Configuring SSL for IBM Data Server Driver for ODBC and CLI

Chetan Papaiah
Ashok V. Jose
Chris Meyer

December 10, 2015

The DB2 database system supports Transport Layer Security (TLS) and Secure Sockets Layer (SSL), which means that a DB2 client application that also supports SSL can connect to a DB2 database by using an SSL-protected socket. The TLS/SSL connection in the Call Level Interface (CLI) can be established by using server authentication or client authentication. This article discusses how to configure a TLS/SSL connection on a client and a server for the different types of authentication and discusses how the TLS/SSL connection process has been simplified.

Introduction to TLS and SSL

The most common method that applications and websites use to protect their transmitted data is Transport Layer Security (TLS). TLS was formerly known as Secure Sockets Layer (SSL) protocol. SSL was developed by Netscape in 1994 and then adopted by the Internet Engineering Task Force (IETF) as RFC 2246.

SSL V3.0 was the last SSL version and all new development of the protocol continues under TLS. Given the similarity between the protocols at the time the IETF adopted TLS, TLS V1.0 is sometimes referred to as SSL V3.1. The most current protocol version as of this writing is TLS V1.2. While TLS is not interoperable with SSL, it does provide the capability to revert to SSL when a server or website is communicating with a non-TLS capable partner. However, the use of SSL is no longer recommended due to known weaknesses in the protocol.

Throughout this document, we will use the terms SSL and TLS interchangeably, unless otherwise noted, but it is preferred to use the most recent protocol version possible to achieve the best connection security.

Both TLS and SSL rely on Public Key Infrastructure (PKI) to identify and authenticate the communication partners. PKI is based on X.509 digital certificates and uses a trusted third party, a certificate authority (CA), to vouch for the identity of one or both ends of the connections.

It's important to keep certain data private, including:
• Credit card information
• Private correspondence
• Personal details
• Sensitive company information
• Bank account information

TLS/SSL uses numerous cryptographic methods to protect data as it traverses the network, giving customers confidence that their data will stay secure. Encryption is an important cryptographic method, and encryption systems generally belong in one of two categories:

• Symmetric-key encryption
• Public-key, or asymmetric, encryption

Symmetric-key encryption

Symmetric-key encryption involves the use of a single secret that is known by both the sender and receiver of the encrypted data. That key is used to encrypt and decrypt the data. Advanced encryption standard (AES) is a commonly used symmetric algorithm. Symmetric encryption can be implemented rather efficiently in software or hardware, so it's a good choice for encrypting large amounts of data. In symmetric encryption systems, securely sharing the secret keys between communication systems is a challenge. If the key is exposed to anyone other than the sender or receiver, then the security of the session is compromised. There are usually at least two secret keys (one for each direction) to share for a conversation. In the Internet world, people expect websites and applications to sustain secure conversations with hundreds or even thousands of partners, rendering symmetric cryptography not a practical means by itself for securing conversations over the Internet.

Public-key (asymmetric) encryption

Public-key encryption or asymmetric encryption uses a pair of mathematically related keys to encrypt and decrypt data. Data encrypted with one of these keys can only be decrypted by using the other key of the pair. Rivest-Shamir-Adleman (RSA) is probably the most well-known asymmetric algorithm. In public-key encryption, one key of the pair is kept secret (the private key), while the other is public (the public key). The use of public keys helps to solve the key distribution problem we saw with symmetric encryption. However, asymmetric encryption requires more computing power than symmetric encryption (by an order of magnitude), so it is not a good candidate for encrypting large amounts of data.

Another useful application of asymmetric cryptography is digital signatures in which a condensed value (a hash) of a message is encrypted with the sender’s private key. This signature value is then sent with the original clear text message from which the hash value was generated. The recipient decrypts the signature by using the sender's public key to reveal the hash value and then runs the same hash algorithm against the received clear text message to obtain his or her own condensed value. This hash value is then compared to the hash value sent after it is decrypted by using the public key of the sender. If the two hash values match, it indicates that:

• The received message was not modified, since the hash value at the reception is the same as the one that was sent.
• The origin of the message was authenticated since the digital signature was decrypted by using the sender’s public key.

The **digital certificate** is a file that is digitally signed by a certificate authority (CA), which is a third party that is trusted by both communicators in an SSL session. A certificate contains information about the owner of the certificate, including the owner’s email address, name, certificate usage, duration of validity, a distinguished name (DN) that includes a website or email address), a serial number that is assigned by the CA, and the certificate ID of the CA that signs this information. It also contains the public key of the certificate owner. The CA’s signature is a hash of the certificate contents that is encrypted by using the CA’s private key, which not only proves that the certificate is authentic, but that its contents were not unintentionally modified.

Usually, a browser or application already loaded the root certificate of well-known CAs or root CA certificates. The CA maintains a list of all signed certificates and a list of revoked certificates. A certificate is not secure until it is signed. Once a certificate is signed, it cannot be modified. All root CA certificates are self-signed. On receiving a digital certificate, one can check the digital signature of the CA by using the CA public key. Verifying the signature proves the integrity and origin of the digital certificate. If the CA is trusted, then it also ensures that this public key value belongs to the certificate’s owner.

**TLS/SSL services**

• **Peer authentication** is achieved by using digital certificates of one or both of the communication partners and verifying the authenticity of those certificates by using the public key of the CA that signed the certificates. The communication partners then use their own public or private keys to complete the authentication by encrypting and decrypting the contents of messages. They do this by agreeing on the set of cryptographic algorithms (the cipher suite) and the binary values that will eventually serve as symmetric encryption keys for protecting the application data that is sent over the secure SSL session. Peer authentication, cipher suite, and session-key agreement is completed during the first phase of the TLS/SSL protocol, which is called the SSL handshake.

• **Data privacy** is achieved by using a symmetric encryption algorithm and keys that were established during the SSL handshake.

• **Message authentication and integrity**: TLS/SSL uses other cryptographic methods, often based on hashing algorithms such as secure hash algorithms (SHA), to ensure that each message received over a secure connection originated from the correct communication partner (message authentication). TLS/SSL also ensures that those messages were not changed between the time they were sent and the time they were received (message integrity). Like data privacy, the message authentication and integrity protection is applied by the algorithms and keys that were established during the SSL handshake.

**How TLS/SSL works when used with HTTPS**

A browser requests a secure page (a URL beginning with https://), initiating a TLS/SSL handshake. During that handshake, the following steps happen:

1. The web server sends its digital certificate, which contains its public key.
2. The browser checks that the certificate was issued by a trusted CA, that the certificate is still valid, and that the certificate is related to the site contacted. Depending on the SSL configuration, the browser might also send its own certificate back to the server, so that the server can authenticate the browser.

3. The browser and server exchange messages to agree on the cipher suite (the set of cryptographic algorithms and key lengths) and the binary data that will eventually serve as the symmetric session keys (called keying material). During this exchange, the browser uses the server's public key to encrypt the message that contains the selected cipher suite and keying material.

4. The web server decrypts the cipher suite and keying material by using its private key, derives the symmetric session keys, and then uses those algorithms and keys to complete the handshake.

After the TLS/SSL handshake is complete, then the communicating systems use the agreed-upon cryptographic algorithms and session keys to provide data privacy, message authentication, and message integrity.

**SSL support in DB2 Connect**

DB2 Connect supports SSL. Command-line interface (CLI), command line processors (CLP), and .Net Data Provider client applications and applications that use the IBM Data Server Driver for JDBC and SQLJ (type 4 connections) also support SSL.

Some things to keep in mind about SSL and DB2 Connect:

- To encrypt data in transit between clients and DB2 databases, you can use the DATA_ENCRYPT authentication type or the DB2 database system support of Secure Sockets Layer (SSL).
- SSL is supported between clients and the HADR primary server. Clients who use SSL to connect to the HADR primary server are able to reroute to the HADR standby database by using SSL. However, SSL is not supported between the HADR primary and standby servers.

**What is keystore?**

A keystore is a protected database that stores the owner's private SSL key and digital certificates. Both the client and server must have the keystore that stores the digital certificates that are used in SSL communication. On most systems, this database is a file in the file system. On z/OS, a SAF-compliant key ring, such as that provided by RACF, often serves as the keystore. We will use this approach throughout the examples in this paper (although we reference RACF specifically, any SAF-compliant security manager can be used).

Each certificate in the keystore has a unique label.

If a label name is not specified, the default certificate in the keystore is used for SSL communication. Only one certificate in a keystore is the default certificate.

When the keystore resides in the file system:
• It is protected by a password to retrieve the private key and digital certificate for SSL communications.
• The keystore password is stored in an encrypted form in a stash (.sth) file. The stash file can open the keystore without the need for an explicit password.
• The password for the keystore on the server is mandatory because this password protects the private key for the server.
• In many cases, the client does not have its own private key and certificate. Instead, the keystore on the client stores the root CA certificates of all servers to which the client is connecting. In these cases, a password for the keystore is optional.

Access to RACF key rings is controlled by using RACF permissions and does not use passwords or stash files.

**Configuration steps for SSL server authentication**

SSL server authentication allows a user to confirm a server's identity, but does not allow the server to confirm the client's identity. Server authentication provides sufficient authentication in many cases and is the most commonly used configuration.

SSL-enabled client software can use standard techniques of public-key cryptography to check that a server's certificate and public ID are valid and were issued by a certificate authority (CA) listed in the client's list of trusted CAs. This confirmation is important when a client needs to ensure that it is connected to the correct server. For example, when it is sending a credit card number over the network and wants to ensure that the connection is the actual merchant's website.

**Example of configuration for SSL server authentication**

Let's look at an example. The steps were verified by using the GSKit version 8.0.14.43 on the client and RACF along with DB2 for z/OS 10.01.0005 on the server.

Create a certificate authority certificate and export it to a file called USRT001.SVL224.CACERT at the server. The private key that is associated with this CA certificate will be used to sign the server's personal certificate. The certificate file will be sent to the client. The CA certificate can then be added to the client keystore database, so the client can verify the server's certificate during an SSL handshake. To create the CA certificate and export it to a file at the server, follow these steps.

**Run the following steps on the server:**

**Step 1.** Generate a certificate authority (CA) key pair and certificate. This is a self-signed CA certificate, so it is essentially its own root CA.

```
RACDCERT CERTAUTH GENCERT SUBJECTSDN(OU('SVL224ServerC')O('IBM') L('SVL') SP('CA') C('US')) NOTAFTER(DATE(2030-12-31)) SIZE(2048) KEYUSAGE(CERTSIGN) WITHLABEL('SVL224ServerCAC')
```

**Output: No errors when executed.**

**Step 2.** Generate a key pair and personal certificate for the DB2 server.
RACDCERT ID(SYSDSP) GENCERT SUBJECTSDN(CN('host1.ibm.com')OU('UTEC224') O('SVL224') C('US')) NOTAFTER(DATE(2030-12-31)) SIZE(2048) WITHLABEL('SVL224ServerCet') SIGNWITH(CERTAUTH LABEL('SVL224ServerCAC'))

**Output: No errors when executed.**

**Step 3.** Create the key ring and add the server's CA certificate.

RACDCERT ID(SYSDSP) ADDRING(DB2KEYRING_SAMPLE)
RACDCERT ID(SYSDSP) CONNECT(CERTAUTH LABEL('SVL224ServerCAC') RING(DB2KEYRING_SAMPLE)) TRUST
RACDCERT ID(SYSDSP) CONNECT(ID(SYSDSP) LABEL('SVL224ServerCet') RING(DB2KEYRING_SAMPLE) DEFAULT)

**Output: No errors when executed.**

**Step 4.** Display the contents of the label SVL224ServerCAC to verify that everything is fine.

RACDCERT CERTAUTH LIST(LABEL('SVL224ServerCAC'))

**Output (your certificate ID, serial number, and date/time stamps will vary)**
Label: SVL224ServerCAC
Certificate ID: 2QiJmZmDhZmjgeLl0/Ly9OKFmaWFmcPBw0BA
Status: TRUST
Start Date: 2014/07/25 00:00:00
End Date: 2030/12/31 23:59:59
Serial Number: 00
Issuer's Name: OU=SVL224ServerC.O=IBM.L=SVL.SP=CA.C=US
Subject's Name: OU=SVL224ServerC.O=IBM.L=SVL.SP=CA.C=US
Key Usage: CERTSIGN
Key Type: RSA
Key Size: 2048
Private Key: YES
Ring Associations:
Ring Owner: SYSDSP
Ring: DB2KEYRING_SAMPLE

**Step 5.** The following command creates the USRT001.SVL224 CACERT file that needs to be added to the client keystore database

RACDCERT CERTAUTH EXPORT(LABEL('SVL224ServerCAC')) DSN('USRT001.SVL224.CACERT')

**Run the following steps on the client:**

**Step 6.** Create the keystore database

gsk8capicmd_64 -keydb -create -db host1.kdb -pw password -stash -fips

Get the server certificate from the server.
FTP host2.ibm.com
FTP> ascii
FTP> mget USRT001.SVL224.CACERT
FTP> quit

Step 7. Add this server certificate into the client kdb.

gsk8capicmd_64 -cert -add -db host1.kdb -pw password -file "USRT001.SVL224.CACERT"
-label "SVL224ServerCAC" -format ascii

Now make an SSL connection as shown below and the connection should succeed.

$db2cli
>$opt echo on
>$opt callererror on
>$opt callererror on
>$sqlallocenv 1
$sqlallocenv: rc = 0 (SQL_SUCCESS)
>sqlallocconnect 1 1
$sqlallocconnect: rc = 0 (SQL_SUCCESS)
>SQLDriverConnect 1 0
"DATABASE=STLEC1;PROTOCOL=TCPIP;PORT=448;HOSTNAME=host2.ibm.com;UID=SYSADM;
PwD=*******;security=ssl;SSL_CLIENT_KEYSTOREDB=host1.kdb;SSL_CLIENT_KEYSTASH=host1.sth" -3 255 SQL_DRIVER_NOPROMPT
>SQLDriverConnect 1 0
"DATABASE=STLEC1;PROTOCOL=TCPIP;PORT=448;HOSTNAME=host2.ibm.com;UID=SYSADM;
PwD=*******;security=ssl;SSL_CLIENT_KEYSTOREDB=host1.kdb;SSL_CLIENT_KEYSTASH=host1.sth"
-3 255 SQL_DRIVER_NOPROMPT
SQLDriverConnect: rc = 0 (SQL_SUCCESS)
SQLDriverConnect: ConnStrOut:
UID=SYSADM;PWd=*******;DATABASE=STLEC1;PROTOCOL=TCPIP;PORT=448;HOSTNAME=host2.ibm.com;
SECURITY=SSL;SSL_CLIENT_KEYSTOREDB=host1.kdb;SSL_CLIENT_KEYSTASH=host1.sth;
cbConnStrOut: 178
>$quit

Configuration steps for SSL client authentication

SSL client authentication is an extension of SSL server authentication. SSL client authentication allows the server to confirm the client's identity after the client confirms the server's identity.

SSL client authentication uses the same techniques as the techniques used for server authentication. With these techniques, SSL-enabled server software checks that a client's certificate and public key are valid and were issued by a certificate authority (CA) listed in the server's list of trusted CAs.

This confirmation is important in transactions that require confidentiality. For example, if the server is a bank that's sending confidential financial information to a customer and wants to check the recipient's identity.

SSL client authentication is sometimes called mutual authentication since it is a two-way authentication between the client and the server.
The following example shows the commands that are used to create the certificates directly on the client and server platforms. The method used to create and obtain the client and server certificates varies according to the customer's PKI requirements and infrastructure. Also, the example uses a self-signed client certificate, so this example is for illustration purpose only. In customer scenarios, certificates that are generated or signed by a separate CA might be used.

**Figure 1. Creating the certificates directly on the clients and servers**

The following steps were verified by using the GSKit version 8.0.14.43 on the client and RACF, along with DB2 for z/OS 10.01.0005 on the server.

To create a certificate file named USRT001.SVL224.CACERT at the server that will be added in the client database, follow the steps below.

**Run the following steps on the server.**

**Step 1. Generate a certificate authority (CA) certificate**

```
RACDCERT CERTAUTH GENCERT SUBJECTSDN(OU('SVL224ServerC') O('IBM') L('SVL') SP('CA')C('US')) NOTAFTER(DATE(2030-12-31)) SIZE(2048) WITHLABEL('SVL224ServerCAC') KEYUSAGE(CERTSIGN)
```

**Output:** No errors when executed.

**Step 2. Generate a key pair and personal certificate for the DB2 server**

```
RACDCERT ID(SYSDSP) GENCERT SUBJECTSDN(CN('host1.ibm.com') OU('UTEC224') O('SVL224') C('US')) NOTAFTER(DATE(2030-12-31)) SIZE(2048) WITHLABEL('SVL224ServerCet') SIGNWITH(CERTAUTH LABEL('SVL224ServerCAC'))
```

**Output:** No errors when executed.

**Step 3. Create the key ring and add the server’s CA certificate.**

```
RACDCERT ID(SYSDSP) ADDRING(DB2KEYRING_SAMPLE)
RACDCERT ID(SYSDSP) CONNECT(CERTAUTH LABEL('SVL224ServerCAC') RING(DB2KEYRING_SAMPLE)) TRUST
RACDCERT ID(SYSDSP) CONNECT(ID(SYSDSP) LABEL('SVL224ServerCet') RING(DB2KEYRING_SAMPLE) DEFAULT)
```
Output: No errors when executed.

**Step 4.** Display the contents of the label SVL224ServerCAC to verify that everything is fine.

```
RACDCERT CERTAUTH LIST('SVL224ServerCAC')
```

**Output (your certificate ID, serial number, and date/timestamps will vary)**

Label: SVL224ServerCAC  
Certificate ID: 2QiJmZmDhZmjgeLi0/Ly9OKFmaWFmcPBw0BA  
Status: TRUST  
Start Date: 2014/07/25 00:00:00  
End Date: 2030/12/31 23:59:59  
Serial Number: 00  
Issuer's Name: OU=SVL224ServerC.O=IBM.L=SVL.SP=CA.C=US  
Subject's Name: OU=SVL224ServerC.O=IBM.L=SVL.SP=CA.C=US  
Key Usage: CERTSIGN  
Key Type: RSA  
Key Size: 2048  
Private Key: YES  
Ring Associations:  
Ring Owner: SYSDSP  
Ring: DB2KEYRING_SAMPLE

**Step 5.** To create the USRT001.SVL224.CACERT file that needs to be added to the client keystore database, issue the following command:

```
RACDCERT CERTAUTH EXPORT('SVL224ServerCAC') DSN('USRT001.SVL224.CACERT')
```

**Step 6.** Display the contents of the label SVL224ServerCet

```
RACDCERT ID(SYSDSP) LIST('SVL224ServerCet')
```

**Output (your certificate ID, serial number, and date/timestamps will vary)**

Label: SVL224ServerCet  
Certificate ID: 2Qbi6OLE4tfi5dPy8vTihZmlhZnDhaNA  
Status: TRUST  
Start Date: 2014/07/25 00:00:00  
End Date: 2030/12/31 23:59:59  
Serial Number: 01  
Issuer's Name: OU=SVL224ServerC.O=IBM.L=SVL.SP=CA.C=US  
Key Type: RSA  
Key Size: 2048  
Private Key: YES
Ring Associations:
Ring Owner: SYSDSP
Ring: DB2KEYRING_SAMPLE

Generate a self-signed client certificate and export it to a file called EC022.CLIENT1.SSCERTA. This file will be sent to the server and added to the server's key ring.

**Run the following steps on the client.**

**Step 7.** Create the client's keystore database

```
gsk8capicmd_64 -keydb -create -db host1.kdb -pw password -stash -fips
```

**Step 8.** Create the client's certificate

```
gsk8capicmd_64 -cert -create -db host1.kdb -pw password -label clntssl -dn "CN=host3.ibm.com,OU=fvt,O=ibm,C=ca" -default_cert yes -expire 1000 -fips -size 2048 -sigalg SHA256WithRSA
```

**Step 9.** Extract the certificate into a file name EC022.CLIENT1.SSCERTA.

```
gsk8capicmd_64 -cert -extract -db host1.kdb -pw password -label clntssl -target EC022.CLIENT1.SSCERTA -format ascii -fips
```

FTP the client certificate to the server in ASCII mode. To do so:

FTP host2.ibm.com > ascii > put EC022.CLIENT1.SSCERTA

Using the same FTP session, download the server certificate (also in ascii mode) and add it to the client keystore database:

> mget
> USRT001.SVL224.CACERT
> quit

**Step 10.** Add the server's CA certificate to the client keystore database.

```
gsk8capicmd_64 -cert -add -db host1.kdb -pw password -file "SVL224.CACERT" -label "SVL224ServerCAC" -format ascii
```

**The following steps take place at the server.**

**Step 11.** At the server, add the self-signed client certificate to RACF database.

```
RACDCERT CERTAUTH ADD('SYSADM.EC022.CLIENT1.SSCERTA') WITHLABEL('CLIENTSSC') TRUST
```

**Output:** The certificate that you are adding is self-signed. The certificate is added with TRUST status. ***

**Step 12.** Display the contents of the label CLIENTSSC to ensure that everything is fine
RACDCERT CERTAUTH LIST(LABEL('CLIENTSSC'))

Output (your certificate ID, serial number, and date/timestamps will vary).
Label: CLIENTSSC
Certificate ID: 2QiJmZmDhZmjgcPTycXV4+Liw0BA
Status: TRUST
Start Date: 2014/07/24 05:19:34
End Date: 2017/04/20 05:19:34
Serial Number: 0283E9AF54C79857
Key Type: RSA
Key Size: 2048
Private Key: NO
Ring Associations:
Ring Owner: SYSDSP
Ring: DB2KEYRING_SAMPLE

Step 13. Connect the self-signed client certificate to the server’s key ring.
RACDCERT CONNECT(CERTAUTH LABEL('CLIENTSSC') RING(DB2KEYRING_SAMPLE) USAGE(CERTAUTH)) ID(SYSDSP)
No output means that it was successful.

Step 14. Display the key ring to ensure that everything is fine.
RACDCERT LISTRING(DB2KEYRING_SAMPLE)ID(SYSDSP)

Ring:
DB2KEYRING_SAMPLE
Certificate Label Name Cert Owner USAGE DEFAULT
--------------------------------- ------------ -------- -------
SVL224ServerCAC CERTAUTH CERTAUTH NO
SVL224ServerCet ID(SYSDSP) PERSONAL YES
CLIENTSSC CERTAUTH CERTAUTH NO
...

Step 15. Use the following commands to create a certificate name filter (CNF) and activate the certificate name by refresh command. This filter will be used during the SSL handshake to determine which z/OS user ID is associated with the client certificate.
RACDCERT MAP ID(USRT001) SDNFILTER('CN=host3.ibm.com.OU=fvt.O=ibm.C=ca') WITHLABEL('IBMers') TRUST SETROPTS RACLIST(DIGTNMAP) REFRESH

Complete step 16 in the client.
**Step 16.** In the client machine, test the SSL connection. The connection and data retrieval should succeed.

```
$db2cli
opt callerror on
opt echo on
SQLAllocEnv 1
SQLAllocconnect 1 1
sqldriverconnect 1 0
"database=STLEC1;hostname=host2.ibm.com;port=448;SSLCLIENTLABEL=clntssl;
authentication=certificate;
SSLClientKeystoredb=host1.kdb;SSLClientKeystoreDBPassword=password;"
-3 500 sql_driver_noprompt
sqlallocstmt 1 1
sqlexecdirect 1 "select count(*) from sysibm.systables" -3
fetchall 1
>quit
```

**Output:** The client connects and gives the queried results.


**Configuration error scenarios and solutions**

**Scenario 1.** If the certificate name filter (created in Step 15) is not created correctly, you would get an error.

```
sqldriverconnect 1 0
"database=STLEC1;hostname=host2.ibm.com;port=448;SSLCLIENTLABEL=clntssl;
authentication=certificate;
SSLClientKeystoredb=host1.kdb;SSLClientKeystoreDBPassword=password;"
-3 500 sql_driver_noprompt
SQLDriverConnect: rc = -1 (SQL_ERROR)
SQLError: rc = 0 (SQL_SUCCESS)
SQLError: SQLState : 08001
fnativeError : -30081
szErrorSMsg : [IBM][CLI Driver] SQL30081N A communication error has been detected.
Communication protocol being used: "TCP/IP".
Communication API being used: "SOCKETS".
Location where the error was detected: "9.30.219.110".
Communication function detecting the error: "recv".
Protocol specific error code(s):"10054", "*", "*". SQLSTATE=08001
cbErrorMsg : 335
SQLERROR: rc = 100 (SQL_NO_DATA_FOUND)
```

The error message on the server would be as follows. The RC value is 5002.

```
0010 JOBNAME: VA1ADIST USERID: SYSDSP RULE: VA1ASecureServer RC: 5002
0010 Initial Handshake 7EAA4E18 7EB1A218 TLSV1.1 02
0010 DSNI511I @ DSNIENO TCP/IP CONVERSATION FAILED 982
0010 TO LOCATION::FFFF:9.77.67.176
```

For more details about SSL function return codes, [see the documentation](https://www.ibm.com/support/knowledgecenter) in the IBM Knowledge Center.
**Scenario 2.** You will get the following error if ASCII mode is not used when you send the certificate file from the client to the server and you try to add the certificate on the server.

```
RACDCERT CERTAUTH ADD('SYSADM.EC022.CLIENT1.SSCERTA') WITHLABEL('CLIENTSSC') TRUST
"The input data set does not contain a valid certificate."
```

**Scenario 3.** On the server, when ASCII mode is not used when you receive the file, the gsk8capicmd_64 command itself gave an error CTGSK3046W.

```
220 Connection will close if idle for more than 5 minutes.
User (host2.ibm.com:(none)): usrt001
331 Send password please.
Password: 230 USRT001 is logged on. Working directory is "USRT001."
FTP> bin
200 Representation type is Image
FTP> mget SVL224.CACERT
200 Representation type is Image
mget SVL224.CACERT? yes
200 Port request OK.
125 Sending data set USRT001.SVL224.CACERT
250 Transfer completed successfully.
FTP: 944 bytes received in 0.00Seconds 944.00Kbytes/sec.
FTP> bye
221 Quit command received. Goodbye.
```

```
C:\Program Files\ibm\gsk8\bin\gsk8capicmd_64 -cert -add -db
host1.kdb -pw password -file "SVL224.CACERT" -label "SVL224ServerCAC" -format
ascii
```

```
CTGSK3046W The key file "SVL224.CACERT" could not be imported.
```

**User response:** On the server, the file needs to be received in ASCII mode only.

**Use cases for SSL client authentication**

Many DB2 for z/OS users are currently converting to SSL client authentication to eliminate the need to store and manage database passwords on the server. By using the advanced client authentication features of AT-TLS and DB2 for z/OS, the mapping between the client's SSL certificate and an associated z/OS user ID eliminates the need to provide both a user ID and a password.

The use case for client authentication describes customers who want to use the certificate to authenticate connections to DB2 but that requires a different database authorization ID to earn the database privileges that are required to access data. So, customers will be able to deploy client certificates that connect only to DB2 but have no privileges to access data.

Certificate-based authentication removes the need to provide a password to authenticate to DB2. But this also means that there needs to be careful control over the certificates and associated private keys. In security-conscious implementations, the user would have a smart card that contains his or her certificate. The application would ask the user to insert his or her smart card into a reader and enter a PIN to unlock the smart card. That is how the certificate and private key are securely provided to the application for use in SSL handshakes.
Consider the keystore database as the smart card and the password for the keystore database as the PIN. This password is really the appropriate mechanism to protect the keystore database. The stash file's only purpose is to allow a server to boot without user intervention, but it is not appropriate for user authentication. There needs to be an option where the user specifies a password to unlock the keystore database because that would be the general use for user authentication.

The certificate-based authentication requires the user to provide the path and name to their keystore database and the authentication password to unlock that keystore database. A shared keystore database and shared password or stash file is not a good security design for user authentication.

**Associating user ID with certificate in DB2 for z/OS**

RACF offers a few ways to associate a user ID with a certificate. The two most commonly used methods are one-to-one certificate to user ID association and certificate name filtering.

- **One-to-one certificate to user ID association:** Whenever a certificate is generated or a previously generated certificate is stored in RACF, the certificate is registered to a user ID when it is added to the RACF database. This method establishes a direct one-to-one association, or mapping, between each certificate and one specific user ID. Registered certificates are stored in certificate profiles. These profiles contain an exact copy of the certificate and for user IDs on the system, the private key, if it exists. Certificates that are stored in this way can be used to simply associate a certificate with a user ID or they can be gathered into a collection, or key ring, for use by other applications as part of a secure network protocol.

- **Certificate name filtering:** A certificate name filter (CNF) allows you to associate many certificates with one user ID, based on rules concerning portions of the subject's or issuer's distinguished names in the certificate. Unlike the one-to-one association, using CNF allows you to establish mappings between a client's identity and a z/OS user ID without storing the client's certificate in the RACF database. This is important when you use a CA to sign your client certificates.

**V9.7 FP6 CLI enhancements to support SSL client authentication**

Earlier CLI supported two SSL parameters for specifying the location of the keystore db (database) and the stash file via SSLClientKeystoredb and SSLClientKeystash, respectively. As part of the enhancement, a parameter is exposed to specify the SSL label to uniquely identify certificate within keystore db. This is necessary so that certificates other than the default can be used from the keystore db. To specify the label (mapped to a specific certificate) that needs to be used for authentication, the data server driver (db2dsdriver.cfg) configuration parameter 'SSLClientLabel' (equivalently, SSLClientLabel keyword in db2cli.ini) can be used.

To allow the user to use SSL certificate as a mechanism to authenticate the client and to differentiate from the rest of the authentication methods that the client supports, the value ‘CERTIFICATE’ can be used in the db2dsdriver.cfg parameter ‘Authentication’.
If the 'Authentication' option has the new value 'CERTIFICATE', the presence of the new label parameter 'SSLClientLabel' is mandatory. Otherwise, an error would be returned (that is, no attempt is made to use the default label of the keystore db).

Regardless of whether the db2dsdriver.cfg attribute 'SecurityTransportMode'(equivalently, 'Security' keyword in db2cli.ini) is set or not, the 'Authentication' option with the new value 'CERTIFICATE' will be supported. So, if 'SecurityTransportMode' is set to 'SSL' and 'Authentication' is set to 'CERTIFICATE', then you can achieve data encryption along with encrypted authentication. Similarly, if 'SecurityTransportMode' is not set to 'SSL' and 'Authentication' is set to 'CERTIFICATE', certificate-based client authentication would happen without encrypting the data stream.

Authentication type CERTIFICATE will be supported by work load balancing (WLB) or automatic client reroute (ACR) enabled. It will also be supported by alternate groups if the alternate group configured the certificate to be the same as with the primary group.

Specifying SSL keystore db password via existing client's SSLClientKeystash keyword forces users to keep the stash file on the file system. For many customers, this is a security issue, as having access to this stash file can virtually enable any user to connect to the database by using someone else’s SSL certificate. To avoid this problem, you can provide an alternate mechanism to specify the keystore db password as opposed to storing something in the file system. To have broader control over the certificate, a new db2dsdriver.cfg parameter 'SSLClientKeystoreDBPassword' (equivalently, 'SSLClientKeystoreDBPassword' keyword in db2cli.ini and a connection string) can be used. The parameter 'SSLClientKeystoredb' provides the path and the name of the keystore db. The 'SSLClientKeystoreDBPassword' parameter will be mutually exclusive with 'SSLClientKeystash' parameter. The specification of this value in the db2dsdriver.cfg or db2cli.ini file could lead to a security exposure.

**Specifying a user ID with authentication-based certification**

If the authentication parameter is set to CERTIFICATE, you can choose to specify the user ID information during the initial connection through the db2dsdriver.cfg parameter 'UserID' or through db2cli.ini parameter 'UID' or through the connection string keyword.

If "userID" is not specified, then the user ID associated with the SSL certificate will be used as a regular user ID for rest of the connection.
Figure 2. Scenario when user ID is unspecified

If the user ID information is specified in this scenario (X), then a connected server will attempt to make an implicit trusted connection if there is trust between this user ID (X) and the user ID (Y) associated with the CERTIFICATE. If the connection requires an authentication, the server will reject the connection request (SQL20361N with reason code '1' from the server will be returned to the application). After the implicit trusted context, specified user ID (X) will be active for rest of the connection.

Figure 3. Scenario when user ID is provided

For a trusted context to occur, no other configuration is expected in CLI. For example, existing SQL_ATTR_USE_TRUSTED_CONTEXT attribute (which is used for explicit trusted context) is irrelevant in this context.
Specifying a password with authentication-based certification

Because the authentication is based on a certificate, specifying "password" in db2dsdriver.cfg (or PWD for db2cli.ini file or in the connection string) is an invalid combination and an SQL error will be returned to the application.

[Explicit] Trusted context behavior when authentication is CERTIFICATE

- Once the connection that uses authentication='CERTIFICATE' is established, the existing (explicit) trusted context behavior continues to behave as is (by using SQL_ATTR_TRUSTED_CONTEXT, SQL_ATTR_TRUSTED_CONTEXT_USERID, or their equivalent db2dsdriver.cfg and connection string keywords).
- However, if the SQL_ATTR_TRUSTED_CONTEXT_PASSWORD is also specified, -30082N with reason code 41 would be returned and the connection will be dropped.

The following is a sample db2dsdriver.cfg file.

```
<configuration>
    <dsncollection>
        <dsn alias="DSN1" name="ACO1" host="host1.ibm.com" port="5912">
            <parameter name="SSLClientKeystash" value="/home/test1/keystore/clientstore.sth"/>
        </dsn>
        <dsn alias="DSN2" name="ACO2" host="host2.ibm.com" port="5912/">
        </dsn>
        <dsn alias="DSN3" name="ACO3" host="host3.ibm.com" port="5912/">
        </dsn>
    </dsncollection>

    <database name="ACO1" host="host1.ibm.com" port="5912">
        <parameter name="Authentication" value="CERTIFICATE"/>
        <parameter name="UserID" value="userid"/>
        <parameter name="SSLClientLabel" value="labelname1"/>
    </database>

    <database name="ACO2" host="host2.ibm.com" port="5912">
        <parameter name="Authentication" value="CERTIFICATE"/>
        <parameter name="UserID" value="userid"/>
        <parameter name="SSLClientKeystoreDBPassword" value="password2"/>
        <parameter name="SSLClientLabel" value="labelname2"/>
    </database>

    <database name="ACO3" host="host3.ibm.com" port="5912">
        <parameter name="Authentication" value="CERTIFICATE"/>
        <parameter name="UserID" value="userid"/>
        <parameter name="SSLClientLabel" value="labelname1"/>
    </database>

    <parameters>
        <parameter name="SSLClientKeystoredb" value="/home/test1/keystore/clientstore.kdb"/>
    </parameters>
</configuration>
```

In the above db2dsdriver.cfg file, the following points are true:

- For DSN1, sslclientKeystash is used to obtain an encrypted password to the key database file.
• For DSN2, SSLClientKeystoreDBPassword alone is used for providing the password for
keystore database (It is not recommended to provide a password in db2dsdriver.cfg for
security reasons. This example is just for explanation purpose.
• For DSN3, SSLClientKeystoreDBPassword can be passed through the
connection string as shown below. This approach is recommended for
providing the SSL client's password during runtime. SQLDriverConnect
(......"DSN=DSN3;SSLClientKeystoreDBPassword=password;........")
• The uppercase value 'CERTIFICATE' for the 'Authentication' attribute is used for illustration
purpose. The value is not case-sensitive.

Restrictions

• The proposed authentication mechanism will work between the client and z/OS server as
a direct connection. This mechanism will not be supported through a gateway between the
client and z/OS server. In a scenario where the DB2 Connect Server is present between the
client and z/OS server and the Authentication field is set to "Certificate," -30082 with reason
code 15 will be given to the application. Currently, the DB2 Connect Server is not supported.
• The database directory will not be supported. Applications must use the connection string
dsdriver.cfg or cli.ini to specify this authentication mechanism.
• This support is being added only to CLI / ODBC. Other APIs like embedded SQL and CLP are
not being supported.

Other important notes:

• Certificate-based authentication is supported on either SSL or TCPIP.
• The DB2 client side always initializes GSKit in FIPS mode. On SSL, we are operating in FIPS
mode. On TCPIP, we are not operating in FIPS mode.
• What is referred to as "SSL" is the scenario where an SSL connection is requested. The data
flow between the client and server will be encrypted. What is referred to as "TCP/IP" is the
scenario where a regular non-SSL connection is requested. No encryption of the data flow
will take place. In both cases, when certificate-based authentication is involved, the server
authenticates the client certificate by using TLS. Intuitively, certificate-based authentication
without data encryption (what we refer to as "TCP/IP") would perform better.
• If the keyword "Security" is not set to "SSL," it means that certificate-based authentication is
provided without encrypting the data stream. If "Security" is set as "SSL," the data would also
get encrypted.
• DB2 for Linux, UNIX, and Windows (LUW) does not retrieve the certificate from GSKit. DB2
for LUW provides the label to the certificate to GSKit. GSKit does not immediately validate
if the label exists in the key database. GSKit uses the label to find the certificate in the key
database during the SSL client authentication handshake.

SSL client authentication behavior with automatic client reroute (ACR)

When a server or client (primary to begin with) goes down, the connection fails over to the next
member on the alternate server list. When this happens, the usual authentication=CERTIFICATE
hand-shaking takes place similar to that of the main primary connection. This could be an
overhead (especially in the case of a single-cluster environment) where necessary trust is already
established between the client and server during initial connection itself.
Error scenarios and solutions when wrong values are given to the connection keywords

Refer to the following link for description on the SSL function return codes: http://www-01.ibm.com/support/knowledgecenter/?lang=en#!/SSLTBW_2.1.0/com.ibm.zos.v2r1.gska100/sssl2msg1000885.htm,

Scenario 1. The label is wrong.
The example below shows an error that is returned when the value for SSLCLIENTLABEL has extra characters.

```
sqldriverconnect 1 0
"database=STLEC1;hostname=host2.ibm.com;port=448;
SSLCLIENTLABEL=clntssl1;authentication=certificate;SSLClientKeystoredb=host1.kdb;
SSLClientKeystoreDBPassword=password;"
-3 500 sql_driver_noprompt
SQLDriverConnect: rc = -1 (SQL_ERROR)
SQLError: rc = 0 (SQL_SUCCESS)
SQLState : 08001
fNativeError : -30081
szErrorMsg : [IBM][CLI Driver] SQL30081N
A communication error has been detected. Communication protocol being used: "SSL".
Communication API being used: "SOCKETS".
Location where the error was detected: "".
Communication function detecting the error: "sqlccSSLSocketSetup".
Protocol specific error code(s): "406", "*", "*".
SQLSTATE=08001
```

Scenario 2. The label is wrong.
For example, the following error message occurs when the value for SSLCLIENTLABEL is truncated.

```
sqldriverconnect 1 0
"database=STLEC1;hostname=host2.ibm.com;port=448;
SSLCLIENTLABEL=clntssl;authentication=certificate;SSLClientKeystoredb=host1.kdb;
SSLClientKeystoreDBPassword=password;"
-3 500 sql_driver_noprompt
SQLDriverConnect: rc = -1 (SQL_ERROR)
SQLError: rc = 0 (SQL_SUCCESS)
SQLState : 08001
fNativeError : -30081
szErrorMsg : [IBM][CLI Driver] SQL30081N
A communication error has been detected. Communication protocol being used: "SSL".
Communication API being used: "SOCKETS".
Location where the error was detected: "".
Communication function detecting the error: "sqlccSSLSocketSetup".
Protocol specific error code(s): "420", "*", "*".
SQLSTATE=08001
```

Scenario 3. The SSLClientKeystoreDBPassword value is wrong.
Configuring SSL for IBM Data Server Driver for ODBC and CLI

Scenario 4. The SSLClientKeystoredb value is not wrong.

```
sqlDriverConnect 1 0
"database=STLEC1;hostname=host2.ibm.com;port=448;
SSLCLIENTLABEL=clntssl;authentication=certificate;
SSLClientKeystoredb=host1.kdb;SSLClientKeystoreDBPassword=WWpassword;"
-3 500 sql_driver_noprompt
SQLDriverConnect: rc = -1 (SQL_ERROR)
SQLError: rc = 0 (SQL_SUCCESS)
SQLState : 42724
fNativeError : -1109
szErrorMsg : [IBM][CLI Driver] SQL1109N
The specified DLL "GSKit Error: 408" could not be loaded. SQLSTATE=42724
```

cbErrMsg : 102

Scenario 5. The port value is wrong.

```
sqlDriverConnect 1 0
"database=STLEC1;hostname=host2.ibm.com;port=447;
SSLCLIENTLABEL=clntssl;authentication=certificate;
SSLClientKeystoredb=host1.kdb;SSLClientKeystoreDBPassword=WWpassword;"
-3 500 sql_driver_noprompt
SQLDriverConnect: rc = -1 (SQL_ERROR)
SQLError: rc = 0 (SQL_SUCCESS)
SQLState : 08001
fNativeError : -30081
szErrorMsg : [IBM][CLI Driver] SQL30081N
A communication error has been detected.
Communication protocol being used: "SSL."
Communication API being used: "SOCKETS."
Location where the error was detected: "".
Communication function detecting the error: "sqlccSSLsocketSetup."
Protocol specific error code(s): "410", ",", ",".
SQLSTATE=08001
```

cbErrMsg : 333

Scenario 6. An invalid value is provided for the authentication keyword
sqlDriverConnect 1 0
"database=STLEC1;hostname=host2.ibm.com;port=448;
SSLCLIENTLABEL=clntssl;authentication=certificate;
SSLClientKeystoredb=host1.kdb;SSLClientKeystoreDBPassword=password;"
-3 500 sql_driver_noprompt
SQLDriverConnect: rc = -1 (SQL_ERROR)
SQLError: rc = 0 (SQL_SUCCESS)
SQLError: SQLState : HY000
szErrorMsg : [IBM][CLI Driver] CLI0222E
Authentication failed because the SSLClientLabel parameter was specified
but the DB2 client or data server driver was not configured to use
certificate-based authentication.
SQLSTATE=HY000
cbErrorMsg : 216
SQLError: rc = 100 (SQL_NO_DATA_FOUND)

Scenario 7. A wrong value is provided for the database keyword.

sqlDriverConnect 1 0
"database=STLE1;hostname=host2.ibm.com;port=448;
SSLCLIENTLABEL=clntssl;authentication=certificate;
SSLClientKeystoredb=host1.kdb;SSLClientKeystoreDBPassword=password;"
-3 500 sql_driver_noprompt
SQLDriverConnect: rc = -1 (SQL_ERROR)
SQLError: rc = 0 (SQL_SUCCESS)
SQLError: SQLState : 08004
fNativeError : -30061
szErrorMsg : [IBM][CLI Driver] SQL30061N
The database alias or database name "STLE1 " was not found at the remote node.
SQLSTATE=08004

SSL connection process in 10.5 FP5

Starting from 10.5 FP5, the SSL connection process is simplified.

As you saw in the previous steps, to configure SSL support at DB2 client and server side, you
need to download the GSKit package from IBM's website, install it, and use the GSKCapiCmd tool
provided by IBM Global Security Kit (GSKit). When we use client drivers for DB2 cloud service
offerings, downloading and installing GSKit is a cumbersome process. It is difficult to manage key
databases and create SSL connections using the GSKCapiCmd tool/executable.

Because of this, the SSL connection process for instance-based clients is simplified so that an
application can set SECURITY=SSL, and the SSL connection to server should work fine. Also, CLI
driver started accepting self-signed certificate files for SSL connection from a connection string.

By default, a DB2 server installer comes with a full GSKit package. But client installers (both
instance-based and non-instance based clients) come only with GSKit crypto-library. This is used
for client-side encryption. To remove the dependency on installation of separate GSKit package
for SSL connection, the required GSKit libraries for client packages that are required for each
application process are included as part of the package.
So, with the new solution, to connect to a database over SSL by using CLI, you don't need to download and install separate GSKit packages to configure SSL. Applications don't need to create key databases or stash files by using the GSKCapiCmd tool. Applications don't need to pass value for SSLClientKeystoredb or SSLClientKeystoreDBPassword or SSLClientKeyStash keywords. The necessary steps to establish the SSL connection will be handled programmatically. However, you can still create your own key database if you want to use the existing mechanism of establishing the SSL connection by using the configuration keywords.

For every process, during the allocation of first environment handle, CLI will internally generate a unique default key database name and corresponding key database password by using application process ID and store them in global variables. On the request of a connection to database, if “SECURITY=SSL” is set and the user did not provide a value for SSLClientKeystoredb keyword, the default key database is used as the value of SSLClientKeystoredb keyword. And the default key database password is used as the value of SSLClientKeystoreDBPassword keyword.

If the DB2 Server is using a self-signed certificate, applications need to pass “Security=SSL;SSLServerCertificate=(certificate_file_path)” to CLI. The SSLServerCertificate is a new connection string keyword. CLI will first add this SSLServerCertificate file into key databases and then proceed for SSL connection. The new keyword SSLServerCertificate can be used to pass both server's self-signed certificate or server's CA certificate. So, this method can be used to establish SSL connections by using a server’s certificates signed by trusted CAs whose CA certificate is not present in the pre-populated GSKit CA certificate list.

SSLServerCertificate can be used with an existing key database, either created by the user with the GSKCapiCmd tool or as a default key database created by the CLI driver during the first SSL connection for that application process. To add a certificate in a key database, you must pass a unique LABEL for the certificate. The client driver will use the path of the certificate file passed as the value of SSLServerCertificate keyword as LABEL while adding this SSLServerCertificate to key database. This step ensures that a certificate at a particular location in the client system will be stored only once in the key database.

The name of the default key database file that is formed by the driver code will be in the format: client_(_PID).kdb

Storage for created key database file on disc using GSKit Key Management APIs:

- For instance-based clients: under the (instance_path)/cfg/ directory.
- For instance-less clients: under the (install_path)/cfg directory.

All threads of a process will use the same SSLClientKeystoredb and SSLClientKeystoreDBPassword for SSL connection. The created default key database files will exist throughout the life of application process and are deleted when the process exits gracefully. In case of abnormal termination of an application's process, no cleanup will happen and key database files will remain on disk. You can manually delete such files if there is no active application process.
SSL connection process in IBM DashDB

IBM provides the DashDB and the SQLDB services through an IBM Cloud offering. All the DashDB and SQLDB instances that are created through the IBM Cloud offering will use the same certificate: DigiCert Global Root CA Certificate certified by DigiCert Inc for the TCP/IP with SSL connections. Currently, all DashDB and SQLDB users need to log in through a web console and download the CA certificate to use in the application for SSL connections. You need to give the same certificate in the connection string through SSLServerCertificate keyword.

IBM DashDB and SQLDB services uses the DigiCert Global Root CA Certificate DigiCertGlobalRootCA.arm certified by DigiCert Inc for SSL connections. To ease the method to use the SSL connections for IBM DashDB and SQLDB, DB2 CLI driver bundles this CA certificate in all the client packages. DB2 CLI driver adds this certificate to the default key db created when an application does not specify the user-created keystore db through a SSLClientKeystoredb parameter for SSL connection. Using this certificate, TCP/IP with SSL connections will succeed to IBM DashDB and SQLDB services because this is the default certificate used by them.

If the user specifies the keystoredb through SSLClientKeystoredb keyword, the DigiCert will not be added to the keystoredb, so the DigiCert will not be used. The connection will be attempted only by using the certificates present in the user specified keystoredb.

DB2 CLI packages the DigiCert Global Root CA certificate (DigiCertGlobalRootCA.arm) with the same name in following locations.

- For instance-based clients: Under (instance_path)/cfg/ directory.
- For instance-less clients: Under ()install_path)/cfg directory.

With this certificate bundled, all CLI applications can specify Security=SSL in the connection string or SecurityTransportMode=SSL in the db2dsdriver.cfg to connect to database server over TCP/IP with SSL. Applications don't need to specify the certificate through the SSLServerCertificate keyword.

Acknowledgements

Thanks to Bimal K Jha and Sujan S. Ghosh for helping us complete the article.
Related topics

- Read Configuring Secure Sockets Layer (SSL) support in a DB2 instance to learn about SSL and DB2.
- Check out Configuring Secure Sockets Layer (SSL) support in non-Java DB2 clients.
- See the in-depth documentation about how to Configure TLS/SSL for secure client/server communications.
- Download the GSKit from Passport Advantage.
- For more information about the GSKit tool GSKCapiCMD, see the GSKCapiCmd User's Guide.
- Learn more about IBM Global Security Toolkit return codes.