Under Construction: Internet Security

by Bob Swart

Delphi contains numerous techniques and features to support internet solutions and web applications, especially when it comes to publishing information from databases. One aspect of these techniques is seldom mentioned, and that's internet security: safeguarding your data. And your customer's data.

For example, have you ever ordered a CD from CD-Universe (www.cduniverse.com) and paid for it by credit card? In that case, I urge you to call your credit card company to block your card and get a new one (with a new number). About a month ago, hackers broke into their database and published 15,000 credit card numbers on the web, including the owners' names, addresses, etc. This page was removed within a day, but meanwhile anyone could have made a copy...

My credit card company already knew about the problem, but they did not block my credit card or send me a new one until I specifically asked for it. So you better check your accounts. Even though you'll get your money back in the end, it may take a while.

Security Levels

Security for internet applications can be defined in a number of different levels, each built upon a lower level. At the lowest level we have the operating system (I'm only talking about Windows NT here, and I assume all the Service Patches have been applied), next the web server itself (Internet Information Server in my case) and on top of that the web server application (CGI or ISAPI).

Apart from those three layers of application, we must not forget that many web server applications also deploy a database with catalogue, customer or orders information, which could be on the same machine (the web server), or on another machine (a database server). If you put it on another machine, you can place a firewall between the web server (which is connected to the internet) and the database server. This makes it much harder for hackers to get access to your database.

Firewall

A firewall can be seen as the intersection between the private and public networks. The web server itself can host the firewall, making sure people can access the website itself but nothing more. Also, if you have more than one machine on your private network, you can even use more than one firewall (one on each machine), keeping your web server on one machine and a database server on another machine (it could even be a different operating system, such as Linux, which many believe is more secure than Windows NT).

Apart from being used on web servers, firewalls are also perfect for protecting client machines that are continuously connected to the internet (through a leased line or DSL connection, for example). And if you don't think this is necessary, check out www.netice.com and examine BlackICE Defender. It scared the living daylights out of me to see multiple port scans and other (relatively innocent) probes on my machine from the outside. If one of these visitors has less than good intentions, then you are better prepared. A personal firewall will connect your client machine, just as a corporate firewall projects your web server.

Finally, when building a commercial website that depends on a web server application, you should always consider the web server (and everything on it) to be expendable. No matter how many precautions you take, hackers will probably be able to break it. It's only a matter of how much it costs to break in versus how much hackers will gain to break in (don't underestimate the sheer value of 'I did it': even non-profit sites are prone to hacker attacks). In other words, make sure you have a backup of your website and the valuable data is stored safely (and please consider erasing credit card numbers right after you've used them, so you don't risk your database being 'visited').

Proxy

A proxy is a special firewall: one that can be used to allow certain users access to another network resource by supplying usernames and passwords. Where a firewall can be used to safeguard an entire resource, a proxy is often used especially to open up a resource for specific users (like internet access from within a private network, which may only be available through a proxy). The proxy ensures nothing can get back, and people can only get through with the correct username/password.

Delphi 5 includes support for proxies with the `TWebConnection` component. This component is used in the client part of the multi-tier application, where it can connect to a remote database (another tier) through an HTTP proxy. A special ISAPI DLL, httpsrvr.dll, is used to make the connection with the database tier, and must be installed and accessible by our web server application. `TWebConnection` has a `Proxy` property in which we can specify the IP address (or name) of the proxy server. We also need to specify the `UserName` and `Password` properties to get through the proxy and access the database tier. Note that the `TWebConnection` component itself is not the place where the username/password are verified, but only routes the username and password to the proxy (as specified in the `Proxy` property).

Alternatively, we can set the `LoginPrompt` property to `True`, but this is not advisable when building a web server application (the web server application would try to show the login dialog, but it would be for the 'default internet user', and by default not even displayed on the web server machine, let alone in the client browser).

If you don't use a proxy, then you can leave the `Proxy` property empty (as well as the `username/password` properties). However, in those cases, you can also use a regular `TDCOMConnection` or `TCORBAConnection` component, rather than connect over 'plain' HTTP.

Secure Sockets Layer

Apart from securing your web server and database servers with firewalls, you can also make sure the incoming traffic (and data) is secured. This is especially important with sensitive input from the web, like customer name and credit card information. For this purpose, the SSL (Secure Socket Layer) protocol has been implemented to encrypt all data transmitted between a client and a server during the (secure) session. In a browser, you can notice this by the 'lock' which is closed during a secure session. SSL uses Secure HTTP (also called S-HTTP) which translates into an https:// prefix instead of http://. In order to be able to use SSL, you should first obtain and install public and private encryption keys from a source like VeriSign or Thawte. During the initial https:// connection, the public key is sent from the server to the client, which uses this public key to encrypt all data. When data is received by the server, the private key is used to decrypt the data. So, apart from using the https:// prefix, SSL can be more or less transparent for web server application developers using Delphi. *[We're planning a future article on Delphi and SSL in a future issue. Ed]*

➤ *Listing 1: Addition to DrBobCGI.*

```
if (Pos('HTTP_AUTHORIZATION',Str) = 1)
  or (Pos('AUTHORIZATION',Str) = 1) then begin
  Delete(Str,1,Pos('=',Str));
  Authorization := Str;
end;
```

Password-Protected Pages

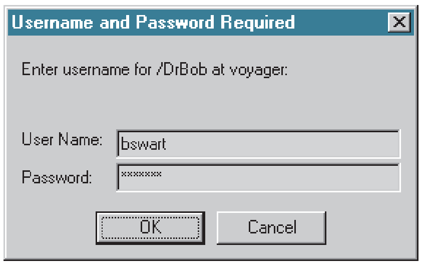
Apart from securing your database on a database server behind a second firewall, or using SSL for incoming data protection, it's sometimes also useful to have certain pages of your website available on a 'members-only' basis. This means you need some kind of authorisation. Usually, this information is not considered critical, so the simple HTTP authorisation which is built into the web server can suffice.

Using this authorisation means that we need to verify the username and password, and must generate an answer in response to a correct username/password (the 'members-only' information) or an incorrect combination (another try or an error message). Although you can set permissions on directories using IIS, it's more fun to use Delphi and dynamically check the usernames and passwords.

HTTP Headers

As you probably know, a CGI (Common Gateway Interface) application is just a console application that outputs a dynamic web page. The output consists of an HTTP header followed by an empty line and the actual content. Inside the HTTP header we specify the MIME-type of the content (usually `text/html`, but it can also be `image/gif`, or just about anything useful you can image), so the browser knows what to expect. The web server itself also adds a special first line to the HTTP header which in most cases says: `HTTP/1.0 200 OK`. This is the line containing the result code of the dynamic web page (all is OK, so display the following content).

Suppose we could eliminate that first line and replace it by a special HTTP error code, namely 401 (which means 'unauthorized') and request a user to login first. We could produce the following HTTP header to force a user to login in the realm /DrBob:


Username and Password Required
Enter username for /DrBob at voyager:
User Name: bswart
Password: *******

➤ *Figure 1*

```
HTTP/1.0 401 Unauthorized
  content-type: text/html
  WWW-Authenticate:
  Basic realm="/DrBob"
```

If we write a simple CGI console application that writes the above few lines, we get an empty page as the result. That's because the web server still adds the special first line (`HTTP/1.0 200 OK`) to this page, and the second HTTP status code line is simply ignored. In order to tell the web server *not* to add this special first line, we must rename the CGI executable and give it an `NPH-` prefix (which stands for *Non Protocol Header*). With the `NPH-` prefix, the result of a CGI executable that produces the above output will be a login dialog, as can be seen in Figure 1.

You can specify a username and password here, and click on OK, which re-executes the CGI application again and indeed again produces the same `HTTP/1.0 401` header which results in the above dialog again. In theory, the dialog should appear a maximum of three times before permission is simply denied. However, in practice I have not experienced this limit (which means that, again in theory, one could keep on trying until a correct username/password combination is found).

In order to respond to the authorization information, we need to take a look at the environment data sent from the client to the server. This login username/password pair is not part of the regular input, but can be obtained by looking at the value of either the `HTTP_AUTHORIZATION` or `AUTHORIZATION` environment variable.

For those of you who remember my `DrBobCGI` unit, in order to support HTTP authorization, we need

```
unit Base64;
interface
function UnBase64(Str: String): String;
implementation
function UnBase64(Str: String): String;
type
  TTriplet = Array[0..2] of Byte;
  TKwartet = Array[0..3] of Byte;
var
  Kwartet: TKwartet;
  Triplet: TTripLet;
  procedure Kwartet2Triplet(
    Kwartet: TKwartet; var Triplet: TTriplet);
  var   i: Integer;
  begin
    for i:=0 to 3 do begin
      case Chr(Kwartet[i]) of
       'A'..'Z': Kwartet[i] :=
          0  + Kwartet[i] - Ord('A') + 32;
       'a'..'z': Kwartet[i] :=
          26 + Kwartet[i] - Ord('a') + 32;
       '0'..'9': Kwartet[i] :=
          52 + Kwartet[i] - Ord('0') + 32;
            '+': Kwartet[i] := 62 + 32;
      else
        Kwartet[i] := 63 + 32
      end
    end;
    Triplet[0] := ((Kwartet[0] - 32) SHL 2) +
                  (((Kwartet[1] - 32) AND $30) SHR 4);
    Triplet[1] := (((Kwartet[1] - 32) AND $0F) SHL 4) +
                  (((Kwartet[2] - 32) AND $3C) SHR 2);
    Triplet[2] := (((Kwartet[2] - 32) AND $03) SHL 6) +
                  ((Kwartet[3] - 32) AND $3F)
  end {Kwartet2Triplet};
var i: Integer;
begin
  Result := '';
  while Length(Str) > 4 do begin
    for i:=1 to 4 do
      Kwartet[Pred(i)] := Ord(Str[i]);
    Delete(Str,1,4);
    Kwartet2Triplet(Kwartet,Triplet);
    for i:=0 to 2 do
      Result := Result + Chr(Triplet[i]);
  end;
  for i:=1 to 4 do
    if i <= Length(Str) then
      Kwartet[Pred(i)] := Ord(Str[i])
    else
      Kwartet[Pred(i)] := Ord('/');
  Kwartet2Triplet(Kwartet,Triplet);
  for i:=0 to 2 do
    Result := Result + Chr(Triplet[i])
end {Unbase64};
end.
```

➤ *Listing 2: Base64 unit for base64-decryption.*

to declare a string variable named `Authorization` and include the code in Listing 1 in the `initialization` section of the unit (the full source code is on the companion disk).

Armed with this new version of the `DrBobCGI` unit, we can write standard console CGI applications that can respond to the HTTP authorization input. There's just one more catch: the `Authorization` string contains encrypted data of the form `Basic XXXXXXXX`, ie the 6-character prefix '`Basic `' (to indicate we wish to use the basic authorization scheme) followed by the Base64-encoded username/password pair. After we strip the first six characters from the `Authorization` string, we can use an `UnBase64` function (see Listing 2) to decrypt it to obtain the username and password as input to the browser's authorization dialog.

Once we add the new `DrBobCGI` and `Base64` units to the `uses` clause of our standard console CGI application, we can invoke and respond to the authorized input. Invoking the dialog is easy: just return the `HTTP/1.0 401` headers (and make sure you've renamed the CGI executable to have an `NPH-` prefix). Responding to the authorization input is also easy, since we can now use the global `Authorization` variable (defined and assigned in the `DrBobCGI` unit). Once we strip the first six characters from it, we can use the `UnBase64` function to decode it, resulting in the username and password.

```
program login;
{$APPTYPE CONSOLE}
uses DrBobCGI, Base64;
begin
  if (Authorization = '') then begin
    writeln('HTTP/1.0 401 Unauthorized');
    writeln('content-type: text/html');
    writeln('WWW-Authenticate: Basic realm="/DrBob"')
  end else begin
    writeln('HTTP/1.0 200 OK');
    writeln('content-type: text/html');
    writeln;
    writeln('<HTML>');
    writeln('<BODY>');
    writeln('['+Authorization+']');
    if Pos('Basic ',Authorization) = 1 then
      Delete(Authorization,1,6);
    writeln('<P>');
    writeln('['+Authorization+']');
    writeln('<P>');
    writeln('['+UnBase64(Authorization)+']');
    writeln('</BODY>');
    writeln('</HTML>')
  end
end.
```

➤ *Listing 3: Standard CGI Authorization example.*

The login program from Listing 3 gives an example usage. Of course, in real life, we need to actually verify the username and password before the `HTTP/1.0 200 OK` header will be generated, together with the dynamic 'members only' web page content. The example in Listing 3 simply presents and decodes the given HTTP authorization input. Which also shows that it's easy for a packet-sniffer to 'decode' this data as well, never trust critical or truly sensitive data to be safe in this way. And you will probably want to ensure the members-only information is stored in encrypted files on the web server.

The output from the sample login application (renamed as NPH-TDM55.EXE) can be seen in Figure 2. Note the username `bswart` and password `drbob42` with a colon in between them.

In theory, for an even more secure way to handle a 'members-only' part of your website, you could consider combining SSL and HTTP authorization. This means you need to obtain an SSL key to enable the use of https:// on your website (so data will be sent encrypted) and even then you can protect individual dynamic pages with a username and password scheme using HTTP authorization. And remember that a combined encryption technique is stronger than the individual encryption techniques alone (assuming the authorization HTTP header gets encrypted as well, which I could not verify for certain at this time).

WebBroker Approach

It's nice to know how to invoke and listen to HTTP authorization from a standard console CGI application, but what about WebBroker and InternetExpress? Well, both are based on WebModules, and inside the `OnAction` event handler for `WebActionItems` we can use the `Request` argument (type `TWebRequest`) which has an `Authorization` property holding the base64-encoded HTTP authorization information (just like the global variable `Authorization` that I defined in the `DrBobCGI` unit).

Specifying custom HTTP headers also works the same way: just rename the CGI executable so it has the `NPH-` prefix, and you can return any HTTP response header using the `Response.StatusCode` property. The authorization type can be specified in the `WWWAuthenticate` property (I've only explored the `Basic` type), and the `Realm` can be set with the `Realm` property of the `Response` argument.

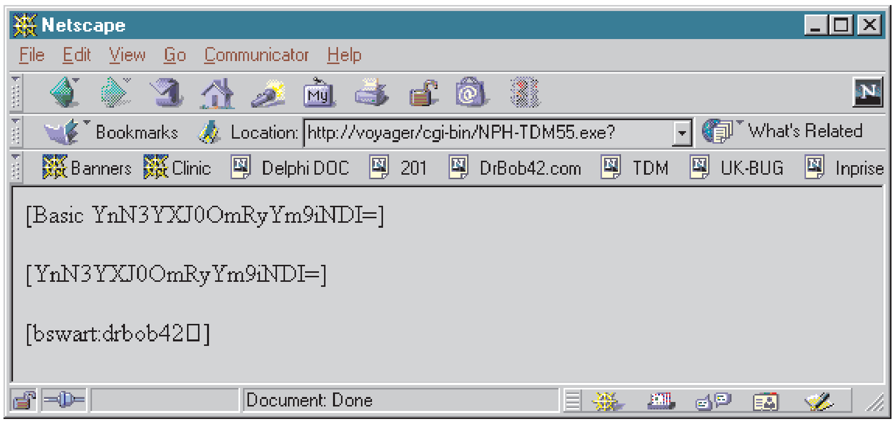
Once I've finished generating the combined custom headers, I always call the `Response.SendResponse` method, to avoid delays and directly send the HTTP headers to the client browser.

Note that the code in Listing 4 allows any 'bswart' to enter, either as part of the username or password. Not very practical, but just an example to show how you can obtain and decode the `Response.Authorization` information inside a WebBroker or InternetExpress application.

ActiveX Security

As most of you should know by now (especially if you've read my articles over the years): ActiveX controls and ActiveForms are also a potential security risk. Not for server databases and applications, but for clients. An ActiveX control, like an ActiveForm, is not restricted to the browser window. It's just another 32-bit application running on your machine: just like your other applications, with the same permissions on your machine (and on your network and on the internet). As a result, an ActiveForm can do virtually anything behind the scenes. Even if it looks like nothing is happening, it can overwrite your registry or send your credit card number and other sensitive information to the ActiveX's author.

To decrease this security threat, ActiveX controls can be code signed, for example by a company like VeriSign (the same company where you can obtain an SSL certificate). However, code signing only indicates that you can indeed backtrack to the author, ie that you can somehow truly identify the author of the ActiveX. There's nobody who claims that code signing will make the ActiveX control indeed safe.


Netscape
File Edit View Go Communicator Help
Bookmarks
Location:
What's Related
Banners Clinic Delphi DOC 201 DrBob42.com TDM UK-BUG Inprise
Document: Done

http://voyager/cgi-bin/NPH-TDM55.exe?

[Basic YnN3YXJ0OmRyYm9iNDI=]

[YnN3YXJ0OmRyYm9iNDI=]

[bswart:drbob42□]

➤ *Figure 2*

As an example, Fred McClain once wrote an ActiveX control called Exploder, which demonstrates security problems with Microsoft's Internet Explorer. Exploder performs a clean shutdown of Windows 95 and turns off the power on machines which have a power conservation BIOS. For more information, visit www.halcyon.com/mclain/ActiveX/welcome.html (this URL won't shut down your machine, it contains the information and a link to the Exploder ActiveX control itself).

Exploder was even code signed for a while (before Fred was 'asked' by VeriSign and Microsoft to remove it). But although removed, it emphasises that users should never blindly accept code signed ActiveX controls from the internet (and certainly not ActiveX controls that are not code signed, of course).

Next Time

After all these security issues, it's time to take it easy and relax.

So next time we'll return to the topic of VisiBroker CORBA exceptions (left uncovered in the last issue) and see just what it means for existing and new CORBA applications written in Delphi 5 Enterprise.

So stay tuned...

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is an IT Consultant for TAS Advanced Technologies, and a freelance technical author.

➤ *Listing 4: WebBroker Authorization example.*

```
procedure TWebModule1.WebModule1WebActionItem3Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var Auth: String;
begin
  Auth := Request.Authorization;
  if Pos('Basic ',Auth) = 1 then
    Delete(Auth,1,6);
  Auth := UnBase64(Auth);
  if Pos('bswart',Auth) = 0 then begin { any "bswart" may enter }
    Response.StatusCode := 401;
    Response.WWWAuthenticate := 'Basic';
    Response.Realm := '/DrBob';
    Response.SendResponse;
  end else begin
    Response.Content := 'Welcome: ['+Request.Authorization+']=['+Auth+'])'
  end
end;
```