SSL With Oracle JDBC Thin Driver

An Oracle Technical White Paper
April 2010

Author: Jean de Lavarene

ORACLE

Introduction....4
1. What SSL gives you....4
2. SSL settings overview....5
2.1. On server....5
2.2. On JDBC Thin driver....7
2.2.1. Which jars are required?....7
2.2.2. How to activate SSL?....7
2.2.3. Which properties are involved?....8
2.2.4. How can Oracle wallets be used in Java?....9
2.2.5. What cipher suites can be used?....10
Case #1: Use SSL for encryption only....11
Server configuration....11
JDBC Thin client configuration....11
Case #2: Use SSL for encryption and Server Authentication....12
Server configuration....12
JDBC Thin client configuration....12
If the truststore format type is JKS....12
If the truststore is a wallet....12
Check the Server's Distinguished Name....13
Case #3: Use SSL for encryption and authentication of both tiers....13
Server configuration....13
JDBC Thin client configuration....14
If the keystore format type is JKS....14
If the keystore is a wallet....14
Case #4: Use SSL as an authentication service in the Database....15
Server configuration....15
JDBC Thin client configuration....15
What's new in 11.1.0.7?....16
Meaningful error messages....16
Information about the cipher suite used....16
Better handling of the PKCS12 provider for wallets....16
Possibility to use the "oracle.net.wallet_location" property....16
Conclusion....17
Appendix A Troubleshooting....17
Appendix B Creating truststores and keystores....20
Using orapki....20
Create a wallet for the test CA....20
Create a wallet for the Oracle server....20

For the client (proceed the same way as for the server) 22
To create a wallet that contains only the trusted certificate 22
Using keytool 22
Create a JKS keystore 22
Create a JKS truststore 24

INTRODUCTION

Oracle Advanced Security (OAS) is an Oracle Database Enterprise Edition Option.

Oracle Advanced Security (OAS) contains a comprehensive suite of security features that protect enterprise networks and securely extend them to the Internet. It provides a single source of integration with multiple network encryption, data integrity and authentication solutions, single sign-on services, and security protocols.

Because Oracle databases contain sensitive information (employee and financial records, customer orders, product information, etc.) and because of the security threats (eavesdropping and data theft, data tampering, falsifying user identities) security is a concern and OAS offers solutions to protect your database.

For data encryption and data integrity, you can configure either Oracle Net native encryption (for example AES and SHA1 at the Oracle Net layer) or Secure Sockets Layer (SSL). OAS also provides a choice of several strong authentication methods, including Kerberos, Radius, and digital certificates.

Refer to the Java Secure Socket Extension (JSSE) Reference Guide for details about SSL and its support in Java.

This paper explains how to use SSL when the network client tier software is the Oracle JDBC Thin driver. The readers should be familiar with SSL and the JDBC Thin driver. For other security features available in the Oracle JDBC Thin driver, please refer to the *JDBC Developer's Guide and Reference*.

The products versions for both the Database and the driver covered in this paper are 10.2.0.3 (10*g* Release 2), 11.1.0.6 (11*g*) and 11.1.0.7. We also assume that the Java version is Java SE 5 or 6.

For more information about how to configure the OAS options on the Database server, you can read the *Advanced Security Administrator's Guide* of the Database documentation.

1. WHAT SSL GIVES YOU

Secure Sockets Layer (SSL) is an industry standard protocol for securing network connections. SSL uses RSA public key cryptography in conjunction with symmetric key cryptography to provide authentication, encryption, and data integrity.

Authentication is accomplished through a certificate authority (CA), which is a third party that is trusted by both of the communication parties.

By using Oracle Advanced Security SSL functionality to secure communications between JDBC Thin clients and Oracle servers, you can:

- Encrypt the connection between clients and servers.
- Authenticate the network client tier: the Database server only accepts connections from clients, or mid-tiers such as the Oracle Application Server, which have a certificate signed by a trusted authority. Any connection attempt from a client tier or an application that the Database doesn't trust will fail.
- Authenticate the Database tier: the JDBC Thin driver can be configured to validate the Database's certificate. If it hasn't been signed by a trusted authority, the connection will fail. From the application standpoint, you have proof that the Database can be trusted.
- Use SSL as an Authenticate Service on the server (starting in 11.1.0.6 for the JDBC Thin driver): the Database user, as opposed to the network client tier, is authenticated through SSL. In this case each Database user must have his own valid certificate.

Note that you can use SSL features by themselves or in combination with other authentication methods supported by Oracle Advanced Security. For example, with the JDBC Thin driver you can use the encryption provided by SSL in combination with the authentication provided by Kerberos (starting in 11.1.0.6).

SSL support in the JDBC Thin driver was first included in the 10*g* Release 2 of the driver. Support for SSL as an authentication service with the Oracle Database was first supported in the 11*g* Release 1 of the driver.

The JDBC Thin driver uses the Java Secure Socket Extension (JSSE) defined by Sun. Sun's provider for JSSE, called SunJSSE, is used by default by the Thin driver but you could use any other provider (PKI or SSL provider). For more details please read the *JSSE Reference Guide*.

2. SSL SETTINGS OVERVIEW

This section provides details of the settings that are specific to SSL.

2.1. On server

First of all, the listener must be configured to use the TCPS protocol:

```
LISTENER = (ADDRESS_LIST=
 (ADDRESS=(PROTOCOL=tcps)(HOST=servername)(PORT=2484))
)
```

A wallet is a password-protected container that is used to store authentication and signing credentials, including private keys, certificates, and trusted certificates required by SSL. You can use Oracle Wallet Manager to create a wallet.

The server's auto-login wallet location must be provided in both sqlnet.ora and listener.ora. In the most common case, both files contain the same wallet location but this is not necessarily the case, the listener could use its own wallet. For the sake of simplicity, in this paper, we consider that both sqlnet.ora and listener.ora use the same wallet location.

```
WALLET_LOCATION=(SOURCE=(METHOD=FILE)(METHOD_DATA=(DIRECTORY=/server/wa
llet/path/)))
```

Finally client authentication can be turned on or off. By default it's on.

```
SSL_CLIENT_AUTHENTICATION=FALSE
```

Or

```
SSL_CLIENT_AUTHENTICATION=TRUE
```

This setting applies to both listener.ora and sqlnet.ora. If SSL client authentication is turned on, then the JDBC Thin driver must be configured to send the client's digital certificate that must be accepted by the server otherwise the connection will fail.

Note that you must always provide the wallet location on the server even if you do not use SSL authentication at all (i.e. you use SSL for encryption only). For more information on how to create a wallet, please refer to the *Advanced Security Administrator's Guide* of the Database documentation. This paper also explains how to use the orapki utility to create wallet and certificates for testing purposes (see section Appendix B).

You can prioritize the cipher suites. When the client negotiates with the server regarding which cipher suite to use, it follows the prioritization you set.

You can also optionally set the cipher suite on the server, in sqlnet.ora, if you want to use a subset of the available cipher suites. The server supports the following cipher suites:

- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_RC4_128_SHA
- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_DES_CBC_SHA
- SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
- SSL_DH_anon_WITH_RC4_128_MD5
- SSL_DH_anon_WITH_DES_CBC_SHA
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
- **SSL_**RSA_WITH_AES_128_CBC_SHA
- **SSL_**RSA_WITH_AES_256_CBC_SHA

The Transport Layer Security (TLS) protocol is based on SSL, but has a different initial handshake protocol and is more extensible.

The last two ciphers use the TLS protocol and in Java are named TLS_xxx instead of SSL_xxx (see section 2.2.5). For more details about the meaning of these ciphers, please read the *Advanced Security Administrator's Guide.*

Prioritize cipher suites starting with the strongest and moving to the weakest to ensure the highest level of security possible.

For example, to configure the server to accept only connections using either `SSL_RSA_WITH_AES_128_CBC_SHA` or `SSL_DH_anon_WITH_3DES_EDE_CBC_SHA`, in sqlnet.ora you would add:

```
SSL_CIPHER_SUITES=(SSL_RSA_WITH_AES_128_CBC_SHA,
                   SSL_DH_anon_WITH_3DES_EDE_CBC_SHA)
```

With such a setting, if the network client does not want to use SSL authentication, it will have to use `SSL_DH_anon_WITH_3DES_EDE_CBC_SHA` otherwise, it will use `SSL_RSA_WITH_AES_128_CBC_SHA`.

2.2. On JDBC Thin driver

2.2.1. Which jars are required?

ojdbc14.jar from 10*g* R2 is compiled with JDK1.4 but can be used with Java SE 5 or 6 (for incompatibility details, please refer to the Java SE documentation).

With 10.2.0.3

The JDBC jar can be found in $ORACLE_HOME/jdbc/lib:

- ojdbc14.jar

If you need the Oracle PKI provider (if you use wallets on the client), you also need the following jars ($ORACLE_HOME/jlib):

- oraclepki.jar
- ojpse.jar

With 11.1.0.x

You need one of the following JDBC jars (from $ORACLE_HOME/jdbc/lib) depending on your Java SE version:

- ojdbc5.jar (compiled with Java SE 5)
- ojdbc6.jar (compiled with Java SE 6)

If you need the Oracle PKI provider (if you use wallets on the client), you also need the following jars ($ORACLE_HOME/jlib):

- oraclepki.jar
- osdt_cert.jar
- osdt_core.jar

2.2.2. How to activate SSL?

First of all the JDBC URL must use the "tcps" protocol in order to activate SSL in the JDBC Thin driver.

For example the following URL activates SSL:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=servername
)(PORT=2484))(CONNECT_DATA=(SERVICE_NAME=servicename)))
```

2.2.3. Which properties are involved?

The following properties may need to be set depending on the functionality of SSL you want to use. Note that these properties can be set either through connection properties or system properties.

If you want to use any cipher suite other than the ones that use Diffie-Hellman anonymous authentication then you must provide a "truststore" which is used to verify that the certificate coming from the server is trusted. A "truststore" does not contain any private keys; instead it contains trusted certificate entries, including Certification Authority (CA) certificates. The following JSSE properties are involved to set the "truststore":

- `javax.net.ssl.trustStore`
- `javax.net.ssl.trustStoreType`
- `javax.net.ssl.trustStorePassword`

If client authentication is enabled on the server, then you must provide a "keystore". The "keystore" contains the client's certificate. The following JSSE properties are involved to set the "keystore":

- `javax.net.ssl.keyStore`
- `javax.net.ssl.keyStoreType`
- `javax.net.ssl.keyStorePassword`

To enable a subset of the cipher suites available by default, use the following property:

- `oracle.net.ssl_cipher_suites`

This next property can be used to force the driver to verify that the server's DN matches:

- `oracle.net.ssl_server_dn_match`

Finally to activate SSL as an authentication service in the Database, the following property is used (introduced in 11.1):

- `oracle.net.authentication_services`

More details about how to use these properties will be provided in the next sections.

> **Note:** How about the "`oracle.net.wallet_location`" property? In 11.1.0.6 and 10.2.0.3, there are bugs and limitations related to using this property. For example under MS Windows, you cannot provide the driver letter in the wallet location. In this paper, we assume that this property is not used. See the section "What's new in 11.1.0.7?" for more information about this property.

2.2.4. How can Oracle wallets be used in Java?

Wallets created by Oracle Wallet Manager or "orapki" (see Appendix B) use the standard PKCS12 format to store X.509 certificates and private keys. The wallet is stored in a file named "ewallet.p12".

Unfortunately there are some incompatibilities with the PKCS12 implementation provided by Sun. Consequently, you must use Oracle's PKI provider, named "OraclePKI", to access Oracle wallets from Java.

If you enable auto-login in the wallet, an obfuscated copy of the wallet is created in the file "cwallet.sso" which can then be used without providing the password. This auto-login wallet format is used in the Oracle Single Sign On infrastructure thus the extension "sso". To access auto-login wallets (also called SSO wallets in this paper) from Java, you also need to use Oracle's PKI provider.

In JSSE, there are two ways to enable a provider: dynamically and statically.

You must enable Oracle's PKI provider if you use wallets.

Enabling Oracle's PKI provider statically

If you use SSO wallets (cwallet.sso), you can statically enable Oracle's PKI provider by adding it at the end of the provider list in the file java.security (this file is part of your JRE install) which would typically look like:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=com.sun.security.sasl.Provider
security.provider.7=oracle.security.pki.OraclePKIProvider
```

Note that if another provider supports the type "SSO", then it should be inserted after Oracle's PKI provider.

If you use PKCS12 wallets (ewallet.p12), you have to move Oracle's PKI provider to position #3 or any position ahead of Sun's provider which also supports PKCS12 but which is not compatible with Oracle's wallets. So the list of providers in java.security would look like:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=oracle.security.pki.OraclePKIProvider
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
security.provider.7=com.sun.security.sasl.Provider
```

Unless you are using the 11.1.0.7 (and onwards) JDBC thin driver, before creating a new connection, you also need to instantiate `OraclePKIProvider` so that the class gets loaded and initialized:

```
new oracle.security.pki.OraclePKIProvider();
```

Enabling Oracle's PKI provider dynamically

Providers can be dynamically enabled by calling `System.addProvider` or `System.insertProviderAt` in your Java code.

If you use SSO wallets, you simply need to "add" Oracle's PKI provider because the order does not matter (assuming that there is not other provider for SSO):

```
Security.addProvider(new oracle.security.pki.OraclePKIProvider());
```

If you use PKCS12 wallets, Oracle PKI's provider needs to be inserted at position #3:

```
Security.insertProviderAt(
   new oracle.security.pki.OraclePKIProvider(),3);
```

TLS_RSA_WITH_AES_256_CBC_SHA requires installation of the JCE Unlimited Strength Jurisdiction Policy Files. See J2SE Download Page.

2.2.5. What cipher suites can be used?

During the SSL handshake, both tiers agree on which cipher suite to use. Although JSSE defines a lot of cipher suites only a subset can be used which corresponds to those supported by the Oracle Database server. Note that all ciphers supported by the Database are defined in JSSE and are supported by the Sun's SSL provider. This subset is (cipher suite names in the Java world):

- `SSL_RSA_WITH_3DES_EDE_CBC_SHA`
- `SSL_RSA_WITH_RC4_128_SHA`
- `SSL_RSA_WITH_RC4_128_MD5 (default)`
- `SSL_RSA_WITH_DES_CBC_SHA`
- `SSL_DH_anon_WITH_3DES_EDE_CBC_SHA`
- `SSL_DH_anon_WITH_RC4_128_MD5`
- `SSL_DH_anon_WITH_DES_CBC_SHA`
- `SSL_RSA_EXPORT_WITH_RC4_40_MD5`
- `SSL_RSA_EXPORT_WITH_DES40_CBC_SHA`
- **`TLS_`**`RSA_WITH_AES_128_CBC_SHA`
- **`TLS_`**`RSA_WITH_AES_256_CBC_SHA`

Transport Layer Security (TLS) is basically an incremental improvement to SSL version 3.0.

These names from the Java world match the cipher suite names of the Oracle Database except for the last two where SSL_xxx (Oracle) is changed to TLS_xxx (Java).

These names can be used in the `oracle.net.ssl_cipher_suites` property to force the SSL handshake to use only the defined suites.

If both the server and the client define a list of preferred cipher suites but their intersection is empty, then the connection will fail.

If a list of preferred cipher suites is specified neither on the server nor on the client, the result of the cipher suite negotiation will be `SSL_RSA_WITH_RC4_128_MD5`.

In the next sections, we will go through the necessary steps to configure the JDBC Thin driver for SSL. We will start with the simplest case where we authenticate neither the server nor the client, using SSL for encryption only. We will then use SSL to authenticate the server, and then both the server and the client and finally we will use SSL as an authentication service in the Database.

Because double encryption is prohibited, if you configure SSL encryption, you must disable non-SSL encryption.

CASE #1: USE SSL FOR ENCRYPTION ONLY

Consider the most basic case where you want to use SSL for encryption and data integrity only. You must use Diffie-Hellman anonymous authentication in this case otherwise the connection will fail.

The following cipher suites are available for this case:

```
(SSL_DH_anon_WITH_3DES_EDE_CBC_SHA,
 SSL_DH_anon_WITH_RC4_128_MD5,
 SSL_DH_anon_WITH_DES_CBC_SHA)
```

With Diffie-Hellman anonymous authentication neither the server nor the client will be authenticated through SSL.

This doesn't mean that there is no authentication in the Oracle database, authentication will have to be done through another way (for example a username and password such as scott/tiger).

Server configuration

The listener needs to be configured to turn off the client authentication. In a typical configuration, the listener.ora file would contain:

```
LISTENER = (ADDRESS_LIST=
 (ADDRESS=(PROTOCOL=tcps)(HOST=servername)(PORT=2484))
)
WALLET_LOCATION=(SOURCE=(METHOD=FILE)(METHOD_DATA=(DIRECTORY=/server/wa
llet/path)))
SSL_CLIENT_AUTHENTICATION=FALSE
```

Similarly, SSL client authentication must me turned off in sqlnet.ora:

```
WALLET_LOCATION=(SOURCE=(METHOD=FILE)(METHOD_DATA=(DIRECTORY=/server/wa
llet/path)))
SSL_CLIENT_AUTHENTICATION=FALSE
```

JDBC Thin client configuration

You do not need to set any "truststore" nor "keystore". However the cipher suite must be forced to use Diffie-Hellman anonymous authentication:

```
String url =
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=servernam
e)(PORT=2484))(CONNECT_DATA=(SERVICE_NAME=servicename)))";
Properties props = new Properties();
props.setProperty("user", "scott");
props.setProperty("password", "tiger");
props.setProperty("oracle.net.ssl_cipher_suites",
"(SSL_DH_anon_WITH_3DES_EDE_CBC_SHA, SSL_DH_anon_WITH_RC4_128_MD5,
SSL_DH_anon_WITH_DES_CBC_SHA)");
Connection conn=DriverManager.getConnection(url,props);
```

Obviously, if you set the property `oracle.net.ssl_server_dn_match` to "`true`" (the default is "`false`"), then the connection will fail and you will get the following exception message "`peer not authenticated`" because the JDBC Thin driver couldn't authenticate the server and thus verify that the server's DN matches the one from the URL.

SSL can be combined with other authentication methods such as Kerberos or Radius with the JDBC Thin driver starting in 11.1.0.6.

CASE #2: USE SSL FOR ENCRYPTION AND SERVER AUTHENTICATION

Any cipher suite other than those that use Diffie-Hellman anonymous authentication can be used.

Server configuration

The server configuration remains unchanged.

JDBC Thin client configuration

The "truststore" is used to validate the server's certificate. If none of the trusted certificate contained in the "truststore" can be used to validate the server's certificate, then the connection will fail with the following exception message "unable to find valid certification path to requested target" (with the default SSL provider from Sun).

A Public Key Infrastructure (PKI) is a substrate of network components that provides a security underpinning, based on trust assertions, for an entire organization.

You can use any format of "truststore" as long as you specify a provider for that format. Sun's default PKI provider supports the JKS format for the "truststore". Wallets can also be used with Oracle's PKI provider.

If the truststore format type is JKS

The following code snippet shows how to set a JKS truststore. Note that the path is specified in the MS Windows style as an example:

```
String url =
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=servernam
e)(PORT=2484))(CONNECT_DATA=(SERVICE_NAME=servicename)))");
Properties props = new Properties();
props.setProperty("user", "scott");
props.setProperty("password", "tiger");
props.setProperty("javax.net.ssl.trustStore",
                  "D:\\truststore\\truststore.jks");
props.setProperty("javax.net.ssl.trustStoreType","JKS");
props.setProperty("javax.net.ssl.trustStorePassword","welcome123");
Connection conn = DriverManager.getConnection(url, props);
```

If the truststore is a wallet

You can use wallets to store the trusted certificates. Remember that you need to enable Oracle's PKI provider to use wallets.

The following Java code snippet shows how to use a PKCS12 wallet as a truststore (path specified in the UNIX style):

```
String url =
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=servernam
e)(PORT=2484))(CONNECT_DATA=(SERVICE_NAME=servicename)))");
Properties props = new Properties();
props.setProperty("user", "scott");
props.setProperty("password", "tiger");
props.setProperty("javax.net.ssl.trustStore",
                  "/truststore/ewallet.p12");
props.setProperty("javax.net.ssl.trustStoreType","PKCS12");
props.setProperty("javax.net.ssl.trustStorePassword","welcome123");
Connection conn = DriverManager.getConnection(url, props);
```

If you use Oracle SSO wallets, i.e. if you turned on "auto login" when you created the "truststore" wallet, there is no need for a password:

```
props.setProperty("javax.net.ssl.trustStore",
                  "/truststore/cwallet.sso");
props.setProperty("javax.net.ssl.trustStoreType","SSO");
```

Check the Server's Distinguished Name

If the server is successfully authenticated (meaning its certificate is trusted), its DN can be checked.

The expected DN is specified in the JDBC URL like in this example:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=servername
)(PORT=2484))(CONNECT_DATA=(SERVICE_NAME=servicename))(SECURITY=(SSL_SE
RVER_CERT_DN=\"CN=server_test,C=US\")))
```

The following property also needs to be used to force the JDBC Thin driver to verify the server's DN:

```
props.setProperty("oracle.net.ssl_server_dn_match", "true");
```

As expected, if the DN in the server's certificate does not match the DN specified in the URL, the connection will fail with the following exception message: "Mismatch with the server cert DN".

CASE #3: USE SSL FOR ENCRYPTION AND AUTHENTICATION OF BOTH TIERS

Server configuration

The listener needs to be configured to turn on client authentication. In a typical configuration, the listener.ora file would contain:

```
LISTENER = (ADDRESS_LIST=
 (ADDRESS=(PROTOCOL=tcps)(HOST=servername)(PORT=2484))
)
WALLET_LOCATION=(SOURCE=(METHOD=FILE)(METHOD_DATA=(DIRECTORY=/server/wa
llet/path)))
SSL_CLIENT_AUTHENTICATION=TRUE
```

Similarly, SSL client authentication must be turned on in sqlnet.ora:

```
WALLET_LOCATION=(SOURCE=(METHOD=FILE)(METHOD_DATA=(DIRECTORY=/server/wa
llet/path)))
SSL_CLIENT_AUTHENTICATION=TRUE
```

The location of the wallet in listener.ora and sqlnet.ora must be the same.

JDBC Thin client configuration

The "truststore" must be specified as indicated in the previous section.

Because the client now needs to be authenticated on the server, you must also specify a "keystore". The "keystore" contains not only the client certificate which will be used for authentication but also a set of private/public keys that will be used for encryption.

You can use any format for the "keystore" as long as you specify a provider for that format. Sun's default PKI provider supports JKS and PKCS12 (see JSSE documentation for more details).

If you use a JKS keystore, Sun's PKI provider will be used. If you use PKCS12 or SSO wallets then Oracle's PKI provider must be used.

If you do not provide a "keystore", then the server cannot verify the client certificate and the SSL handshake fails. The connection fails with the following exception message: "Received fatal alert: bad_certificate".

If the keystore format type is JKS

In the following code snippet, "props" are the connection properties and the keystore location is specified in the MS Windows style:

```
props.setProperty("javax.net.ssl.keyStore",
                  "D:\\client_jks\\keystore.jks");
props.setProperty("javax.net.ssl.keyStoreType","JKS");
props.setProperty("javax.net.ssl.keyStorePassword","welcome123");
```

If the keystore is a wallet

Again, Oracle's PKI provider needs to be enabled.

If you use PKCS12 wallets (path is specified in the UNIX style):

```
props.setProperty("javax.net.ssl.keyStore",
                  "/client_wallet/ewallet.p12");
props.setProperty("javax.net.ssl.keyStoreType","PKCS12");
props.setProperty("javax.net.ssl.keyStorePassword","welcome123");
```

And if you use SSO wallets (no password required):

```
props.setProperty("javax.net.ssl.keyStore",
                  "/truststore/cwallet.sso");
props.setProperty("javax.net.ssl.keyStoreType","SSO");
```

Database user authentication through SSL is supported in the JDBC Thin driver starting in 11.1.0.6.

CASE #4: USE SSL AS AN AUTHENTICATION SERVICE IN THE DATABASE

A database user identified by his DN can be authenticated through SSL. This requires that SSL client authentication is enabled. The server verifies the client credentials during the SSL handshake and if the SSL authentication service is enabled then the Database user is authenticated with the Database through his SSL credential.

Note that in the previous sections, we have used SSL to authenticate network tiers such as the database or the mid tier. Comparatively in this section, SSL will be used to authenticate a Database user: each Database user will have to possess his own certificate.

Server configuration

The listener configuration is the same as in the previous section.

In addition to the settings from the previous section, you need to enable the SSL authentication service in the file sqlnet.ora:

```
SQLNET.AUTHENTICATION_SERVICES = (tcps, beq, none)
```

A user that is identified externally as his DN has to be created. For example:

```
SQL> create user sslclient identified externally as
'CN=client_test,C=US';
User created.
SQL> grant connect,create session to sslclient;
Grant succeeded.
```

JDBC Thin client configuration

On the JDBC side, the connection property "`oracle.net.authentication_services`" needs to be used to activate SSL authentication. Note that if you use SSL authentication, you no longer need to provide a username and password but if you do, then the specified username will be used during the Database authentication.

In the following code snipped, SSO wallets are used but you could also use JKS or PKCS12:

```
String url =
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=servernam
e)(PORT=2484))(CONNECT_DATA=(SERVICE_NAME=servicename)))");

Properties props = new Properties();

props.setProperty("oracle.net.authentication_services","(TCPS)");

props.setProperty("javax.net.ssl.trustStore",
                  "D:\\truststore\\cwallet.sso");
```

```
props.setProperty("javax.net.ssl.trustStoreType","SSO");

props.setProperty("javax.net.ssl.keyStore",
                  "D:\\client_wallet\\cwallet.sso");
props.setProperty("javax.net.ssl.keyStoreType","SSO");

Connection conn = DriverManager.getConnection(url, props);
```

WHAT'S NEW IN 11.1.0.7?

Some enhancements related to SSL have been done in the 11.1.0.7 JDBC Thin driver. They include:

Meaningful error messages

The exceptions contain a cause. This makes it a lot easier to debug the configuration issues. You get more meaningful error messages such as "Unable to initialize the key store.java.io.FileNotFoundException: D:\truststore (The system cannot find the path specified)" as the cause of the generic "java.sql.SQLException: Io exception: The Network Adapter could not establish the connection" exception, etc.

Since you can use the 11.1.0.7 JDBC Thin driver against a 11.1.0.6 database, you should always consider giving it a try if you are having trouble configuring SSL.

Information about the cipher suite used

The "`OracleConnection.getEncryptionAlgorithmName()`" method returns the SSL cipher suite used (the result of the SSL handshake). For example it can return the character string:

```
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
```

Better handling of the PKCS12 provider for wallets

To enable Oracle's PKI provider statically for PKCS12 wallets, you had to change the "java.security" file **and** create an instance of OraclePKIProvider in your java code before creating a new connection. In 11.1.0.7, this is no longer the case. Adding the Oracle's PKI provider at position #3 in the "java.security" file is all you need to do.

Possibility to use the "oracle.net.wallet_location" property

In 10.2.0.3 and 11.1.0.6 it is recommended to not use the "oracle.net.wallet_location" connection property because of some bugs related to this property. These bugs have been fixed in 11.1.0.7.

This property can now be used to specify the wallet location using two formats:

1. "`(SOURCE=(METHOD=FILE)(METHOD_DATA=(DIRECTORY=...)))`" where you specify the directory where the client's wallet is located (for example `D:\\path\\to\\directory` under MS Windows). You can also set the wallet password through the "oracle.net.wallet_password" property (new

in 11.1.0.7). If the password is specified, the driver looks for the file "ewallet.p12", otherwise it uses "cwallet.sso".

2. "file:…" where you specify either a directory or a filename. For example under MS Windows: "`file:D:\\path\\to\\directory`" or "`file:D:\\path\\to\\directory\\ewallet.p12`". If the wallet password is specified through the "oracle.net.wallet_password" property then the wallet type is assumed to be PKCS12 otherwise SSO is assumed.

When "oracle.net.wallet_location" is used, the wallet is used for both the truststore and the keystore and it overrides any setting done through the JSSE properties "javax.net.ssl…".

Note that the same wallet can be used for SSL and to store a username and password pair using the "mkstore" utility.

CONCLUSION

SSL offers a high level of network security including encryption, data integrity and authentication of both tiers using the latest standards in cryptography. As always with security, there is a cost: the initial SSL handshake and very little additional CPU afterwards. But when you think of all the security features that come all at once, it's worth the cost.

All the SSL features are available when using the JDBC Thin driver and the configuration is fairly easy.

So if you are looking for a flexible, robust and easy way to secure your Java applications on the network with the JDBC Thin driver, using SSL is a good option.

APPENDIX A TROUBLESHOOTING

This section contains a non-exhaustive list of exceptions that you may encounter when configuring the JDBC Thin driver for SSL.

- "java.lang.NoClassDefFoundError: com/phaos/crypto/AuthenticationException": you are using 10.2.0.4 and you need to include ojpse.jar in your classpath.

- "java.lang.NoClassDefFoundError: oracle/security/crypto/core/AuthenticationException": you are using 11.1.0.x and you need to include osdt_core.jar in your classpath.

- "java.lang.NoClassDefFoundError: oracle/security/crypto/cert/PKCS12": you are using 11.1.0.x and you need to include osdt_cert.jar in your classpath.

- "java.sql.SQLException: Io exception: The Network Adapter could not establish the connection": this exception is very generic and can be caused by multiple configuration issues. In 10.2.0.3 and 11.1.0.6, you do not get the cause which could be:
    - "Unable to initialize the key store.java.io.FileNotFoundException: D:\truststore (The system cannot find the path specified)" in other words the path specified in the truststore (the same could happen for the keystore) property is wrong.
    - "java.security.KeyStoreException: SSO not found" because you specified an SSO truststore type but you didn't enable the Oracle's PKI provider (`Security.addProvider(new oracle.security.pki.OraclePKIProvider());`).
    - "java.io.IOException: Invalid keystore format" if for example you set `javax.net.ssl.trustStore = D:\\cwallet.sso` without specifying the truststoretype: `javax.net.ssl.trustStoreType = SSO`.
    - "java.io.IOException: failed to decrypt safe contents entry: java.lang.ArithmeticException: / by zero because this is needed": if you are using a PKCS12 wallet, you need to use the Oracle PKI provider which needs to bet set at position #3 or higher (`Security.insertProviderAt(new oracle.security.pki.OraclePKIProvider(),3);`).
    - "com.phaos.ASN1.ASN1FormatException: com.phaos.crypto.CipherException: Invalid padding string (or incorrect password)" because the password is missing (when using PKCS12 wallets).
- "java.sql.SQLException: Io exception: The Network Adapter could not establish the connection": you may get this exception when attempting to connect with SSL through a **firewall** because the listener may send a REDIRECT packet to the client with a different TCP port; when the client attempts to reconnect using this new port the firewall blocks it.
    - The following note describes a workaround: "Note 125021.1: Using SSL will cause Port redirection. The workaround is to select and set the ports using MTS in the INIT<SID>.ORA".

    - Starting in 11.2.0.2.0, the JDBC thin driver has the ability to renegociate the SSL handshake and hence no longer needs to reconnect through a different port. There is a direct handoff by the listener to the server process. This addition to JDBC thin driver was done in the fix for bug 8935561 which was also backported into 11.1.0.7.0 (backport label ST_JAVAVM_BLR_8935561_BACKPORT_11.1.0.7.0).

- "java.sql.SQLException: Io exception: Remote host closed connection during handshake". Look in the listener.log file. You should see: either "TNS-12560: TNS:protocol adapter error" or "TNS-00540: SSL protocol adapter failure". You need to stop the listener, add the wallet location in listener.ora and restart it.
- "java.sql.SQLException: Io exception: sun.security.validator.ValidatorException: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target": the client truststore does not contain the path to approve the certificate of the server (contained in the server's wallet). You should also make sure that both sqlnet.ora and listener.ora point to the same wallet location. If you have changed the wallet location in the "listener.ora", you also need to restart the listener.
- "java.sql.SQLException: Io exception: Received fatal alert: bad_certificate", turn off SSL client authentication (ssl_client_authentication=false) in listener.ora file and sqlnet.ora or provide a valid keystore.
- "java.sql.SQLException: Io exception: sun.security.validator.ValidatorException: No trusted certificate found": you need to provide the truststore or use a cipher suite that uses anonymous authentication.
- "java.sql.SQLException: Io exception: Received fatal alert: handshake_failure": if the client and server for example cannot agree on which cipher suite to use.
- "java.sql.SQLException: Io exception: java.lang.RuntimeException: Unexpected error: java.security.InvalidAlgorithmParameterException: the trustAnchors parameter must be non-empty": if you are using PKCS12 wallets and Oracle's PKI provider isn't properly enabled. This exception comes from the PKCS12 implementation from Sun (Sun's PKI provider) which isn't compatible with Oracle wallets. Oracle's PKI provider must be properly enabled for PKCS12 either statically or dynamically if you use PKCS12 wallets.
- "java.sql.SQLException: Io exception: Broken pipe": this exception occurs when client authentication is enabled on the server and the

certificate sent by the client couldn't be checked by the server so the SSL handshake fails. Usually this means that your truststore is correct but your keystore isn't.

APPENDIX B CREATING TRUSTSTORES AND KEYSTORES

Using orapki

Create a wallet for the test CA

For test purposes, we are going to use a CA called "root" which has a self-signed certificate.

Create an empty wallet in the root directory:

```
> orapki wallet create -wallet ./root
```

You end up with ewallet.p12 in the ./root directory

Add a self-signed certificate to the wallet:

```
> orapki wallet add -wallet ./root -dn CN=root_test,C=US -keysize 2048
-self_signed -validity 3650
```

View the wallet:

```
> orapki wallet display -wallet ./root

Requested Certificates:
User Certificates:
Subject:        CN=root_test,C=US
Trusted Certificates:
Subject:        CN=root_test,C=US
(…)
```

Export the certificate:

```
> orapki wallet export -wallet ./root -dn CN=root_test,C=US -cert
./root/b64certificate.txt
```

Create a wallet for the Oracle server

Create an empty wallet with auto login enabled:

```
> orapki wallet create -wallet ./server -auto_login
```

Two files are created under the server directory:

```
server/cwallet.sso  server/ewallet.p12
```

Add a user in the wallet (a new pair of private/public keys is created):

```
> orapki wallet add -wallet ./server -dn CN=server_test,C=US -keysize
2048
```

If you display the server's wallet you will see the following requested certificate:

```
Requested Certificates:
Subject:        CN=server_test,C=US
```

Export the certificate request to a file:

```
> orapki wallet export -wallet ./server -dn CN=server_test,C=US -
request ./server/creq.txt
```

Using the test CA, sign the certificate request:

```
> orapki cert create -wallet ./root -request ./server/creq.txt -cert
./server/cert.txt -validity 3650
```

You now have the following files under the server directory:

```
server/cert.txt  server/creq.txt  server/cwallet.sso
server/ewallet.p12
```

View the signed certificate:

```
> orapki cert display -cert ./server/cert.txt -complete
```

```
{ fingerprint = cb384d05b627d2cb20f0499781f704f6, notBefore = Tue Nov
13 17:44:47 PST 2007, notAfter = Fri Nov 10 17:44:47 PST 2017, holder =
CN=server_test,C=US, issuer = CN=root_test,C=US, serialNo = 0,
sigAlgOID = 1.2.840.113549.1.1.4, key = { modulus =
19593679513746015765355962711079952774176644245360983953992652691247218
37799436134516119827593421444747722682023020838584911001892449638770444
84639443652466378093963161320192391160905740289465375255115252978607983
40990134659538369793777897678910491880573044079214697664783396711473736
71373082779621690875555437771056651083920634171604505885359922675484607
49873033793093373387298332477942247788814090235867746623126621826931950
55288771727761868895535312229718865977983610913559597159181862643061313
98447800360776201784250574411699704826790543407179460023192497496919803
22403368755903660354311823839357137717512645813 03, exponent = 65537 } }
```

Add the test CA's trusted certificate to the wallet

```
> orapki wallet add -wallet ./server -trusted_cert -cert
./root/b64certificate.txt
```

If you display the server's certificate at this point, you should see a new entry in the list of trusted certificates:

```
Subject:        CN=root_test,C=US
```

Finally add the user certificate to the wallet:

```
> orapki wallet add -wallet ./server -user_cert -cert ./server/cert.txt
```

Displaying the server's certificate will show:

```
Requested Certificates:
User Certificates:
Subject:        CN=server_test,C=US
Trusted Certificates:
Subject:        CN=root_test,C=US
(...)
```

Note that if you had not added the trusted certificate in the previous step you would have run into this error:

```
Could not install user cert at./client/cert.txt
Please add all trusted certificates before adding the user certificate
```

For the client (proceed the same way as for the server)

```
> orapki wallet create -wallet ./client_wallet -auto_login

> orapki wallet add -wallet ./client_wallet -dn CN=client_test,C=US -
keysize 2048

> orapki wallet export -wallet ./client_wallet -dn CN=client_test,C=US
-request ./client_wallet/creq.txt

> orapki cert create -wallet ./root -request ./client_wallet/creq.txt
-cert ./client_wallet/cert.txt -validity 3650

> orapki wallet add -wallet ./client_wallet -trusted_cert -cert
./root/b64certificate.txt

> orapki wallet add -wallet ./client_wallet -user_cert -cert
./client_wallet/cert.txt
```

To create a wallet that contains only the trusted certificate

For a wallet that would be used for the truststore only:

```
> orapki wallet create -wallet ./truststore -auto_login
> orapki wallet add -wallet ./truststore -trusted_cert -cert
./root/b64certificate.txt
> orapki wallet display -wallet ./truststore
Requested Certificates:
User Certificates:
Trusted Certificates:
(...)
Subject:        CN=root_test,C=US
```

Using keytool

Create a JKS keystore

Create a new private/public key pair for 'CN=client_test, C=US':

```
> keytool -genkey -alias testclient -dname 'CN=client_test, C=US' -
storepass 'welcome123' -storetype JKS -keystore ./client_jks/client.jks
-keyalg RSA
```

Generate a CSR (Certificate Signing Request):

```
> keytool -certreq -alias testclient -file ./client_jks/csr.txt -
keystore ./client_jks/client.jks -storepass 'welcome123'
```

Sign the client certificate using the test CA (root):

```
> orapki cert create -wallet ./root -request ./client_jks/csr.txt  -
cert ./client_jks/cert.txt -validity 3650
```

Import the signed certificate:

```
> keytool -import -v -alias testclient -file ./client_jks/cert.txt -
keystore ./client_jks/client.jks -storepass 'welcome123'
keytool error: java.lang.Exception: Failed to establish chain from
reply
```

Oops you need to import the test CA's certificate:

```
> keytool -import -v -alias testroot -file ./root/b64certificate.txt -
keystore ./client_jks/client.jks -storepass 'welcome123'
```

And retry:

```
> keytool -import -v -alias testclient -file ./client_jks/cert.txt -
keystore ./client_jks/client.jks -storepass 'welcome123'
```

At any time you can display the keystore by calling:

```
> keytool -list -keystore ./client_jks/client.jks -storepass
'welcome123' -v

Keystore type: jks
Keystore provider: SUN

Your keystore contains 2 entries

Alias name: testroot
Creation date: Nov 13, 2007
Entry type: trustedCertEntry

Owner: CN=root_test, C=US
Issuer: CN=root_test, C=US
Serial number: 0
Valid from: Tue Nov 13 17:33:19 PST 2007 until: Fri Nov 10 17:33:19 PST
2017
Certificate fingerprints:
         MD5:  71:66:1B:34:F3:40:75:5C:A1:B1:5E:D5:98:E6:60:ED
         SHA1:
13:C6:35:F8:EF:48:0F:75:04:99:02:F2:B4:A4:DA:CE:BE:E0:65:9F


*******************************************
*******************************************


Alias name: testclient
Creation date: Nov 13, 2007
Entry type: keyEntry
Certificate chain length: 2
Certificate[1]:
Owner: CN=client_test, C=US
Issuer: CN=root_test, C=US
Serial number: 0
Valid from: Tue Nov 13 18:44:49 PST 2007 until: Fri Nov 10 18:44:49 PST
2017
Certificate fingerprints:
         MD5:  67:5A:17:E8:08:B6:49:3D:71:9F:C6:83:C8:5F:4D:3A
```

```
        SHA1:
D3:DF:4E:6C:6E:4E:11:3B:83:7C:12:B8:1E:8B:D7:F1:47:AE:DF:80
Certificate[2]:
Owner: CN=root_test, C=US
Issuer: CN=root_test, C=US
Serial number: 0
Valid from: Tue Nov 13 17:33:19 PST 2007 until: Fri Nov 10 17:33:19 PST
2017
Certificate fingerprints:
         MD5:  71:66:1B:34:F3:40:75:5C:A1:B1:5E:D5:98:E6:60:ED
         SHA1:
13:C6:35:F8:EF:48:0F:75:04:99:02:F2:B4:A4:DA:CE:BE:E0:65:9F


*******************************************
*******************************************
```

Create a JKS truststore

We import the test CA certificate:

```
> keytool -import -v -alias testroot -file ./root/b64certificate.txt -
keystore ./truststore/truststore.jks -storetype JKS -storepass
welcome123
```

Display the truststore to make sure that the test CA certificate is present:

```
> keytool -list -keystore ./truststore/truststore.jks -storepass
'welcome123' -v

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: testroot
Creation date: Nov 13, 2007
Entry type: trustedCertEntry

Owner: CN=root_test, C=US
Issuer: CN=root_test, C=US
Serial number: 0
Valid from: Tue Nov 13 17:33:19 PST 2007 until: Fri Nov 10 17:33:19 PST
2017
Certificate fingerprints:
         MD5:  71:66:1B:34:F3:40:75:5C:A1:B1:5E:D5:98:E6:60:ED
         SHA1:
13:C6:35:F8:EF:48:0F:75:04:99:02:F2:B4:A4:DA:CE:BE:E0:65:9F


*******************************************
*******************************************
```

SSL With Oracle JDBC Thin Driver
Initial version: November 2007
Update: November 2008, April 2010
Author: Jean de Lavarene
Contributors: Benjamin Job, Kuassi Mensah

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

Copyright © 2008, Oracle. All rights reserved.
This document is provided for information purposes only and the
contents hereof are subject to change without notice.
This document is not warranted to be error-free, nor subject to any
other warranties or conditions, whether expressed orally or implied
in law, including implied warranties and conditions of merchantability
or fitness for a particular purpose. We specifically disclaim any
liability with respect to this document and no contractual obligations
are formed either directly or indirectly by this document. This document
may not be reproduced or transmitted in any form or by any means,
electronic or mechanical, for any purpose, without our prior written permission.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.