue }
  Finally
    UnlockList;
  End;
End;
```

➤ *Listing 4: Disconnecting the client.*

```
procedure TIBSecMainForm.ServerSocketClientDisconnect(Sender: TObject;
  Socket: TCustomWinSocket);
begin
  LogMessage('Client '+Socket.RemoteAddress+' disconnect');
  With TClientSocket(Socket.Data) do Begin
    Active := False;
    Free;
  End;
  Socket.Data := nil;  { make sure no dangling pointers exist }
end;
```

operation can be seen in Listing 5. Note that we must be prepared to handle the situation where the initial acquiring fails. This might happen if CAPI has not yet been used by any application, for instance because the operating system has just been installed.

After initialising CAPI, we need to create a key to be used for encryption and decryption. The easiest way to create a key in CAPI is first to hash a password string, and then derive the key from the hash. We do this by calling the CAPI functions `CryptCreateHash`, `CryptHashData` and `CryptDeriveKey`.

After a key has been generated, IBSec is ready to encrypt and decrypt data. As you might guess, data coming from the InterBase client application is encrypted before being sent to IBSec server, and then decrypted by the IBSec server before being sent to InterBase. To keep the data secure both ways, this also happens vice versa.

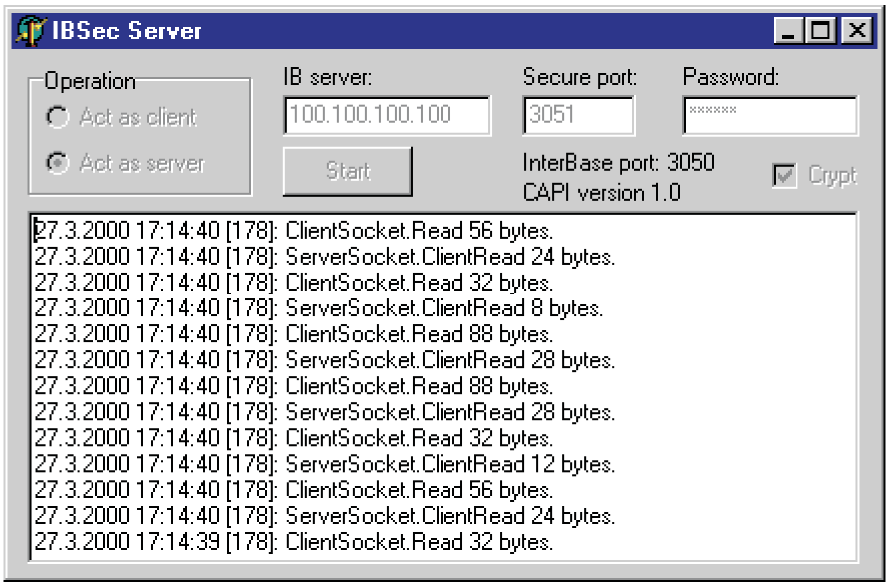
Encrypting and decrypting data is simple. Whenever data needs to be encrypted, IBSec calls a method called `Encrypt`, which takes a pointer to the data buffer and the buffer length as parameters. Then, it calls CAPI's `CryptEncrypt` function, which will encrypt the data in the buffer (see Listing 6).

Similarly, when IBSec must decrypt data, it calls the `Decrypt` method, which will use the `CryptDecrypt` function to do all the dirty work. Note that RC4 is an interesting algorithm, in that encrypting the same data twice results in the original (plaintext, unencrypted) data. Because of this we could have replaced all of the calls to `Decrypt` with calls to `Encrypt`.

Putting IBSec To Work

Now that you understand quite a bit about IBSec's internal workings, it is time to see how IBSec works in practice. To run IBSec, you need at least two computers connected using a TCP/IP network. This doesn't need to be a LAN, it could be a modem connection just as well.

To proceed, first make sure that the InterBase server is started on the server machine. Then start the IBSec server on that machine (see Figure 2). Optionally, you can start IBSec on a different computer, but then the communications between the InterBase server and IBSec will not be secure.



IB server: 100.100.100.100
Secure port: 3051
Password: ******
InterBase port: 3050
CAPI version 1.0

```
27.3.2000 17:14:40 [178]: ClientSocket.Read 56 bytes.
27.3.2000 17:14:40 [178]: ServerSocket.ClientRead 24 bytes.
27.3.2000 17:14:40 [178]: ClientSocket.Read 32 bytes.
27.3.2000 17:14:40 [178]: ServerSocket.ClientRead 8 bytes.
27.3.2000 17:14:40 [178]: ClientSocket.Read 88 bytes.
27.3.2000 17:14:40 [178]: ServerSocket.ClientRead 28 bytes.
27.3.2000 17:14:40 [178]: ClientSocket.Read 88 bytes.
27.3.2000 17:14:40 [178]: ServerSocket.ClientRead 28 bytes.
27.3.2000 17:14:40 [178]: ClientSocket.Read 32 bytes.
27.3.2000 17:14:40 [178]: ServerSocket.ClientRead 12 bytes.
27.3.2000 17:14:40 [178]: ClientSocket.Read 56 bytes.
27.3.2000 17:14:40 [178]: ServerSocket.ClientRead 24 bytes.
27.3.2000 17:14:39 [178]: ClientSocket.Read 32 bytes.
```

➤ *Figure 2: IBSec server running.*

In the `Operation` group box, select the option `Act as server`, and then set the IP address of the InterBase server. If you want to, you can change the secure port number and the password, but this is not really needed. Please note that the port and password must be the same on both the IBSec client and server. By default the password is 'secret', which is probably the most difficult password for any hacker to guess... ☺ You will change it, won't you?

After all the settings have been made, click the `Start` button. This will then put IBSec into listening mode, and after the 'Starting server services' message appears on the log, IBSec is ready to accept communications.

Now, go to your client machine and fire up IBSec client. The `Operation` group box already indicates the correct operation mode, so all you need do is type in the IP address of the IBSec server. If you changed the port and/or password on the IBSec server, modify the settings so that they are equal.

After clicking the `Start` button on the IBSec client (see Figure 3), you're ready to start any InterBase client application. For example, try the InterBase Interactive SQL (WISQL) tool. In the connection dialog box, create a remote server connection to the IBSec client

➤ *Listing 5: Initialising CryptoAPI.*

```
Procedure TIBSecMainForm.AcquireCAPIContext;
Begin
  If (Not CryptAcquireContext(@CAPIProvider,nil,MSDefProv, ProvRSAFull,0))
    Then Begin
    { Couldn't aquire context -- try to create a new keyset (init user). }
    If (Not CryptAcquireContext(@CAPIProvider,nil,MSDefProv, ProvRSAFull,
      CryptNewKeySet)) Then Begin
      Raise Exception.Create('Cannot acquire context ' +
       'to default provider: '+ SysErrorMessage(GetLastError));
    End;
  End;
End;
```

➤ *Listing 6: Encrypting data with CAPI.*

```
Procedure TIBSecMainForm.Encrypt(Buffer : PChar; BufLen : Integer);
Begin
  If (Not CryptEncrypt(CAPIKey,0,False,0,Buffer,BufLen,BufLen)) Then
    Raise Exception.Create('Cannot encrypt data: '+
    SysErrorMessage(GetLastError));
End;
```

(normally you would connect to the InterBase server directly). Use the InterBase example database Employee.gdb.

After clicking `OK`, you should see many log entries running up in IBSec's logs, both on the client and the server. If everything goes smoothly, Interactive SQL will simply go into the connected state. Now try to enter a SQL statement like `SELECT * FROM EMPLOYEE` and hit `Ctrl+Enter`. After a lot of communications back and forth, you should finally see the results in WISQL.

Further Testing

If you want to test IBSec even more, try to extract metadata from the database by choosing the `Extract database` command from the Metadata menu in WISQL. Let me warn you though: this can take a while! If you want to speed up the process, disable logging in both the IBSec client and server.

To disable logging, try right clicking the log in IBSec and choosing the `Log Enabled` command (note the `Clear Log` command also) from the popup menu which appears. When there's a check mark next to the command name, logging is enabled. I found that disabling the logging feature could dramatically improve IBSec's performance.

You can further increase the performance of IBSec by disabling encryption, although by doing this you lose the whole point of using IBSec! To disable encryption, click the `Crypt` checkbox in IBSec before hitting the `Start` button. It is not possible to change this setting on the fly.

Another interesting way to test IBSec is to use Delphi's SQL Explorer (Figure 4). Of course, you will need to define a BDE alias that points to the IBSec client. In my opinion this test is definitely worth the few seconds it takes to define an alias. In SQL Explorer you can truly see what happens when you click around inside the database. Almost every click creates network transmissions, something I was never aware of.

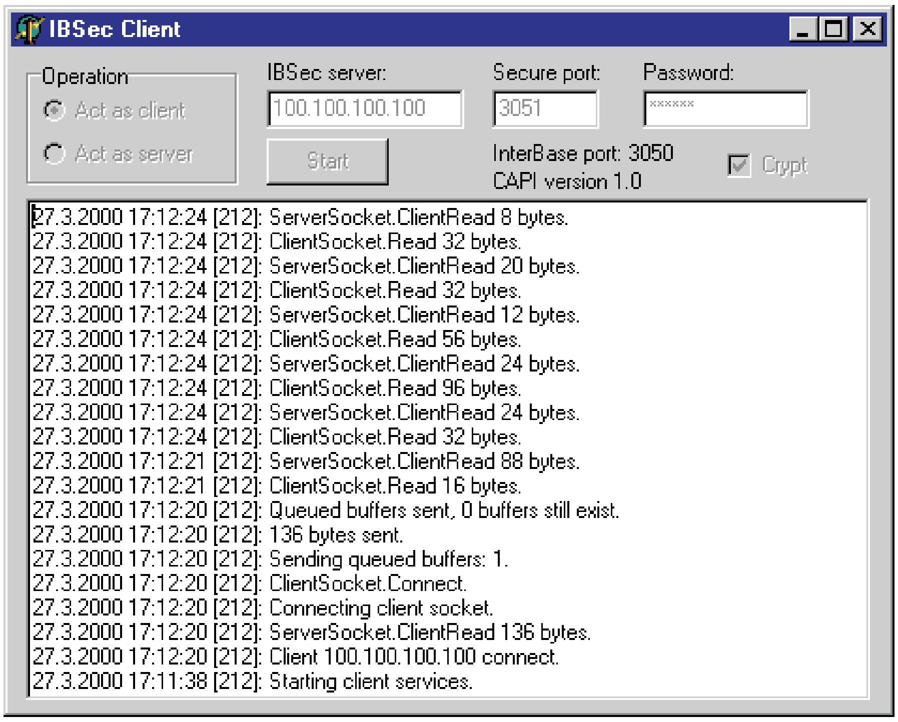
And if you're really anxious, enhance IBSec by adding a hex dump of the transmitted data. By doing this you could easily see how InterBase communicates with its client application. That would be SQL in binary format, I suppose.

Wrapping It All Up

Having got this far, you should have a good understanding of what IBSec can do for you. As you have seen, Delphi's native socket components can be used even for these kinds of advanced solutions.

I'm sure you have already thought of it, but IBSec is not

➤ *Figure 3: IBSec in the client mode.*



Operation: [✓ Act as client] Act as server

IBSec server: 100.100.100.100

Secure port: 3051

Password: ******

Start

InterBase port: 3050
CAPI version 1.0

[✓ Crypt]

```
27.3.2000 17:12:24 [212]: ServerSocket.ClientRead 8 bytes.
27.3.2000 17:12:24 [212]: ClientSocket.Read 32 bytes.
27.3.2000 17:12:24 [212]: ServerSocket.ClientRead 20 bytes.
27.3.2000 17:12:24 [212]: ClientSocket.Read 32 bytes.
27.3.2000 17:12:24 [212]: ServerSocket.ClientRead 12 bytes.
27.3.2000 17:12:24 [212]: ClientSocket.Read 56 bytes.
27.3.2000 17:12:24 [212]: ServerSocket.ClientRead 24 bytes.
27.3.2000 17:12:24 [212]: ClientSocket.Read 96 bytes.
27.3.2000 17:12:24 [212]: ServerSocket.ClientRead 24 bytes.
27.3.2000 17:12:24 [212]: ClientSocket.Read 32 bytes.
27.3.2000 17:12:21 [212]: ServerSocket.ClientRead 88 bytes.
27.3.2000 17:12:21 [212]: ClientSocket.Read 16 bytes.
27.3.2000 17:12:20 [212]: Queued buffers sent, 0 buffers still exist.
27.3.2000 17:12:20 [212]: 136 bytes sent.
27.3.2000 17:12:20 [212]: Sending queued buffers: 1.
27.3.2000 17:12:20 [212]: ClientSocket.Connect.
27.3.2000 17:12:20 [212]: Connecting client socket.
27.3.2000 17:12:20 [212]: ServerSocket.ClientRead 136 bytes.
27.3.2000 17:12:20 [212]: Client 100.100.100.100 connect.
27.3.2000 17:11:38 [212]: Starting client services.
```

Defining Custom Message Handlers

Let's say you need to create an `AfterShow` event for your application's main form. One way to solve this problem is to define a custom Windows message, and handle it like any normal event. When your app needs to define a custom Windows message, you first need to declare a unique constant value for it:

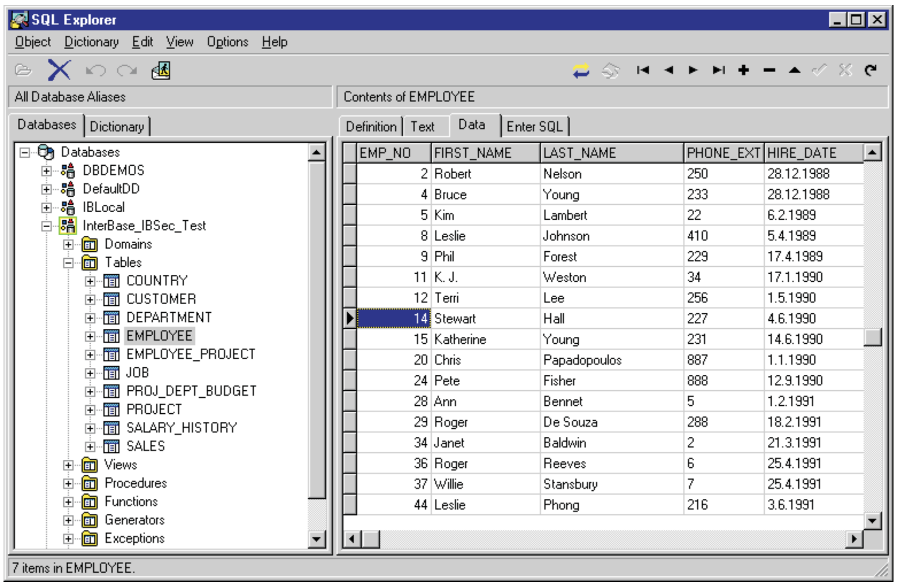
```
Const
  wm_AfterShow = wm_User+1000;
```

It is important to use the `wm_User` constant as a base value, because otherwise you could interfere with the normal message processing. After defining the constant, you need to declare a message-handling method for it:

```
type
  TForm1 = class(TForm)
    ...
    Procedure WMAfterShow(Var Msg); Message wm_AfterShow;
  end;
```

After this, you could handle the message just as you see fit. But remember that the code will not be executed without a call to `PostMessage`. It is best to call it in the form's `OnShow` event handler:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  PostMessage(Handle,wm_AfterShow,0,0);
end;
```



```
Databases
  DBDEMOS
  DefaultDD
  IBLocal
  InterBase_IBSec_Test
    Domains
    Tables
      COUNTRY
      CUSTOMER
      DEPARTMENT
      EMPLOYEE
      EMPLOYEE_PROJECT
      JOB
      PROJ_DEPT_BUDGET
      PROJECT
      SALARY_HISTORY
      SALES
    Views
    Procedures
    Functions
    Generators
    Exceptions
```

EMP_NO	FIRST_NAME	LAST_NAME	PHONE_EXT	HIRE_DATE
2	Robert	Nelson	250	28.12.1988
4	Bruce	Young	233	28.12.1988
5	Kim	Lambert	22	6.2.1989
8	Leslie	Johnson	410	5.4.1989
9	Phil	Forest	229	17.4.1989
11	K. J.	Weston	34	17.1.1990
12	Terri	Lee	256	1.5.1990
14	Stewart	Hall	227	4.6.1990
15	Katherine	Young	231	14.6.1990
20	Chris	Papadopoulos	887	1.1.1990
24	Pete	Fisher	888	12.9.1990
28	Ann	Bennet	5	1.2.1991
29	Roger	De Souza	288	18.2.1991
34	Janet	Baldwin	2	21.3.1991
36	Roger	Reeves	6	25.4.1991
37	Willie	Stansbury	7	25.4.1991
44	Leslie	Phong	216	3.6.1991

7 items in EMPLOYEE.

➤ *Figure 4: Accessing an InterBase database through IBSec.*

limited to securing only InterBase network traffic. It can secure *any* network traffic if you just change the port numbers and recompile! You could have secure Telnet, FTP, HTTP and email sessions, just by deploying a customised IBSec version to the right servers. It's that easy with TCP!

If you are interested in CAPI, I would suggest getting your hands on a Microsoft Developers Network (MSDN) Library CD-ROM, which includes a lot of information about CAPI, as well as other Microsoft technologies. You can also find most of the MSDN material for free on the internet, at http://msdn.microsoft.com (I'm sure you will immediately bookmark the site!).

I hope you will find IBSec a useful tool. If you want to create customised versions of IBSec, feel free to do so. Email is always welcome, so if you create cool IBSec customisations, do let me know!

Reference

Bruce Schneier: *Applied Cryptography, Second Edition*. John Wiley & Sons, 1996.

Jani Järvinen works as a technical support person for Inprise products. He specializes in internet and Windows API technologies. Contact him at janij@dystopia.fi