SSL Sockets From RPG? Of Course You Can!

By Scott Klement

Sockets enable a program to communicate over a TCP/IP network. Any data that you write into a socket gets broken into network packets and sent over your network or the Internet to the destination computer where it's put back together and read by another program. This communication is the cornerstone of the Internet, sending billions of bytes whizzing all over the world. But it has a flaw! Every byte you send over the Internet is sent through dosens of networks, and computers running on those networks can see all of your bytes! If you want to say something private, you need to encrypt it!

The industry standard for encrypting TCP/IP communications is a protocol known to most people as Secure Sockets Layer (SSL). This article provides some background about SSL, and teaches you how to use the Global Secure Toolkit (GSKit) API provided with i5/OS to write your own SSL applications in ILE RPG.

Transport Layer Security

SSL was developed by Netscape Communications Corporation back in the early years of the Web. They released three versions aptly numbered 1.0, 2.0 and 3.0. After this, the SSL technology was developed by the Internet Engineering Task Force (IETF), the organization that decides the open standards for the Internet. The IETF renamed the protocol to be called Transport Layer Security (TLS) . IETF released version 1.0 in 1999, and followed it up recently with version 1.1 in April of 2006. Despite the name change, TLS and SSL are really the same thing, there are only minor differences between them. For all intents and purposes, TLS version 1.0 is actually the 4th version of SSL.

For the remainder of this article, when I refer to SSL, I mean both TLS and SSL.

A Secret Agent's Encoder Device

SSL is a "layer" above sockets. When I talk about sockets, I like to use a telephone call as an analogy. In both cases, you create a communications channel to another entity and you talk back and forth to exchange information. Just as a telephone operator (or the NSA!) might listen in on your telephone calls, a third-party can view your Internet communications.

Imagine a special encoder device that a secret agent might use when making a telephone call. The agent hooks this special device over the telephone receiver, and it scrambles any data that he speaks so that the operator won't understand it. On the other end of the line, another spy has a decoder device that descrambles it so that he can understand the words! Pretty cool, hey! I need one of those. What was I talking about, again? Oh yeah, SSL.

SSL provides that same level of privacy when talking with sockets. Anything you send over the socket using the SSL API will be encrypted and unreadable by third parties while in transit. When it arrives, it's decrypted and ready for the program on the other end to uise.

i5/OS provides two sets of APIs for SSL programming, the original SSL APIs first became available in V4R3, and the newer GSKit APIs became available with the release of V5R1, though they're available in V4R5 by installing a PTF. IBM is no longer enhancing the original SSL APIs, and the GSKit ones are easier to use, so this article will work with GSKit, only.

Authentication

SSL solves two problems, that of encryption and that of authentication. In other words, not only does SSL encrypt data so that it's unreadable to third parties, but it also verifies the identity of the program at the other end of the connection. After all, it doesn't do you any good to encrypt data if it's sent to, and decoded by, a malicious hacker! You need to know that you're sending it to someone you trust.

How does it know who it's talking to? Each side of the connection has a digital identity called a "certificate." This certificate contains a cryptographic key that provides positive identification. For example, I could have a certificate on my Web server that identifies it as www.klements.com. When you connect to me with SSL, that certificate is sent to you.

How do you know that I didn't lie about being www.klements.com? When certificates are issued, they must be issued by a certificate authority (CA). The CA digitally signs the certificate to verify that the CA did some research and that it vouches that I'm really Scott Klement, and that I really own www.klements.com. If you can trust the CA, then you know that I'm really who I say I am.

The way you say that you trust a certificate authority is by installing their CA certificate on your system. When you connect with SSL to a system claiming to be signed by that CA, cryptography is used to verify that the key in the CA certificate was used to sign the server certificate of the system you connected to.

Imagine running a site like Amazon.com, where millions of people are connecting with SSL. Surely, it's impractical for Amazon to ask every single customer to download its CA certificate and install it into that customer's Web browser? For that reason, some of the major certificate authority companies have their CA certificates automatically installed when SSL is installed.

VeriSign and Thawte are two of the best-known CA companies. You can order a certificate from them, and they will then do a little bit of research to make sure that you are who you claim to be. Once issued, their certificates are trusted by everyone, because their CA certificate was installed automatically when SSL was installed.

On the other hand, if I create my own certificate authority, and issue my own certificates, it can save me some money — especially if I have hundreds or thousands of certificates to issue! The only problem is that my certificate authority certificate would have to be manually installed into the browser, or other SSL software, of anyone who wants to use my services. This actually works very nicely within my company. I can create my own certificates for my TN5250 server, and I can issue a separate certificate for each

employee who wants to connect to it. I set things up so that only my own CA is trusted for TN5250. That way, I know that everyone who connects or uses my system is really an employee!

Enough Background!
There are lots of details involved in cryptography, but fortunately the APIs take care of most of the work, and we don't have to worry about it. I hope that the introduction that I've given here gives you enough information to get started. If not, I suggest reading the SSL introduction that Apache has on its Web site. You can find it at the following link: http://httpd.apache.org/docs/2.2/ssl/ssl_intro.html

Required IBM Software
Before you can start writing software that uses SSL, you need to have the right stuff on your iSeries. Here's a link to the Information Center area that has information about the licensed programs required:
https://www.ibm.com/docs/en/i/7.4?topic=security-tls-prerequisites

All the required software is available on your OS/400 or i5/OS installation media, except for the IBM Cryptographic Access Provider (5722-AC3). If you don't already have that, you need to order it from your IBM Business Partner. It's a no-charge item, but it has to be ordered separately due to the various laws governing cryptography around the world.

If you're already using SSL for something else on your system, you already have these licensed programs installed.

Configuring the Software
After the software is installed, you need to create a "certificate store." This is where SSL certificates and their digital keys are stored on the iSeries. The default certificate store is called *SYSTEM. Again, if you're already using SSL on your system, you've already created the *SYSTEM certificate store. If not, you can do so by following these steps:

1. Certificates are managed in a tool called the Digital Certificate Manager (DCM). The DCM is an easy-to-use Web interface for managing certificates provided by the HTTP Administration server. If you don't already have it running, you need to start it with the following command:

   STRTCPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)

2. Connect to the admin server by pointing your browser at port 2001 of your iSeries system. If your iSeries is called "www.example.com," you point the browser at http://www.example.com:2001.

3. Click "Digital Certificate Manager."

4. Click "Create New Certificate Store."

5. Follow the on-screen prompts to create a new *SYSTEM certificate store.

6. You do not need to create or assign any certificates for the examples that I'll provide with this article.

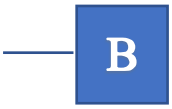
Creating an Environment

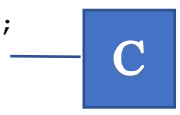
The first step to writing an SSL program is creating an environment from which SSL certificates can be created or verified. The environment contains all the cryptographic components for the local side of the connection. You can reuse this environment for many sockets if you like, because the local part of the information won't need to change.

A lot of different parameters are needed to create an SSL environment, but the API does a good job of hiding the complexity for me by providing default values for all the settings. The only values that I need to change are the ones that differ from the defaults.

Figure 1 shows the RPG code that I used to create a GSKit environment:

```
P CreateEnv       B
D CreateEnv       PI                  like(gsk_handle)
D rc              s             10I 0
D SslEnv          s                   like(Gsk_handle)

 /free

   // Create an SSL environment with default values:

    rc = gsk_environment_open(SslEnv);
    if (rc <> GSK_OK);
       return *NULL;
    endif;

   // Tell the environment to use the *SYSTEM certificate
   //  store

    rc = gsk_attribute_set_buffer( SslEnv
                                 : GSK_KEYRING_FILE
                                 : '*SYSTEM'
                                 : 0 );
    if (rc <> GSK_OK);
       gsk_environment_close( SslEnv );
       return *NULL;
    endif;

   // Tell the environment that this is a client connection

    rc = gsk_attribute_set_enum( SslEnv
                               : GSK_SESSION_TYPE
                               : GSK_CLIENT_SESSION );
    if (rc <> GSK_OK);
       gsk_environment_close( SslEnv );
       return *NULL;
    endif;
```





```
   // Activate the new environment.

    rc = gsk_environment_init( SslEnv );
    if (rc <> GSK_OK);
       gsk_environment_close( SslEnv );
       return *NULL;
    endif;

    return SslEnv;
 /end-free
P                 E
```

I start by calling the gsk_environment_open() API (A in Figure 1). This creates a data structure in the API's memory that contains all of the default SSL settings. It provides me with a pointer that I can't use directly but can pass to the other GSKit APIs to tell them which environment I'm referring to.

There are APIs called gsk_attribute_set_buffer(), gsk_attribute_set_enum(), and gsk_attribute_set_numeric_value() that I can call to change the values of fields in the data structure that contains the environment's parameters. For a simple client application, I need to tell the system that I want to use the *SYSTEM certificate store (B in Figure 1), and that I want to use it to create client sessions (C in Figure 1).

After I've changed any of the parameters that I need to change, I call the gsk_environment_init() API (D in Figure 1). It uses the parameters in the data structure to initialize an SSL environment and it does any of the cryptographic processing that can be done at this point. After I've called gsk_environment_init(), my environment is set up, and I can no longer change the parameters for it.

Establishing a TCP Connection

The SSL protocol is a "layer" that runs on top of a standard socket. That's why it's called Secure Socket Layer. To use it, I need to create a standard socket and connect it to a server.

Figure 2 shows how to create a connection.

```
P ConnSock        B
D ConnSock        PI            10I 0
d   host                       256A   const
D   port                        10I 0 value
D s               s             10I 0
D addr            s             10U 0
 /free

    // look up host

    addr = inet_addr(%trim(host));
    if (addr = INADDR_NONE);
```

```
            p_hostent = gethostbyname(%trim(host));
            if (p_hostent = *NULL);
                errMsg = 'Host not found!';
                EscapeMsg();
            endif;
            addr = h_addr;
        endif;

        // Create a socket

        s = socket(AF_INET: SOCK_STREAM: IPPROTO_IP);
        if (s < 0);
           ReportError();
        endif;

        // connect to the host

        connto = *ALLx'00';
        connto.sin_family = AF_INET;
        connto.sin_addr   = addr;
        connto.sin_port   = port;

        if (connect(s: %addr(Connto): %size(connto)) = -1);
           callp close(S);
           ReportError();
        endif;

        return s;
     /end-free
    P                 E
```

There's nothing special about the socket code in Figure 2. It simply creates a connection to a server. If you're not familiar with socket programming in RPG, take a look at the article entitled "TCP/IP and Sockets in RPG" published in the May 2006 issue of iSeries NEWS (article ID 20520).

Upgrading the Connection to SSL

Now that I have a TCP connection to a server, I need to "upgrade" it to SSL. Upgrading to SSL uses a similar paradigm to that of creating an SSL environment. You first create a data structure containing all the default settings for the new SSL session using the gsk_secure_soc_open() API. You can then change any of those default settings. When you have all the settings the way you want them, you initialize the secure socket by calling the gsk_secure_soc_init() API.

The gsk_secure_soc_init() API does more than just cryptography. It also uses your connected socket to exchange certificates as well as other cryptographic information. This exchange is often called the "SSL Handshake."

Figure 3 demonstrates upgrading a socket to SSL.

```
P UpgradeSock     B
D UpgradeSock     PI                  like(gsk_handle)
```

```
D    SslEnv                           like(gsk_handle) value
D    sock                        10I 0 value
D Handle          s                   like(Gsk_handle)
 /free
     rc = gsk_secure_soc_open(SslEnv: Handle);
     if (rc <> GSK_OK);
        errMsg = %str(gsk_strerror(rc));
        return *NULL;
     endif;

     rc = gsk_attribute_set_numeric_value( Handle
                                         : GSK_HANDSHAKE_TIMEOUT
                                         : 30 );
     if (rc <> GSK_OK);
        errMsg = %str(gsk_strerror(rc));
        gsk_secure_soc_close(Handle);
        return *NULL;
     endif;

     rc = gsk_attribute_set_numeric_value( Handle
                                         : GSK_FD
                                         : sock );
     if (rc <> GSK_OK);
        errMsg = %str(gsk_strerror(rc));
        gsk_secure_soc_close(Handle);
        return *NULL;
     endif;

     rc = gsk_secure_soc_init( Handle );
     if (rc <> GSK_OK);
        errMsg = %str(gsk_strerror(rc));
        gsk_secure_soc_close(Handle);
        return *NULL;
     endif;

     return Handle;
 /end-free
P                 E
```

It starts by calling the gsk_secure_soc_open() API (A in Figure 3) to tell the API to create a data structure, then proceeds to change two settings in that structure.. The first one is a timeout value of 30 seconds for the handshake process, and it prevents a long delay from happening if you connect to a site that doesn't support SSL (B in Figure 3). The other setting is the socket descriptor itself. For the SSL API to use your connected socket, you have to tell it the descriptor (C in Figure 3). That's how it knows which socket to communicate over.

Finally, it calls the gsk_secure_soc_open() API (D in Figure 3) and this is when the SSL handshake takes place.

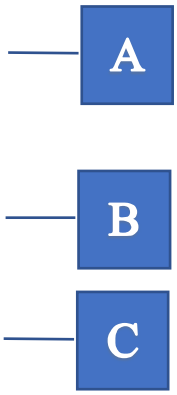
Sending and Receiving Secure Data

After the handshake is complete, your socket has successfully been upgraded to SSL, and the certificates have been verified, and the encryption is active. Relax! The hard part is

done! You can now send and receive encrypted data over the connection. To do that, use the gsk_secure_soc_write() and gsk_secure_soc_read() APIs. (Do not use the send() or recv() APIs that you'd use in a normal socket program, because those send the data unencrypted!)

Figure 4 demonstrates sending data over a secure connection.

```
cmd = 'GET /cgi-bin/ssltest HTTP/1.0' + CRLF
    + 'Host: www.klements.com' + CRLF
    + 'Connection: close' + CRLF
    + CRLF;
len = %len(%trimr(cmd));

QDCXLATE( len
        : cmd
        : 'QTCPASC' );

callp gsk_secure_soc_write( Handle
                          : %addr(cmd)
                          : len
                          : bytesSent );
```



I start by putting the data that I want to send into an RPG variable (A in Figure 4). Because the server expects the data to be in ASCII, I use the QDCXLATE API to translate it from EBCDIC to ASCII (B in Figure 4).

The gsk_secure_soc_write() API (C in Figure 4) takes care of encrypting the data and sending it to the server. The *Handle* variable in is the secure socket handle returned from the gsk_secure_soc_open() API. The *cmd* variable contains the data that I wanted to send, the *len* variable contains the length of the data, and the bytesSent variable is returned to me to tell me how many bytes were sent successfully.

Receiving data works the same way, except that you call the gsk_secure_soc_read() API.

Figure 5 demonstrates receiving data in a loop until an error occurs.

```
Reply = *blanks;
left = %size(Reply);
buf = %addr(reply);
received = 0;

// keep reading until we get the entire response

dou (rc <> GSK_OK);

    rc = gsk_secure_soc_read( Handle
                            : buf
                            : left
                            : bytesRead );
    if (rc = GSK_OK);
         received = received + bytesRead;
         buf = buf + bytesRead;
```

```
            left = left - bytesRead;
        endif;
    enddo;

    QDCXLATE( received
            : reply
            : 'QTCPEBC' );
```

Closing the Connection
When you're done sending and receiving data, you need to close the socket normally and also close the secure socket using the gsk_secure_soc_close() API. When you're done with the GSKit environment, you should also close that by calling the gsk_environment_close() API.

Figure 6 demonstrates calling these APIs.

```
gsk_secure_Soc_close( handle);
callp close(sock);
.
.
gsk_environment_close( SslEnv );
```

It's All About Trust
SSL applications determine which programs to trust and which programs not to trust based on their certificates. In theory, a server named www.acme.com will provide a certificate that's issued to Acme, Inc, proving to you that you're really communicating with Acme. But what's to stop a hacker from creating his own Acme, Inc certificate?

To protect against falsified information in certificates, there are certificate authorites (CA for short). A certificate authority is a company who's job is to provide assurance that someone really is who they claim to be. For example, if Klement Sausage Co, Inc. wanted an SSL certificate that everyone would trust, it could order one from a CA. The CA would do some checks to make sure that there really is a company with that name located at the address provided in the application. They'd check to make sure that the business exists and that it's really the same business who applied for the certificate. Once that has been done, they'll issue an SSL certificate to Klement Sausage Co, Inc and they'll digitally sign that certificate so that everyone knows that the CA trusts them.

Someone who wants to communicate with Klement Sausage would install the CA's certificate onto their machine, and tell their application to trust that CA. Once this has been configured, their program will trust any certificates that the CA trusts, including Klement Sausage.

When you create the *SYSTEM certificate store, it automatically installs the CA certificates of the largest CA companies into your Digital Certificate Manager. The most popular CAs are named VeriSign and Thawte, and virtually all SSL software trusts them by default.

Controlling SSL Details with an SSL Application Profile

The SSL environment that I demonstrated in Figure 1 specifies that the secure socket will use the default settings for the *SYSTEM certificate store. It will automatically trust any CA certificates installed into that certiicate store and will successfully communicate with any server that has a certificate signed by one of those CAs. In many situations, it's advantageous to have finer-grained control over which CAs you trust. Fortunately, there's a way to create a profile for your application, so that you can control all of the SSL details through the DCM's GUI user interface..

To do that, go back to the DCM and select the following options:

1. Click the "Select a Certificate Store" button.

2. Choose the *SYSTEM certificate store, click Continue, and then enter the password for the certificate store and click Continue again.

3. On the left-hand side of the screen, click Manage Applications, then Add Application

4. Select "Client" because this is a client program.

5. Key in the Application ID. By convention, the application ID consists of the company name, followed by an underscore, then the application name, another underscore, and finally the specific component of the application.

   For example, if you work for Acme, and are creating a program for your accounts payable system that posts checks, you might use an application ID of ACME_ACCTPAY_POSTCHECK

6. The rest of the settings can be configured to suit the application you're writing. For example, if you want to control which certificate authorites your program trusts, you can select "Define the CA trust list" for this profile. If you do that, you'll need to use the "Define CA trust list" option (also under Manage Applications) to specify which CA certificates are trusted and which are not.

7. Once you've selected all of the necessary options for your application, click Add to add the new application profile.

8. If you need a certificate assigned to this application, you can use the "Update certicicate assignment" option of the Manage Applications menu to assign an SSL certificate to the application. This is manditory for a server, but optional for a client applcation.

To tell your RPG program to use this profile, you set an "application ID" when you create the SSL environment. The system will use this ID to find the corresponding profile in the digital certificate manager.

Figure 7 shows the code that specifies the application ID to the GSKit API.

```
rc = gsk_attribute_set_buffer( SslEnv
                             : GSK_OS400_APPLICATION_ID
                             : 'ACME_ACCTPAY_POSTCHECK'
                             : 0 );
if (rc <> GSK_OK);
   errMsg = %str(gsk_strerror(rc));
   gsk_environment_close( SslEnv );
   return *NULL;
endif;
```

GSKit will find that application ID in the Digital Certificate Manager and will use it to configure the SSL settings for your program. The code in Figure 7 should be used in lieu of (and not in addition to) the code from callout B of Figure 1.

Since the SSL protocol was originally designed as a way to secure shopping carts on Web sites, client programs are not usually required to send an SSL certificate. It's important for a shopper to know that the server is geniune so that he doesn't send credit card information to an untrusted third-party. However, the store doesn't typically use certificates to verify that the customer is who he claims to be. For this reason, the SSL protocol requires that the server always has a valid, trusted certificate. A client-side certificate is optional.

In business applications, it's often nice to send a client-side certificate. In fact, if you write your own server applications, you can configure their application profile to only allow connections from clients that have trusted client-side certificates. For example, the Digital Certificate Manager has an option to create your own CA and issue your own certificates. You could create a program that only trusts certificates from that CA, and therefore, will only trust certificates that you issue yourself. You could issue client-side certificates to all of your business associates, and write a server program that only accepts connetions from programs that have those certificates.

To tell your RPG application to send a certificate, use the "Update Certificate Assignment" option from the Manage Applications menu of the DCM. If you assign the certificate to a client application, that application will use the certificate when a client-side certificate is needed. If you assign one to a server application, the server will send that certificate to any client that connects, and it's up to the client to decide whether or not to trust it.

Server Applications

Like SSL client applications, server applications are implemented as a layer on top of the standard socket API calls. To understand how to write an SSL server application, you should first be familiar with writing an standard TCP server application. You can learn more about that by reading the article entitled TCP/IP Server Programming from the November 17, 2005 issue of Club Tech iSeries Programming Tips (iSeriesNetwork.com article ID 51809).

With server programs, you always want to use an application ID, since you always want to assign a certificate to the application. If you don't assign a certificate, a server application will fail with an error message.

Although the socket programming is different when you write a server application, the SSL part of it doesn't change much. You have to select "server" instead of "client" when creating the profile in the DCM. You also have to tell the GSKit API that you're providing a server application by specifying GSK_SERVER_SESSION instead of GSK_CLIENT_SESSION when you create the SSL environment. Other than those two things, SSL server programming is just like client programming.

A standard TCP server program will call the accept() API when it wants to wait for a client program to connect to it. Once that client has connected, the socket can be upgraded to SSL, just as you would in a client application. Data is also sent and received the same way it was in the client connection, with the gsk_secure_soc_read() and gsk_secure_soc_write() APIs.

Because server programming is so similar to client programming, I haven't provided any figures demonstrating it in this article. However, the code downloads for this article will provide an entire sample client/server application demonstrating both the client and server side programming.

Sample Code and Documentation

To demonstrate everything that I discuss in this article, I've written three sample programs.

1. SSLCHECK is a simple example of an HTTP client. It connects to my web server at www.klements.com using an SSL connection. It runs a program on the Web server that reports whether I'm running SSL or not.

2. SSLCLIENT is an example of a custom application that uses SSL. It connects to an SSL server program and asks for information about a customer number. It then receives the customer information over the secure channel and displays it on the screen.

3. SSLSERVER provides the server program for the SSLCLIENT program to communicate with. It asks the client for the customer information, and sends back the results.

You can download these sample programs from iSeriesNetwork.com/code. There are instructions for compiling each one at the top of each source member.

The reference information for the GSKit API is in the Information Center at the following link:
https://www.ibm.com/docs/en/ssw_ibm_i_74/apis/unix9.htm

It's Easy!

The GSKit APIs hide most of the complexity of the cryptography from you. They make it easy to write applications that talk over a secure, private, communications channel. I hope you've enjoyed my introduction to SSL programming in RPG!