9

JDBC Client-Side Security Features

This chapter discusses support for IAM authentication for Autonomous Database, login authentication, network encryption and integrity with respect to features of the Oracle Advanced Security options in the JDBC OCI and the JDBC Thin drivers.

Note:

- This discussion is not relevant to the server-side internal driver because all communication through server-side internal driver is completely internal to the server.
- Using `SHA-1` (Secure Hash Algorithm 1) with the parameters `SQLNET.CRYPTO_CHECKSUM_TYPES_CLIENT` and `SQLNET.CRYPTO_CHECKSUM_TYPES_SERVER` is deprecated in this release, and can be desupported in a future release. Using `SHA-1` ciphers with `DBMS_CRYPTO` is also deprecated (`HASH_SH1`, `HMAC_SH1`). Instead of using `SHA1`, Oracle recommends that you start using a stronger `SHA-2` cipher in place of the `SHA-1` cipher.

Oracle Advanced Security, previously known as the Advanced Networking Option (ANO) or Advanced Security Option (ASO), provides industry standards-based network encryption, network integrity, third-party authentication, single sign-on, and access authorization. Both the JDBC OCI and JDBC Thin drivers support all the Oracle Advanced Security features.

Note:

If you want to use the security policy file for JDBC `ojdbc.policy`, then you can download the file from the following link:

http://www.oracle.com/technetwork/index.html

The `ojdbc.policy` file contains the granted permissions that you need to run your application in control environment of the Java Security Manager. You can either use this file itself as your Java policy file, or get contents from this file and add the content in your Java policy file. This file contains permissions like:

- A few mandatory permissions that are always required, for example, permission `java.util.PropertyPermission "user.name", "read";`
- A few driver-specific permissions, for example, JDBC OCI driver needs permission `java.lang.RuntimePermission "loadLibrary.ocijdbc12";`
- A few feature-based permissions, for example, permissions related to XA, XDB, FCF and so on

You can set the system properties mentioned in the file or direct values for permissions as per your requirement.

This chapter contains the following sections:

9.1 Support for Token-Based Authentication for IAM

In Oracle Database release 23ai, the JDBC Thin drivers provide enhanced support for Oracle Cloud Infrastructure (OCI) Identity Access Management (IAM).

While connecting to the database, the JDBC application provides a token to the database. The database verifies the token with a public key that it requests from the authentication service, and retrieves the corresponding user group membership information to find the database schema and role mappings to complete the user authorization to the database.

Additionally, the application sends a signed header, which proves that it possesses a private key that is paired to a public key embedded in the token. If both the token and the signature are valid, and there exists a mapping between the IAM user and a database user, then access to the database is granted to the JDBC application.

The token-based authentication is supported in the following ways:

9.1.1 Using the File System

When a database token is available on the file system, for example, if you use the Oracle Cloud Infrastructure Command-Line Interface (OCI CLI), then you can configure the JDBC driver to use this token for connecting to the Database.

You can use the `CONNECTION_PROPERTY_TOKEN_AUTHENTICATION` for this purpose, which can be specified in the following ways:

- As an `ojdbc.properties` file
- As a JVM system property
- As a parameter in the query section of a connection string
- With a `Properties` object passed to `OracleDataSource.setConnectionProperties(Properties)`
- As a parameter in the `SECURITY` section of an Oracle Net Descriptor

JDBC supports the following two types of connection strings, where you can specify this parameter:

- As an Oracle Net descriptor, where you can specify the parameter as:

  ```
  jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=dbhost)(PORT=1522)
  (PROTOCOL=tcps))(SECURITY=(SSL_SERVER_DN_MATCH=ON)
  (TOKEN_AUTH=OCI_TOKEN))(CONNECT_DATA=(SERVICE_NAME=my.example.com)))
  ```

- As a connection property, where you can specify the parameter as:

  ```
  jdbc:oracle:thin:@tcps:dbhost:1522/my.example.com?
  oracle.jdbc.tokenAuthentication=OCI_TOKEN&oracle.jdbc.tokenLocation
  =”/path/to/my/token”
  ```

When `CONNECTION_PROPERTY_TOKEN_AUTHENTICATION` is set to `OCI_TOKEN`, then the `CONNECTION_PROPERTY_TOKEN_LOCATION` specifies the file system path, from where the driver obtains the access tokens. The default location is `$HOME/.oci/db-token/`. You can set this property to a different value to specify a nondefault location. The path specified by this property must be a directory containing files named `token` and `oci_db_key.pem`.

Note:

- If an Oracle Net Descriptor style URL includes the `TOKEN_LOCATION` parameter, then the value of that parameter takes precedence over a value defined by `CONNECTION_PROPERTY_TOKEN_LOCATION`.
- The token file must contain a JSON Web Token (JWT) on a single line of UTF-8 encoded text. The JWT format is specified by RFC 7519.
- The token location must also contain a private key file named `oci_db_key.pem`. The private key file must use the PEM format and contain the base64 encoding of an RSA private key in the PKCS#8 encoding

You can also set the connection property `CONNECTION_PROPERTY_TOKEN_AUTHENTICATION` (`oracle.jdbc.tokenAuthentication`) to `OAUTH` and the connection property `CONNECTION_PROPERTY_TOKEN_LOCATION` (`oracle.jdbc.tokenLocation`) to point to the bearer token on the file system. There is no default location set in this case, so you must set the location to either of the following:

- A directory, in which case, the driver loads a file named `token`
- A fully qualified file name

You can perform this in the following ways:

- **Configuring the `ojdbc.properties` file**

  ```
  # Enable the OAUTH authentication mode
  oracle.jdbc.tokenAuthentication=OAUTH
  # Specify the location of the Bearer token location
  oracle.jdbc.tokenLocation=/home/user1/mytokens/jwtbearertoken
  ```

- **Using the JDBC URL**

  ```
  jdbc:oracle:thin:@tcps:adb.mydomain.oraclecloud.com:1522/
  xyz.adb.oraclecloud.com?
  ```

```
oracle.jdbc.tokenAuthentication=OAUTH&oracle.jdbc.tokenLocation=/home/user/
token
```

- **Using the TNS format for a JDBC URL**:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCPS)(PORT=1521)
(HOST=adb.mydomain.oraclecloud.com))(CONNECT_DATA=
(SERVICE_NAME=xyz.adb.oraclecloud.com))(SECURITY=(TOKEN_AUTH=OAUTH)
(TOKEN_LOCATION=/home/user1/mytokens/jwtbearertoken)))"
```

9.1.2 Using the oracle.jdbc.accessToken Connection Property

Set the `CONNECTION_PROPERTY_ACCESS_TOKEN (oracle.jdbc.accessToken)` to the access token value.

You can perform this in the following ways:

> **Note:**
>
> You must enclose the token value with double quotation marks ("") to escape the equal signs (=) that may appear in the token value.

- **Using the JDBC URL**

```
jdbc:oracle:thin:@tcps:adb.mydomain.oraclecloud.com:1522/
xyz.adb.oraclecloud.com?oracle.jdbc.accessToken="ey...5c"
```

- **Using the Descriptor URL**

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=mydomain.com)
(PORT=5525))
      (CONNECT_DATA=(SERVICE_NAME=myservice.com)))?
        oracle.jdbc.accessToken="ey...5c"
```

- **Configuring the `ojdbc.properties`** File

```
# Enable the OAUTH authentication mode
oracle.jdbc.accessToken="ey...5c"
```

Or,

```
# Enable the OAUTH authentication mode
oracle.jdbc.accessToken=${DATABASE_ACCESS_TOKEN}
```

Where, the access token is the value of the `DATABASE_ACCESS_TOKEN` environment variable.

> **Note:**
>
> You do not need to set `oracle.jdbc.tokenAuthentication=OAUTH` as the driver automatically sets the `OAUTH` mode, when the access token is provided.

9.1.3 Using the OracleConnectionBuilder Interface

Call the `OracleConnectionBuilder.accessToken` method for authentication with a database access token. This method accepts a token value that the application obtains from the authentication service.

If you pass a token using this method, then it overrides the connection string setting of `TOKEN_AUTH=OCI_TOKEN`, which means that JDBC does not read the token from the file system as it typically does. Instead, JDBC uses the `AccessToken` object provided to the `accessToken` method.

In this case, JDBC also generates a signature using the private key and sends it to the Database along with the IAM database access token. First, the Database verifies the token with the public signing key from IAM. Then, it verifies the JDBC-generated signature by decrypting it with a public key that is embedded in the token. If the decrypted signature is valid, then it proves that JDBC possesses the private key.

A single instance of the `OracleDataSource` class, configured with a single URL, creates instances of the `OracleConnectionBuilder` interface. These instances support traditional authentication with O5Logon, while also supporting token-based authentication. The application then calls the methods to configure a user and a password, or calls methods to configure a token. However, it is invalid to configure this builder with both a token and with a user name or a password. If both the `accessToken` method and the `password` or `user` methods are invoked with non-null values, then a `SQLException`, indicating an invalid configuration, is thrown when creating a connection with this builder.

9.1.4 Using the OracleDataSource Class

Call the `OracleCommonDataSource.setTokenSupplier(AccessToken accessToken)` method for authentication with a database access token.

This method sets a supplier function that generates an access token when creating a connection with this `DataSource`. The supplier function is invoked each time this `DataSource` creates a connection. Instances of access tokens, which are generated by the supplier, must represent a token type that is supported by Oracle Database for client authentication. The supplier must be thread safe.

Note:

Use the `AccessToken.createJsonWebTokenCache(Supplier)` method to create a thread safe Supplier that caches tokens from a user defined Supplier.

It is invalid to configure this `DataSource` with both a token supplier and with a user name or password. If you invoke the `setUser(String)`, `setPassword(String)`, `setConnectionProperties(java.util.Properties)`, or `setConnectionProperty(String, String)` methods to configure this `DataSource` with a user name or a password, and also invoke the `setTokenSuppliersetTokenSupplier(AccessToken accessToken)` method to configure a token supplier, then a `SQLException` indicating an invalid configuration is thrown, when creating a connection with this `DataSource`.

The access tokens are ephemeral in nature and expire within an hour or less. So, use the `Supplier` type that enables an instance of the `OracleDataSource` class to obtain a newly generated token, each time it creates a connection. The `Supplier` can generate the same token multiple times, until the expiration time of that token passes. After the expiration time of a token is over, the `Supplier` must no longer generate that token, and instead begin to generate a new token with a later expiration time.

See Also:

- Authenticating and Authorizing IAM Users for Oracle Autonomous Databases
- About TCP/IP with TLS Protocol

9.2 Support for Token-Based Authentication for Azure AD

In this release of Oracle Database, the JDBC Thin drivers provide support for Azure Active Directory (Azure AD) OAuth2 access tokens.

While connecting to the database, the JDBC application provides a token to the database. The database verifies the token with a public key that it requests from the authentication service, and retrieves the corresponding user group membership information to find the database schema and role mappings to complete the user authorization to the database.

The token-based authentication is supported in the following ways:

9.2.1 Using the File System

When a database token is available on the file system, then you can configure the JDBC driver to use this token for connecting to the Database.

You can use the `CONNECTION_PROPERTY_TOKEN_AUTHENTICATION` for this purpose, which can be specified in the following ways:

- As an `ojdbc.properties` file
- As a JVM system property
- As a parameter in the query section of a connection string
- With a `Properties` object passed to `OracleDataSource.setConnectionProperties(Properties)`
- As a parameter in the `SECURITY` section of an Oracle Net Descriptor

JDBC supports the following two types of connection strings, where you can specify this parameter:

- As an Oracle Net descriptor, where you can specify the parameter as:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=dbhost)(PORT=1522)
(PROTOCOL=tcps))(SECURITY=(SSL_SERVER_DN_MATCH=ON)
(TOKEN_AUTH=OAUTH)(TOKEN_LOCATION=/path/to/my/token)))
```

- As a connection property, where you can specify the parameter as:

```
jdbc:oracle:thin:@tcps:dbhost:1522/my.example.com?
oracle.jdbc.tokenAuthentication=OAUTH&oracle.jdbc.tokenLocation
="/path/to/my/token"
```

When `CONNECTION_PROPERTY_TOKEN_AUTHENTICATION` is set to `OAUTH`, then the `CONNECTION_PROPERTY_TOKEN_LOCATION` specifies the file system path, from where you can obtain the access tokens. There is no default location in this case. You must set the token location, which can be a directory containing the token in a filed named `token`. For instance, if the directory `mytokendirectory` contains the file named `token`, then you set the token location in the following way:

```
/path/to/mytokendirectory
```

Note:

- If an Oracle Net Descriptor style URL includes the `TOKEN_LOCATION` parameter, then the value of that parameter takes precedence over a value defined by `CONNECTION_PROPERTY_TOKEN_LOCATION`.
- The token file must contain a JSON Web Token (JWT) on a single line of UTF-8 encoded text. The JWT format is specified by RFC 7519.

You can also set the connection property `CONNECTION_PROPERTY_TOKEN_AUTHENTICATION` (`oracle.jdbc.tokenAuthentication`) to `OAUTH` and the connection property `CONNECTION_PROPERTY_TOKEN_LOCATION` (`oracle.jdbc.tokenLocation`) to point to the bearer token on the file system. There is no default location set in this case, so you must set the location to either of the following:

- A directory, in which case, the driver loads a file named `token`
- A fully qualified file name

You can perform this in the following ways:

- **Configuring the `ojdbc.properties`** file

```
# Enable the OAUTH authentication mode
oracle.jdbc.tokenAuthentication=OAUTH
# Specify the location of the Bearer token location
oracle.jdbc.tokenLocation=/home/user1/mytokens/jwtbearertoken
```

- **Using the JDBC URL**

```
jdbc:oracle:thin:@tcps:adb.mydomain.oraclecloud.com:1522/
xyz.adb.oraclecloud.com?
oracle.jdbc.tokenAuthentication=OAUTH&oracle.jdbc.tokenLocation=/home/user/
token
```

- **Using the TNS format for a JDBC URL**:

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCPS)(PORT=1521)
(HOST=adb.mydomain.oraclecloud.com))(CONNECT_DATA=
(SERVICE_NAME=xyz.adb.oraclecloud.com))(SECURITY=(TOKEN_AUTH=OAUTH)
(TOKEN_LOCATION=/home/user1/mytokens/jwtbearertoken)))"
```

9.2.2 Using the oracle.jdbc.accessToken Connection Property

Set the `CONNECTION_PROPERTY_ACCESS_TOKEN` (`oracle.jdbc.accessToken`) to the access token value.

You can perform this in the following ways:

> **Note:**
>
> You must enclose the token value with double quotation marks ("") to escape the equal signs (=) that may appear in the token value.

- **Using the JDBC URL**

```
jdbc:oracle:thin:@tcps:adb.mydomain.oraclecloud.com:1522/
xyz.adb.oraclecloud.com?oracle.jdbc.accessToken="ey...5c"
```

- **Using the Descriptor URL**

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=mydomain.com)
(PORT=5525))
      (CONNECT_DATA=(SERVICE_NAME=myservice.com)))?
        oracle.jdbc.accessToken="ey...5c"
```

- **Configuring the `ojdbc.properties` File**

```
# Enable the OAUTH authentication mode
oracle.jdbc.accessToken="ey...5c"
```

Or,

```
# Enable the OAUTH authentication mode
oracle.jdbc.accessToken=${DATABASE_ACCESS_TOKEN}
```

Where, the access token is the value of the `DATABASE_ACCESS_TOKEN` environment variable.

> **Note:**
>
> You do not need to set `oracle.jdbc.tokenAuthentication=OAUTH` as the driver automatically sets the `OAUTH` mode, when the access token is provided.

9.2.3 Using the OracleConnectionBuilder Interface

Call the `OracleConnectionBuilder.accessToken` method for authentication with a database access token.

This method accepts a token value that the application obtains from the authentication service. If you pass a token using this method, then it overrides the connection string setting of `TOKEN_AUTH=OCI_TOKEN`then `TOKEN_AUTH=OAUTH`, which means that JDBC does not read the token from the file system as it typically does. Instead, JDBC uses the `AccessToken` object provided to the `accessToken` method.

A single instance of the `OracleDataSource` class, configured with a single URL, creates instances of the `OracleConnectionBuilder` interface. These instances support traditional authentication with O5Logon, while also supporting token-based authentication. The application then calls the methods to configure a user and a password, or calls methods to configure a token. However, it is invalid to configure this builder with both a token and with a user name or a password. If both the `accessToken` method and the `password` or `user` methods are invoked with non-null values, then a `SQLException`, indicating an invalid configuration, is thrown when creating a connection with this builder.

9.2.4 Using the OracleDataSource Class

Call the `OracleCommonDataSource.setTokenSupplier(AccessToken accessToken)` method for authentication with a database access token.

This method sets a supplier function that generates an access token when creating a connection with this `DataSource`. The supplier function is invoked each time this `DataSource` creates a connection. Instances of access tokens, which are generated by the supplier, must represent a token type that is supported by Oracle Database for client authentication. The supplier must be thread safe.

Note:

Use the `AccessToken.createJsonWebTokenCache(Supplier)` method to create a thread safe Supplier that caches tokens from a user defined Supplier.

It is invalid to configure this `DataSource` with both a token supplier and with a user name or password. If you invoke the `setUser(String)`, `setPassword(String)`, `setConnectionProperties(java.util.Properties)`, or `setConnectionProperty(String, String)` methods to configure this `DataSource` with a user name or a password, and also invoke the `setTokenSuppliersetTokenSupplier(AccessToken accessToken)` method to configure a token supplier, then a `SQLException` indicating an invalid configuration is thrown, when creating a connection with this `DataSource`.

The access tokens are ephemeral in nature and expire within an hour or less. So, use the `Supplier` type that enables an instance of the `OracleDataSource` class to obtain a newly generated token, each time it creates a connection. The `Supplier` can generate the same token multiple times, until the expiration time of that token passes. After the expiration time of a token is over, the `Supplier` must no longer generate that token, and instead begin to generate a new token with a later expiration time.

See Also:

- Use Azure Active Directory (Azure AD) with Autonomous Database
- About TCP/IP with TLS Protocol

9.3 Support for Oracle Advanced Security

This section describes the following concepts:

- Overview of Oracle Advanced Security
- JDBC OCI Driver Support for Oracle Advanced Security
- JDBC Thin Driver Support for Oracle Advanced Security

9.3.1 Overview of Oracle Advanced Security

Oracle Advanced Security provides the following security features:

- Network Encryption

  Sensitive information communicated over enterprise networks and the Internet can be protected by using encryption algorithms, which transform information into a form that can be deciphered only with a decryption key. For example, AES.

  To ensure network integrity during transmission, Oracle Advanced Security generates a cryptographically secure message digest. Starting from Oracle Database 12*c* Release 1 (12.1), the SHA-2 list of hashing algorithms are also supported and Oracle Advanced Security uses the following hashing algorithms to generate the secure message digest and includes it with each message sent across a network.

  This protects the communicated data from attacks, such as data modification, deleted packets, and replay attacks.

  The following code snippet shows how to calculate the checksum using any of the algorithms mentioned previously:

  ```
  prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES,
  "( SHA1)");
  prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL,
  "REQUIRED");
  ```

- Strong Authentication

  To ensure network security in distributed environments, it is necessary to authenticate the user and check their credentials. Password authentication is the most common means of authentication. Oracle Database enables strong authentication with Oracle authentication adapters, which support various third-party authentication services, including TLS with digital certificates. Oracle Database supports the following industry-standard authentication methods:

  – Kerberos
  – Remote Authentication Dial-In User Service (RADIUS)
  – Transport Layer Security (TLS)

See Also:

Oracle Database Security Guide

9.3.2 JDBC OCI Driver Support for Oracle Advanced Security

If you are using the JDBC OCI driver, which presumes that you are running from a computer with an Oracle client installation, then support for Oracle Advanced Security and incorporated third-party features is fairly similar to the support provided by in any Oracle client situation. Your use of Advanced Security features is determined by related settings in the `sqlnet.ora` file on the client computer.

Note:

Starting from Oracle Database 12*c* Release 1 (12.1), Oracle recommends you to use the configuration parameters present in the new XML configuration file `oraaccess.xml` instead of the OCI-specific configuration parameters present in the `sqlnet.ora` file. However, the configuration parameters present in the `sqlnet.ora` file are still supported.

The JDBC OCI driver attempts to use external authentication if you try connecting to a database without providing a password. The following are some examples using the JDBC OCI driver to connect to a database without providing a password:

TLS Authentication

The following code snippet shows how to use TLS authentication to connect to the database:

Example 9-1 Using TLS Authentication to Connect to the Database

```
import java.sql.*;
import java.util.Properties;

public class test
{
    public static void main( String [] args ) throws Exception
    {
        String url = "jdbc:oracle:oci:@"
         +"(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=localhost)(PORT=5221))"
         +"(CONNECT_DATA=(SERVICE_NAME=orcl)))";
        Driver driver = new oracle.jdbc.OracleDriver();
        Properties props = new Properties();
        Connection conn = driver.connect( url, props );
        conn.close();
    }
}
```

Using a Data Source

The following code snippet shows how to use a data source to connect to the database:

Example 9-2 Using a Data Source to Connect to the Database

```
import java.sql.*;
import javax.sql.*;
import java.util.Properties;
import oracle.jdbc.pool.*;

public class testpool {
    public static void main( String args ) throws Exception
    { String url = "jdbc:oracle:oci:@" +"(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)
(HOST=localhost)(PORT=5221))"
 +"(CONNECT_DATA=(SERVICE_NAME=orcl)))";
    OracleConnectionPoolDataSource ocpds = new OracleConnectionPoolDataSource();
    ocpds.setURL(url);
    PooledConnection pc = ocpds.getPooledConnection();
    Connection conn = pc.getConnection();
    }
    }
```

Note:

The key exception to the preceding, with respect to Java, is that the Transport Layer Security (TLS) protocol is supported by the Oracle JDBC OCI drivers only if you use native threads in your application. This requires special attention, because green threads are generally the default.

9.3.3 JDBC Thin Driver Support for Oracle Advanced Security

The JDBC Thin driver cannot assume the existence of an Oracle client installation or the presence of the `sqlnet.ora` file.

Therefore, it uses a Java approach to support Oracle Advanced Security. Java classes that implement Oracle Advanced Security are included in the `ojdbc8.jar`, `ojdbc11.jar`, and `ojdbc17.jar` files. Security parameters for encryption and integrity, usually set in the `sqlnet.ora` file, are set using a Java `Properties` object or through system properties.

9.4 Support for Login Authentication

Basic login authentication through JDBC consists of user names and passwords, as with any other means of logging in to an Oracle server. Specify the user name and password through a Java properties object or directly through the `getConnection` method call. This applies regardless of which client-side Oracle JDBC driver you are using, but is irrelevant if you are using the server-side internal driver, which uses a special direct connection and does not require a user name or password.

Starting with Oracle Database 12*c* Release 1 (12.1.0.2), the Oracle JDBC Thin driver supports the `O7L_MR` client ability when you are running your application with a JDK such as JDK 8, which supports the `PBKDF2-SHA2` algorithm. If you are running an application with JDK 7, then you must add a third-party security provider that supports the `PBKDF2-SHA2` algorithm, otherwise the driver will not support the new `12a` password verifier that requires the `O7L_MR` client ability.

If you are using Oracle Database 12*c* Release 1 (12.1.0.2) with the `SQLNET.ALLOWED_LOGON_VERSION_SERVER` parameter set to `12a`, then keep the following points in mind:

- You must also use the 12.1.0.2 Oracle JDBC Thin driver and JDK 8 or JDK 7 with a third-party security provider that supports the `PBKDF2-SHA2` algorithm
- If you use an earlier version of Oracle JDBC Thin driver, then you will get the following error:

  ```
  ORA-28040: No matching authentication protocol
  ```

- If you use the 12.1.0.2 Oracle JDBC Thin driver with JDK 7, then also you will get the same error, if you do not add a third-party security provider that supports the `PBKDF2-SHA2` algorithm.

9.5 Support for Strong Authentication

Oracle Advanced Security enables Oracle Database users to authenticate externally. External authentication can be with RADIUS, Kerberos, Certificate-Based Authentication, Token Cards, and Smart Cards. This is called strong authentication. Oracle JDBC drivers provide support for the following strong authentication methods:

- Kerberos
- RADIUS
- TLS

See Also:

Oracle Database Net Services Reference

9.6 Support for Network Encryption and Integrity

The section describes the support for network encryption and integrity.

Note:

- Starting with Oracle Database 21c, older encryption and hashing algorithms are deprecated.

  The deprecated algorithms for `DBMS_CRYPTO` and native network encryption include MD4, MD5, DES, 3DES, and RC4-related algorithms as well as 3DES for Transparent Data Encryption (TDE). Removing older, less secure cryptography algorithms prevents accidental use of these algorithms. To meet your security requirements, Oracle recommends that you use more modern cryptography algorithms, such as the Advanced Encryption Standard (AES).
- Oracle provides a patch that you can download to address necessary security enhancements that affect native network encryption environments in Oracle Database release 11.2 and later. This patch is available in My Oracle Support note 2118136.2.

See Also:

Oracle Database Security Guide

This section describes the following concepts:

- Overview of JDBC Support for Data Encryption and Integrity

- JDBC OCI Driver Support for Encryption and Integrity
- JDBC Thin Driver Support for Encryption and Integrity
- Setting Encryption and Integrity Parameters in Java

9.6.1 Overview of JDBC Support for Network Encryption and Integrity

You can use Oracle Database and Oracle Advanced Security network encryption and integrity features in your Java database applications, depending on related settings in the server. When using the JDBC OCI driver, set parameters as you would in any Oracle client situation. When using the Thin driver, set parameters through a Java properties object.

Encryption is enabled or disabled based on a combination of the client-side encryption-level setting and the server-side encryption-level setting. Similarly, integrity is enabled or disabled based on a combination of the client-side integrity-level setting and the server-side integrity-level setting.

Encryption and integrity support the same setting levels, `REJECTED`, `ACCEPTED`, `REQUESTED`, and `REQUIRED`. Table 9-1 shows how these possible settings on the client-side and server-side combine to either enable or disable the feature. By default, remote OS authentication (through TCP) is disabled in the database for security reasons.

Table 9-1 Client/Server Negotiations for Encryption or Integrity

Client/Server Settings Matrix	Client Rejected	Client Accepted (default)	Client Requested	Client Required
Server Rejected	OFF	OFF	OFF	connection fails
Server Accepted (default)	OFF	OFF	ON	ON
Server Requested	OFF	ON	ON	ON
Server Required	connection fails	ON	ON	ON

Table 9-1 shows, for example, that if encryption is requested by the client, but rejected by the server, it is disabled. The same is true for integrity. As another example, if encryption is accepted by the client and requested by the server, it is enabled. The same is also true for integrity.

See Also:

Oracle Database Security Guide for more information about network encryption and integrity features

Note:

The term checksum still appears in integrity parameter names, but is no longer used otherwise. For all intents and purposes, checksum and integrity are synonymous.

9.6.2 JDBC OCI Driver Support for Encryption and Integrity

If you are using the JDBC OCI driver, which presumes an Oracle-client setting with an Oracle client installation, then you can enable or disable network encryption or integrity and set related parameters as you would in any Oracle client situation, through settings in the `sqlnet.ora` file on the client.

> **Note:**
>
> Starting from Oracle Database 12*c* Release 1 (12.1), Oracle recommends you to use the configuration parameters present in the new XML configuration file `oraaccess.xml` instead of the OCI-specific configuration parameters present in the `sqlnet.ora` file. However, the configuration parameters present in the `sqlnet.ora` file are still supported.

To summarize, the client parameters are shown in Table 9-2:

Table 9-2 OCI Driver Client Parameters for Encryption and Integrity

Parameter Description	Parameter Name	Possible Settings
Client encryption level	`SQLNET.ENCRYPTION_CLIENT`	`REJECTED ACCEPTED REQUESTED REQUIRED`
Client encryption selected list	`SQLNET.ENCRYPTION_TYPES_CLIENT`	`AES128, AES192, AES256`
Client integrity level	`SQLNET.CRYPTO_CHECKSUM_CLIENT`	`REJECTED ACCEPTED REQUESTED REQUIRED`
Client integrity selected list	`SQLNET.CRYPTO_CHECKSUM_TYPES_CLIENT`	`SHA-1`

9.6.3 JDBC Thin Driver Support for Encryption and Integrity

The JDBC Thin driver support for network encryption and integrity parameter settings parallels the JDBC OCI driver support discussed in the preceding section. You can set the corresponding parameters through a Java properties object that you can use while opening a database connection.

The default value for the encryption and integrity level is `ACCEPTED` for both the server side and the client side. This enables you to achieve the desired security level for a connection pair by configuring only one side of a connection, either the server side or the client side. This increases the efficiency of your program because if there are multiple Oracle clients connecting to an Oracle Server, then you need to change the encryption and integrity level to `REQUESTED` in the `sqlnet.ora` file only on the server side to turn on encryption or integrity for all connections. This saves time and effort because you do not have to change the settings for each client separately.

Following is the list of parameters for the JDBC Thin driver, which are defined in the `oracle.jdbc.OracleConnection` interface:

- The CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL Parameter
- The CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES Parameter

- The CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL Parameter
- The CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES Parameter
- The CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_SERVICES Parameter

Note:

- Oracle Advanced Security support for the Thin driver is incorporated directly into the JDBC classes JAR file. So, there is no separate version for domestic and export editions. Only parameter settings that are suitable for an export edition are possible.

9.6.3.1 The CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL Parameter

The `CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL` parameter defines the level of security that the client uses to negotiate with the server.

The following table describes the attributes of this parameter:

Table 9-3 CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL Attributes

Attribute	Value
Parameter Type	String
Parameter Class	Static
Permitted Values	`REJECTED ACCEPTED REQUESTED REQUIRED`
Default Value	`ACCEPTED`
Syntax	`prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL,`*`level`*`);` where `prop` is an object of the `Properties` class
Example	`prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL,"REQUIRED");` where `prop` is an object of the `Properties` class

9.6.3.2 The CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES Parameter

The `CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES` parameter defines the encryption algorithm that you should use.

The following table describes the attributes of this parameter:

Table 9-4 CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES Attributes

Attribute	Description
Parameter Type	String
Parameter Class	Static

Table 9-4 (Cont.) CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES Attributes

Attribute	Description
Permitted Values	`AES256` (AES 256-bit key), `AES192` (AES 192-bit key), `AES128` (AES 128-bit key),
Syntax	`prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES,`*`algorithm`*`);` where `prop` is an object of the `Properties` class
Example	`prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES, "( AES256, AES192 )");` where `prop` is an object of the `Properties` class

9.6.3.3 The CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL Parameter

The `CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL` parameter defines the level of security to negotiate with the server for data integrity.

The following table describes the attributes of this parameter:

Table 9-5 CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL Attributes

Attribute	Description
Parameter Type	String
Parameter Class	Static
Permitted Values	`REJECTED`; `ACCEPTED`; `REQUESTED`; `REQUIRED`
Default Value	`ACCEPTED`
Syntax	`prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL,`*`level`*`);` where `prop` is an object of the `Properties` class
Example	`prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL,"REQUIRED");` where `prop` is an object of the `Properties` class

9.6.3.4 The CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES Parameter

The `CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES` parameter defines the data integrity algorithm to be used.

The following table describes the attributes of this parameter.

Table 9-6 CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES Attributes

Attribute	Description
Parameter Type	String

Table 9-6 (Cont.) CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES Attributes

Attribute	Description
Parameter Class	Static
Permitted Values	`SHA1`
Syntax	`prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES, *algorithm*);` where `prop` is an object of the `Properties` class
Example	`prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES,"( SHA1 )");` where `prop` is an object of the `Properties` class

9.6.3.5 The CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_SERVICES Parameter

The `CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_SERVICES` parameter determines the authentication service to be used.

The following table describes the attributes of this parameter:

Table 9-7 CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_SERVICES Attributes

Attribute	Description
Parameter Type	String
Parameter Class	Static
Permitted Values	`RADIUS, KERBEROS5, BEQ, TCPS`
Syntax	`prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_SERVICES,*authentication*);` where `prop` is an object of the `Properties` class
Example	`prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_SERVICES,"( RADIUS, KERBEROS5,TCPS)")` where `prop` is an object of the `Properties` class

9.6.4 Setting Encryption and Integrity Parameters in Java

Use a Java properties object, that is, an instance of `java.util.Properties`, to set the network encryption and integrity parameters supported by the JDBC Thin driver.

The following example instantiates a Java properties object, uses it to set each of the parameters in Table 9-3, and then uses the properties object in opening a connection to the database:

```
...
Properties prop = new Properties();
```

```
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVEL,
"REQUIRED");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPES,
"( AES256 )");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL,
"REQUESTED");
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES,
"( SHA1 )");

OracleDataSource ods = new OracleDataSource();
ods.setConnectionProperties(prop);
ods.setURL("jdbc:oracle:thin:@localhost:5221:main");
Connection conn = ods.getConnection();
...
```

The parentheses around the values encryption type and checksum type allow for lists of values. When multiple values are supplied, the server and the client negotiate to determine which value is to be actually used.

Example

Example 9-3 is a complete class that sets network encryption and integrity parameters before connecting to a database to perform a query.

Note:

In the example, the string `REQUIRED` is retrieved dynamically through the functionality of the `AnoServices` and `Service` classes. You have the option of retrieving the strings in this manner or including them in the software code as shown in the previous examples.

Before running this example, you must turn on encryption in the `sqlnet.ora` file. For example, the following lines will turn on AES256, AES192, and AES128 for the encryption and SHA1 for the checksum:

```
  SQLNET.ENCRYPTION_SERVER = ACCEPTED
  SQLNET.CRYPTO_CHECKSUM_SERVER = ACCEPTED
  SQLNET.CRYPTO_CHECKSUM_TYPES_SERVER = (SHA1)
  SQLNET.ENCRYPTION_TYPES_SERVER = (AES256, AES192, AES128)
```

Example 9-3 Setting Network Encryption and Integrity Parameters

```
import java.sql.*;
import java.util.Properties;
import oracle.net.ano.AnoServices;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;

public class DemoAESAndSHA1
{
  static final String USERNAME= "HR";
  static final String PASSWORD= "hr";
  static final String URL =
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=localhost)
(PORT=5221))"
      +"(CONNECT_DATA=(SERVICE_NAME=orcl)))";
  public static final void main(String[] argv)
  {
    DemoAESAndSHA1 demo = new DemoAESAndSHA1();
    try
    {
      demo.run();
    }catch(SQLException ex)
    {
      ex.printStackTrace();
    }
  }

  void run() throws SQLException
  {
    OracleDataSource ods = new OracleDataSource();
    Properties prop = new Properties();

    // We require the connection to be encrypted with either AES256 or AES192.
    // If the database does not accept such a security level, then the
connection attempt will fail.

prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_LEVE
L, AnoServices.ANO_REQUIRED);

prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_ENCRYPTION_TYPE
S, "( " + AnoServices.ENCRYPTION_AES256 + "," + AnoServices.ENCRYPTION_AES192
+ ")");

    // We also require the use of the SHA1 algorithm for network integrity
checking.

prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_LEVEL,
 AnoServices.ANO_REQUIRED);

prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_CHECKSUM_TYPES,
 "( " + AnoServices.CHECKSUM_SHA1 + " )");
    prop.setProperty("user", DemoAESAndSHA1.USERNAME);
    prop.setProperty("password", DemoAESAndSHA1.PASSWORD);
    ods.setConnectionProperties(prop);
    ods.setURL(DemoAESAndSHA1.URL);
    OracleConnection oraConn = (OracleConnection) ods.getConnection();
    System.out.println("Connection created! Encryption algorithm is: " +
oraConn.getEncryptionAlgorithmName() + ", network integrity algorithm is: " +
oraConn.getDataIntegrityAlgorithmName());
    oraConn.close();
  }

}
```

9.7 Support for TLS

This section describes the following topics:

9.7.1 Overview of JDBC Support for TLS

Oracle Database 23ai provides support for the Transport Layer Security (TLS) protocol. TLS is a widely used industry standard protocol that provides secure communication over a network. TLS provides authentication, data encryption, and data integrity. It provides a secure enhancement to the standard TCP/IP protocol, which is used for Internet communication.

TLS uses digital certificates that comply with the X.509v3 standard for authentication and a public and private key pair for encryption. TLS also uses secret key cryptography and digital signatures to ensure privacy and integrity of data. When a network connection over TLS is initiated, the client and server perform a TLS handshake that includes the following steps:

- Client and server negotiate about the cipher suites to use. This includes deciding on the encryption algorithms to be used for data transfer.
- Server sends its certificate to the client, and the client verifies that the certificate was signed by a trusted certification authority (CA). This step verifies the identity of the server.
- If client authentication is required, the client sends its own certificate to the server, and the server verifies that the certificate was signed by a trusted CA.
- Client and server exchange key information using public key cryptography. Based on this information, each generates a session key. All subsequent communications between the client and the server is encrypted and decrypted by using this set of session keys and the negotiated cipher suite.

TLS Terminology

The following terms are commonly used in the TLS context:

- **Certificate**: A certificate is a digitally signed document that binds a public key with an entity. The certificate can be used to verify that the public key belongs to that individual.
- **Certification authority**: A certification authority (CA), also known as certificate authority, is an entity which issues digitally signed certificates for use by other parties.
- **Cipher suite**: A cipher suite is a set of cryptographic algorithms and key sizes used to encrypt data sent over a TLS-enabled network.
- **Private key**: A private key is a secret key, which is never transmitted over a network. The private key is used to decrypt a message that has been encrypted using the corresponding public key. It is also used to sign certificates. The certificate is verified using the corresponding public key.
- **Public key**: A public key is an encryption key that can be made public or sent by ordinary means such as an e-mail message. The public key is used for encrypting the message sent over TLS. It is also used to verify a certificate signed by the corresponding private key.

- **Key Store or Wallet**: 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 TLS.
- **Security Provider**: A Java implementation that provides some functionality related to security. A provider is responsible for decoding a key store file.
- **Key Store Service (KSS)**: A component of Oracle Platform Security services. KSS enables a key store to be referenced as a URI with `kss://` scheme (rather than a file name).

Java Version of TLS

The Java Secure Socket Extension (JSSE) provides a framework and an implementation for a Java version of the TLS protocol. JSSE provides support for data encryption, server and client authentication, and message integrity. It abstracts the complex security algorithms and handshaking mechanisms and simplifies application development by providing a building block for application developers, which they can directly integrate into their applications. JSSE is integrated into Java Development Kit (JDK) 1.4 and later, and supports TLS version 2.0 and 3.0.

Oracle strongly recommends that you have a clear understanding of the JavaTM Secure Socket Extension (JSSE) framework before using TLS in the Oracle JDBC drivers.

The JSSE standard application programming interface (API) is available in the `javax.net`, `javax.net.ssl`, and `javax.security.cert` packages. These packages provide classes for creating and configuring sockets, server sockets, TLS sockets, and TLS server sockets. The packages also provide a class for secure HTTP connections, a public key certificate API compatible with JDK1.1-based platforms, and interfaces for key and trust managers.

TLS works the same way, as in any networking environment, in Oracle Database 18c.

Note:

In order to use JSSE in your program, you must have clear understanding of JavaTM Secure Socket Extension (JSSE) framework.

9.7.2 About Managing Certificates and Wallets

To establish a TLS connection with a JDBC client, either Thin or OCI, the Oracle Database server sends its certificate, which is stored in its wallet. The client may or may not need a certificate or wallet, depending on the server configuration.

Note:

- The Oracle JDBC Thin driver uses the JSSE framework to create a TLS connection. It uses the default provider (*SunJSSE*) to create a TLS context, but you can provide your own provider.
- You do not need a certificate for the client, unless the `SSL_CLIENT_AUTHENTICATION` parameter is set on the server.

This client certificate may be in an Oracle wallet, Microsoft Certificate Store (MCS), Java key store, or Linux `certs` folder (`/etc/ssl/certs`). When multiple certificates are present, then the correct client certificate for a connection is pulled by the client in either of the following two ways:

- **Using the certificate alias:** In this case, the client certificate can be uniquely identified by its alias. Starting from Oracle Database 23ai Release, you can configure the alias using the `SSL_CERTIFICATE_ALIAS SECURITY` parameter in the connection string.
- **Using the certificate thumbprint:** In this case, the client certificate can be uniquely identified by its thumbprint. Starting from Oracle Database 23ai Release, you can set the thumbprint using the `SSL_CERTIFICATE_THUMBPRINT SECURITY` parameter in the connection string. You can also set it using either the `SSL_CERTIFICATE_THUMBPRINT` parameter of the Easy Connect Plus URL or the `CONNECTION_PROPERTY_THIN_SSL_CERTIFICATE_THUMBPRINT` JDBC property.

See Also:

- SSL_CLIENT_AUTHENTICATION Parameter
- CONNECTION_PROPERTY_THIN_SSL_CERTIFICATE_ALIAS
- SSL_CERTIFICATE_THUMBPRINT Parameter
- CONNECTION_PROPERTY_THIN_SSL_CERTIFICATE_THUMBPRINT

9.7.3 About Keys and certificates containers

Java clients can use multiple types of containers such as Oracle wallets, JKS, PKCS12, and so on, as long as a provider is available. For Oracle wallets, *OraclePKI* provider must be used because the PKCS12 support provided by *SunJSSE* provider does not support all the features of PKCS12. In order to use the `OraclePKI` provider, you must have the `oraclepki.jar` file under the `$ORACLE_HOME/jlib` directory.

9.7.4 Database Connectivity with TLS Version 1.3 Using the JDBC Thin Driver

This section describes the steps to configure the Oracle JDBC thin driver to connect to the Database using TLS v1.3.

Note:

- Starting from JDK 8 (JDK 8u341), TLS 1.3 version is auto enabled.
- Oracle recommends not to configure the protocol version and let the JDK implementation choose the most secure version of the protocol. If you must choose the protocol version for a specific need, then you can achieve it in one of the following ways:
  - Using connection property: Set `CONNECTION_PROPERTY_THIN_SSL_VERSION={1.3, 1.2}`
  - Using system property: Set `-Doracle.net.ssl_version="1.3", "1.2"`
  - Using the connection descriptor: Use `(security=(ssl_server_dn_match=yes)(SSL_VERSION=1.3))` in the connection URL

See Also:

Oracle Database JDBC Java API Reference for more information

- Always use the latest update of the JDK

  Use the latest update of JDK 17, JDK 11, or JDK 8 because the updated versions include bug fixes that are required for using TLS version 1.3.
- Use JKS files or Oracle Wallet

  Note:

  Starting from Oracle Database Release 18c, you can specify TLS configuration properties in a new configuration file called `ojdbc.properties`. The use of this file eases the connectivity to Database services on Cloud.

  See Also:

  Oracle Database JDBC Java API Reference

After performing all the preceding steps, if you run into more issues, then you can turn on tracing to diagnose the problems using `-Djavax.net.debug=all` option.

9.7.5 Automatic TLS Connection Configuration

Starting from Oracle Database Release 18c, you can use default values or programmatic logic for resolving the connection configuration values without manually adding or updating the security provider. You can resolve the configuration values in the following two ways:

- Provider Resolution
- Automatic Key Store Type (KSS) Resolution

9.7.5.1 Provider Resolution

For certain key store types, the JDBC driver resolves the provider implementation that is used to load the key store. For these types, it is not necessary to register the provider with Java security. If the provider implementation is on the `CLASSPATH`, the driver can instantiate the security provider.

The following key store types map to a known provider:

- **SSO:** `oracle.security.pki.OraclePKIProvider`
- **KSS:** `oracle.security.jps.internal.keystore.provider.FarmKeyStoreProvider`

The driver attempts to resolve the provider only if there is no provider registered for the specified type.

If the `oraclepki.jar` file is on the `CLASSPATH`, then the driver can automatically load the Oracle PKI Provider in the following way:

```
java -cp oraclepki.jar:ojdbc11.jar -D javax.net.ssl.keyStore=/path/to/wallet/
cwallet.sso MyApp
```

Similarly, for a specified value of the `oracle.net.wallet_location` connection property, the driver can automatically load the Oracle PKI Provider in the following way:

```
java -cp .:oraclepki.jar:ojdbc11.jar -D oracle.net.wallet_location=file:/
path/to/wallet/cwallet.sso MyApp
```

Note:

For `PKCS12` types created by the `orapki` tool (The `ewallet.p12` file), you may still need to register the `OraclePKIProvider` with Java security because the `PKCS12` file created by the `orapki` tool includes the ASN1 *Key Bag* element (Type Code: 1.2.840.113549.1.12.10.1.1). The Sun `PKCS12` implementation does not support the *Key Bag* type and throws an error when attempting to read the `ewallet.p12` file. For HotSpot and Open JDK users, the Sun Provider comes bundled as the `PKCS12` provider. This means that the `PKCS12` provider will already have a registered provider, and the driver will make no attempt to override this.

9.7.5.2 Automatic Key Store Type (KSS) Resolution

The JDBC driver can resolve common key store types based on the value of the `javax.net.ssl.keyStore` and `javax.net.ssl.trustStore` properties, eliminating the need to specify the type using these properties.

Key Store or Trust Store with a Recognized File Extension

A key store or trust store with a recognized file extension maps to the following types:

- File extension `.jks` resolves to `javax.net.ssl.keyStoreType` as JKS:

  ```
  java -cp ojdbc11.jar -D javax.net.ssl.keyStore=/path/to/keystore/
  keystore.jks MyApp
  ```

- File extension `.sso` resolves to `javax.net.ssl.keyStoreType` as SSO:

  ```
  java -cp ojdbc11.jar -D javax.net.ssl.keyStore=/path/to/keystore/
  keystore.sso MyApp
  ```

- File extension `.p12` resolves to `javax.net.ssl.keyStoreType` as PKCS12:

  ```
  java -cp ojdbc11.jar -D javax.net.ssl.keyStore=/path/to/keystore/
  keystore.p12 MyApp
  ```

- File extension `.pfx` resolves to `javax.net.ssl.keyStoreType` as PKCS12:

  ```
  java -cp ojdbc11.jar -D javax.net.ssl.keyStore=/path/to/keystore/
  keystore.pfx MyApp
  ```

Key Store or Trust Store with a URI

If the key store or the trust store is a URI with a `kss://` scheme, this maps to type KSS:

```
java -cp ojdbc11.jar -D javax.net.ssl.keyStore=kss://MyStripe/MyKeyStore MyApp
```

Note:

You can set the `javax.net.ssl.trustStoreType` and `javax.net.ssl.keyStoreType` properties for overriding the default type resolution.

9.7.6 Support for Default TLS Context

For applications that require finer control over the TLS configuration, you can configure the JDBC driver to use the `SSLContext` returned by the `SSLContext.getDefault` method. Use one of the following methods for the driver to use the default `SSLContext`:

- `javax.net.ssl.keyStore=NONE`
- `javax.net.ssl.trustStore=NONE`

You can use the default `SSLContext` to support key store types that are not file-based. Common examples of such key store types include hardware-based smart cards. Key store types that require programmatic call to the `load(KeyStore.LoadStoreParameter)` method also belong to this category.

See Also:

- https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLContext.html#getDefault--
- https://docs.oracle.com/javase/8/docs/api/java/security/KeyStore.html#load-java.security.KeyStore.LoadStoreParameter-

9.7.6.1 Support for Caching SSLContext Instance

Starting from Oracle Database Release 23ai (23.6.0.0.0), the JDBC Thin driver supports caching SSLContext instances.

When a connection requires an SSLContext for establishing a TLS session with an Oracle Database instance, it queries the SSLContextCache with a SSLConfig instance (key). If an SSLContext exists for an identical SSLConfig, then it is returned to the connection. Otherwise, a new SSLContext is created, cached, and returned to the connection.

The SSLContextCache stores the following information about the SSLContext instances:

- Full path of the files (keystore, truststore, or wallet files) used in the SSLContext creation
- The last modified time of each of the previously-mentioned files
- The last used timestamp of the SSLContext instance
- The name of the providers used for creating the SSLContext instance

Caching of the SSL Context instances provides the following benefits:

- **Transport Layer Security (TLS) Session Resumption Support:** The TLS Session resumption works only if the same SSLContext instance is used for creating various SSL Sessions (SSLEngine). The session cache is maintained with the SSLContext instance.
- **Performance Improvement:** Caching the SSLContext instances improves performance because it avoids reading the truststore and keystore from the file system for each connection. Connections with similar TLS configuration can reuse the same SSLContext instance.

The default size of the SSLContextCache is 15. If it reaches its maximum allowed size, then the least recently used entry is removed from the cache before adding a new entry. You can modify the cache size by using the `oracle.net.sslContextCacheSize` system property. Caching is disabled if you set this property to `0` (zero).

See Also:

Oracle® Database JDBC Java API Reference

9.7.7 SNI Support for TLS Connections

Starting from Oracle Database Release 23ai (23.7), the JDBC Thin driver supports Server Name Indication (SNI). SNI is a TLS extension that is used to send an unencrypted host name in a `ClientHello` message.

For every TLS connection establishment, you must perform a TLS handshake with both the listener and the foreground database process, which delays the overall connection establishment time.

Using SNI, the client can send the information of the target database service in an unencrypted form in the TLS `ClientHello` message. The listener uses this information to hand-off the request to the foreground database process, without performing the TLS handshake. So, the connection establishment time is reduced due to absence of the TLS handshake with the listener.

You can enable SNI support using one of the following options:

- The `oracle.net.useSNI` connection property. For example:

  ```
  OracleDataSource ods = new OracleDataSource();
  ods.setURL("jdbc:oracle:thin:@tcps://mydbhost:3484/mydbservice");
  ods.setUser("myusername");
  ods.setPassword(<password>);
  ods.setConnectionProperty(OracleConnection.CONNECTION_PROPERTY_NET_USE_SNI,
   "true");
  Connection conn = ods.getConnection();
  conn.close();
  ```

- The Easy Connect URL. For example:

  ```
  jdbc:oracle:thin:@tcps://mydbhost:3484/myservice?
  wallet_location=mywallet&use_sni=true
  ```

- The connection descriptor. For example:

  ```
  jdbc:oracle:thin:@((DESCRIPTION=(USE_SNI=TRUE)(ADDRESS=(PROTOCOL=TCPS)
  (HOST=mydbhost)(PORT=3484))
  (CONNECT_DATA=(SERVICE_NAME=myservice))
  (SECURITY=(WALLET_LOCATION=mywallet)))
  ```

9.7.8 Support for Key Store Service

This release of Oracle Database introduces support for Key Store Service (KSS) in the JDBC driver. So, if you have configured a Key Store Service in a WebLogic server, then JDBC applications can now integrate with the existing Key Store Service configuration.

The driver can load the key stores that are managed by the Key Store Service. If the value of the `javax.net.ssl.keyStore` property or the `javax.net.ssl.trustStore` property is a URI with `kss://` scheme, then the driver loads the key store from Key Store Service.

For permission-based protection, the following permission must be granted to the ojdbc JAR file:

```
permission KeyStoreAccessPermission "stripeName=*,keystoreName=*,alias=*",
"read";
```

This permission grants access to every key store. For limiting the scope of access, you can replace the asterisk wild cards (*) with a specific application stripe and a key store name. The driver does not load the key store as a privileged action, which means that the `KeyStoreAccessPermission` must also be granted to the application code base.

9.8 Support for Kerberos

This section discusses the following topics:

- Overview of JDBC Support for Kerberos
- Configuring Windows to Use Kerberos
- Configuring Oracle Database to Use Kerberos
- Code Example for Using Kerberos
- Support for Kerberos Constrained Delegation
- Kerberos Authentication Enhancements

9.8.1 Overview of JDBC Support for Kerberos

Kerberos is a network authentication protocol that provides the tools of authentication and strong cryptography over the network. Kerberos helps you secure your information systems across your entire enterprise by using secret-key cryptography. The Kerberos protocol uses strong cryptography so that a client or a server can prove its identity to its server or client across an insecure network connection. After a client and server have used Kerberos to prove their identity, they can also encrypt all of their communications to assure privacy and data integrity as they go about their business.

The Kerberos architecture is centered around a trusted authentication service called the key distribution center, or KDC. Users and services in a Kerberos environment are referred to as principals; each principal shares a secret, such as a password, with the KDC. A principal can be a user such as `HR` or a database server instance.

Starting from 12*c* Release 1, Oracle Database supports cross-realm authentication for Kerberos. If you add the referred realm appropriately in the `domain_realms` section of the kerberos configuration file, then being in one particular realm, you can access the services of another realm.

Starting from Release 19c, Oracle Database supports Kerberos Constrained Delegation. This feature added a new method `OracleConnectionBuilder gssCredential(GSSCredential credential)` in the `oracle.jdbc.OracleConnectionBuilder` interface. This method accepts the GSSCredential of the user and then delegates it during the Kerberos authentication in the driver.

See Also:

Oracle Database JDBC Java API Reference

9.8.2 Configuring Windows to Use Kerberos

A good Kerberos client providing `klist`, `kinit`, and other tools, can be found at the following link:

http://web.mit.edu/kerberos/dist/index.html

This client also provides a nice GUI.

You need to make the following changes to configure Kerberos on your Windows machine:

1. Right-click the **My Computer** icon on your desktop.
2. Select **Properties**. The System Properties dialog box is displayed.
3. Select the **Advanced** tab.
4. Click **Environment Variables**. The Environment Variables dialog box is displayed.
5. Click **New** to add a new user variable. The New User Variable dialog box is displayed.
6. Enter `KRB5CCNAME` in the Variable name field.
7. Enter `FILE:C:\Documents and Settings\<user_name>\krb5cc` in the Variable value field.
8. Click **OK** to close the New User Variable dialog box.
9. Click **OK** to close the Environment Variables dialog box.
10. Click **OK** to close the System Properties dialog box.

Note:

`C:\WINDOWS\krb5.ini` file has the same content as `krb5.conf` file.

9.8.3 Configuring Oracle Database to Use Kerberos

Perform the following steps to configure Oracle Database to use Kerberos:

1. Use the following command to connect to the database:

```
SQL> connect system
Enter password: password
```

2. Use the following commands to create a user `CLIENT@MYORACLE.COM` that is identified externally:

```
SQL> create user "CLIENT@MYORACLE.COM" identified externally;
SQL> grant create session to "CLIENT@MYORACLE.COM";
```

3. Use the following commands to connect to the database as `sysdba` and dismount it:

```
SQL> connect / as sysdba
SQL> shutdown immediate;
```

4. Add the following line to `$T_WORK/t_init1.ora` file:

```
OS_AUTHENT_PREFIX=""
```

5. Use the following command to restart the database:

```
SQL> startup pfile=t_init1.ora
```

6. Modify the `sqlnet.ora` file to include the following lines:

   ```
   names.directory_path = (tnsnames)
   #Kerberos
   sqlnet.authentication_services = (beq,kerberos5)
   sqlnet.authentication_kerberos5_service = dbji
   sqlnet.kerberos5_conf = /home/Jdbc/Security/kerberos/krb5.conf
   sqlnet.kerberos5_keytab = /home/Jdbc/Security/kerberos/dbji.oracleserver
   sqlnet.kerberos5_conf_mit = true
   sqlnet.kerberos_cc_name = /tmp/krb5cc_5088
   # logging (optional):
   trace_level_server=16
   trace_directory_server=/scratch/sqlnet/
   ```

7. Use the following commands to verify that you can connect through SQL*Plus:

   ```
   > kinit client
   > klist
        Ticket cache: FILE:/tmp/krb5cc_5088
        Default principal: client@MYORACLE.COM

        Valid starting     Expires            Service principal
        06/22/06 07:13:29  06/22/06 17:13:29  krbtgt/MYORACLE.COM@MYORACLE.COM


        Kerberos 4 ticket cache: /tmp/tkt5088
        klist: You have no tickets cached
   > sqlplus '/@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=oracleserver.mydomain.com)
   (PORT=5221))
   (CONNECT_DATA=(SERVICE_NAME=orcl)))'
   ```

9.8.4 Code Example for Using Kerberos

This following example demonstrates the Kerberos authentication feature that is part of Oracle Database 23ai JDBC thin driver. This demo covers two scenarios:

- In the first scenario, the OS maintains the user name and credentials. The credentials are stored in the cache and the driver retrieves the credentials before trying to authenticate to the server. This scenario is in the module `connectWithDefaultUser()`.

  Note:

  1. Before you run this part of the demo, use the following command to verify that you have valid credentials:

     ```
     > /usr/kerberos/bin/kinit client
     where, the password is welcome.
     ```

  2. Use the following command to list your tickets:

     ```
     > /usr/kerberos/bin/klist
     ```

- The second scenario covers the case where the application wants to control the user credentials. This is the case of the application server where multiple web users have their own credentials. This scenario is in the module `connectWithSpecificUser()`.

Note:

To run this demo, you need to have a working setup, that is, a Kerberos server up and running, and an Oracle database server that is configured to use Kerberos authentication. You then need to change the URLs used in the example to compile and run it.

Example 9-4 Using Kerberos Authentication to Connect to the Database

```
import com.sun.security.auth.module.Krb5LoginModule;
import java.io.IOException;

import java.security.PrivilegedExceptionAction;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import java.util.HashMap;
import java.util.Properties;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;

import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import oracle.net.ano.AnoServices;
public class KerberosJdbcDemo
{
  String url ="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)"+
    "(HOST=oracleserver.mydomain.com)(PORT=5221))(CONNECT_DATA=" +
    "(SERVICE_NAME=orcl)))";

  public static void main(String[] arv)
  {
    /* If you see the following error message [Mechanism level: Could not load
     * configuration file c:\winnt\krb5.ini (The system cannot find the path
     * specified] it's because the JVM cannot locate your kerberos config file.
     * You have to provide the location of the file. For example, on Windows,
     * the MIT Kerberos client uses the config file: C\WINDOWS\krb5.ini:
     */
    // System.setProperty("java.security.krb5.conf","C:\\WINDOWS\\krb5.ini");
    System.setProperty("java.security.krb5.conf","/home/Jdbc/Security/kerberos/
krb5.conf");

    KerberosJdbcDemo kerberosDemo = new KerberosJdbcDemo();
    try
    {
      System.out.println("Attempt to connect with the default user:");
      kerberosDemo.connectWithDefaultUser();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
    try
    {
      System.out.println("Attempt to connect with a specific user:");
      kerberosDemo.connectWithSpecificUser();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }


  void connectWithDefaultUser() throws SQLException
  {
    OracleDriver driver = new OracleDriver();
    Properties prop = new Properties();


prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_SERVICES,
      "("+AnoServices.AUTHENTICATION_KERBEROS5+")");

prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_KRB5_MUTUAL
,
      "true");

    /* If you get the following error [Unable to obtain Principal Name for
     * authentication] although you know that you have the right TGT in your
     * credential cache, then it's probably because the JVM can't locate your
     * cache.

     *
     * Note that the default location on windows is "C:\Documents and
Settings\krb5cc_username".
     */

    //
prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_KRB5_CC_NAM
E,

    /*
      On Linux:
         > which kinit
         /usr/kerberos/bin/kinit
         > ls -l /etc/krb5.conf
         lrwxrwxrwx    1 root  root   47 Jun 22 06:56 /etc/krb5.conf -> /home/Jdbc/
Security/kerberos/krb5.conf

         > kinit client
         Password for client@MYORACLE.COM:
         > klist
         Ticket cache: FILE:/tmp/krb5cc_5088
         Default principal: client@MYORACLE.COM

         Valid starting     Expires            Service principal
         11/02/06 09:25:11  11/02/06 19:25:11  krbtgt/MYORACLE.COM@MYORACLE.COM


         Kerberos 4 ticket cache: /tmp/tkt5088
         klist: You have no tickets cached
    */

prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_KRB5_CC_NAM
E,
                     "/tmp/krb5cc_5088");
    Connection conn  = driver.connect(url,prop);
    String auth = ((OracleConnection)conn).getAuthenticationAdaptorName();
    System.out.println("Authentication adaptor="+auth);
    printUserName(conn);
    conn.close();
  }


  void connectWithSpecificUser() throws Exception
  {
    Subject specificSubject = new Subject();

    // This first part isn't really meaningful to the sake of this demo. In
    // a real world scenario, you have a valid "specificSubject" Subject that
    // represents a web user that has valid Kerberos credentials.
    Krb5LoginModule krb5Module = new Krb5LoginModule();
    HashMap sharedState = new HashMap();
    HashMap options = new HashMap();
    options.put("doNotPrompt","false");
    options.put("useTicketCache","false");
    options.put("principal","client@MYORACLE.COM");

    krb5Module.initialize(specificSubject,newKrbCallbackHandler(),sharedState,options);
    boolean retLogin = krb5Module.login();
    krb5Module.commit();
    if(!retLogin)
      throw new Exception("Kerberos5 adaptor couldn't retrieve credentials (TGT) from 
the cache");

    // to use the TGT from the cache:
    // options.put("useTicketCache","true");
    // options.put("doNotPrompt","true");
    // options.put("ticketCache","C:\\Documents and Settings\\user\\krb5cc");
    // krb5Module.initialize(specificSubject,null,sharedState,options);


    // Now we have a valid Subject with Kerberos credentials. The second scenario
    // really starts here:
    // execute driver.connect(...) on behalf of the Subject 'specificSubject':
    Connection conn =
      (Connection)Subject.doAs(specificSubject, new PrivilegedExceptionAction()
        {
          public Object run()
          {
            Connection con = null;
            Properties prop = new Properties();
            prop.setProperty(AnoServices.AUTHENTICATION_PROPERTY_SERVICES,
                             "(" + AnoServices.AUTHENTICATION_KERBEROS5 + ")");
            try
            {
              OracleDriver driver = new OracleDriver();
              con = driver.connect(url, prop);

            } catch (Exception except)
            {
              except.printStackTrace();
            }
            return con;
          }
        });

    String auth = ((OracleConnection)conn).getAuthenticationAdaptorName();
    System.out.println("Authentication adaptor="+auth);
    printUserName(conn);
    conn.close();
  }

  void printUserName(Connection conn) throws SQLException
  {
    Statement stmt = null;
    try
    {
      stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("select user from dual");
      while(rs.next())
        System.out.println("User is:"+rs.getString(1));
      rs.close();
    }
    finally
    {
      if(stmt != null)
        stmt.close();
    }
  }
}

class KrbCallbackHandler implements CallbackHandler
{
 public void handle(Callback[] callbacks) throws IOException,
                                                 UnsupportedCallbackException
 {
   for (int i = 0; i < callbacks.length; i++)
   {
     if (callbacks[i] instanceof PasswordCallback)
     {
       PasswordCallback pc = (PasswordCallback)callbacks[i];
       System.out.println("set password to 'welcome'");
       pc.setPassword((new String("welcome")).toCharArray());
     } else
     {
       throw new UnsupportedCallbackException(callbacks[i],
                                             "Unrecognized Callback");
     }
   }
 }
}
```

9.8.5 Support for Kerberos Constrained Delegation

Starting from Oracle Database Release 19c, the Thin driver supports the Kerberos constrained delegation feature.

To make it a secure practice, the implementation relies on the use of the `GSSCredential` utility. For this purpose, the `OracleConnectionBuilder` interface has been enhanced, so that it can accept a `GSSCredential` object.

Example 9-5   Using Constrained Delegation

The following example shows how to implement constrained delegation in your code:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL(url);
ods.setConnectionProperties(prop);
con = ods.createConnectionBuilder()
```

```
                 .gssCredential(impersonatedGssUserCreds)
                .build();
```

See Also:

- Java Platform, Standard Edition Security Developer’s Guide
- The OracleConnectionBuilder Interface

9.8.6 Kerberos Authentication Enhancements

Starting with Oracle Database Release 23ai, Kerberos authentication does not need instantiating the `KerberosLoginModule` or the availability of Ticket Granting Ticket (TGT) in the `CredentialCache`.

In this release, you can connect to Oracle Database using the following enhanced Kerberos Authentication methods:

- Kerberos Authentication Using the User and the Password Properties
- Kerberos Authentication Using the JAAS Configuration

9.8.6.1 Kerberos Authentication Using the User and the Password Properties

Now you can configure the Kerberos Principal and Password properties in the same way as you configure the user and password properties for a simple authentication (`O5Logon`). Using these values, the JDBC Thin driver initializes the `KerberosLoginModule` for your application, simplifying the Kerberos Authentication configuration.

For using the configured user name and password, you must set the `PASSWORD_AUTH` parameter to `KERBEROS5` in the connection string. You can also set `PASSWORD_AUTH` to `KERBEROS5` using the `oracle.jdbc.passwordAuthentication` connection property. However, the value specified in the connection string has a higher priority.

Example 9-6 Kerberos Authentication: Using the User and the Password Properties

The following example shows how you can configure the Kerberos Principal and the Password:

```
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Properties;

public class KerberosExample {

  private static String KERBEROS_PRINCIPAL = "client@EXAMPLE.COM";
  private static String PASSWORD = <password>;
  private static String URL
="jdbc:oracle:thin:@(DESCRIPTION=(SECURITY=(PASSWORD_AUTH=KERBEROS5))
(ADDRESS=(PROTOCOL=TCP)"+
      "(HOST=myserver.example.com)(PORT=5221))
(CONNECT_DATA=(SERVICE_NAME=orcl)))";
```

```
  public static void main(String a[]) {
    try{
      Properties prop = new Properties();
      prop.setProperty("oracle.net.authentication_services", "(KERBEROS5)");
      prop.setProperty("user", KERBEROS_PRINCIPAL); // Kerberos Principal
      prop.setProperty("password", PASSWORD); // Kerberos Password
      OracleDriver driver = new OracleDriver();
      Connection conn  = driver.connect(URL, prop);
      String auth = ((OracleConnection)conn).getAuthenticationAdaptorName();
      System.out.println ("Authentication adaptor=" + auth);
      conn.close();
    }
    catch(SQLException e) {
      e.printStackTrace();
    }
  }

}
```

9.8.6.2 Kerberos Authentication Using the JAAS Configuration

In this release, you can specify the JAAS configuration file through the JDBC connection properties. By default, the Thin driver uses the default kerberos login module that is bundled with Oracle JDK (`com.sun.security.auth.module.Krb5LoginModule`). If you want to override this behaviour in your application, then use the JAAS configurations.

The following code example shows how you can configure the Kerberos Authentication using the JAAS configurations:

```
import java.sql.Connection;
import java.util.Properties;

import oracle.jdbc.OracleDriver;
import oracle.jdbc.OracleConnection;

public class JAASKerberosAuthentication {

  // Set the following properties
  private final static String DB_URL = "jdbc:oracle:thin:@(DESCRIPTION=" +
      "(ADDRESS=(PROTOCOL=TCP)(HOST=myserver.example.com)(PORT=5221))" +
      "(CONNECT_DATA=(SERVICE_NAME=orcl)))";
  private final static String KERBEROS_CONFIG_FILE = "/etc/krb5.conf";
  private final static String JAAS_CONFIG_FILE_PATH = "/myworkdir/jaas.conf";
  private final static String KERBEROS_LOGIN_MODULE_NAME = "kprb5module";

  public static void main(String a[]) throws Exception {
    // Set JAAS configuration

System.setProperty("java.security.auth.login.config",JAAS_CONFIG_FILE_PATH);

    // Set Kerberos Configuration
    System.setProperty("java.security.krb5.conf", KERBEROS_CONFIG_FILE);

    OracleDriver driver = new OracleDriver();
    Properties prop = new Properties();
    prop.setProperty("oracle.net.authentication_services", "(KERBEROS5)");
    prop.setProperty("oracle.net.KerberosJaasLoginModule",
KERBEROS_LOGIN_MODULE_NAME);
    Connection conn = driver.connect(DB_URL, prop);
    String auth = ((OracleConnection) conn).getAuthenticationAdaptorName();
    System.out.println("Got Connection. Authentication adaptor=" + auth);
    conn.close();
  }
}
```

The preceding example uses the following JAAS configuration file in the `/myworkdir/jaas.conf` directory:

```
kprb5module {
     com.sun.security.auth.module.Krb5LoginModule required
                doNotPrompt=true useTicketCache=true ticketCache="file:/tmp/
krb5cc_5088";
};
```

9.9 Support for RADIUS

This section describes the following concepts:

- Overview of JDBC Support for RADIUS
- Configuring Oracle Database to Use RADIUS
- Code Example for Using RADIUS
- Support for Challenge-Response Authentication

9.9.1 Overview of JDBC Support for RADIUS

Oracle Database 11*g* Release 1 introduced support for Remote Authentication Dial-In User Service (RADIUS). RADIUS is a client/server security protocol that is most widely known for enabling remote authentication and access. Oracle Advanced Security uses this standard in a client/server network environment to enable use of any authentication method that supports the RADIUS protocol. RADIUS can be used with a variety of authentication mechanisms, including token cards and smart cards.

9.9.2 Configuring Oracle Database to Use RADIUS

Perform the following steps to configure Oracle Database to use RADIUS:

1. Use the following command to connect to the database:

   ```
   SQL> connect system
   Enter password: password
   ```

2. Use the following commands to create a new user `aso` from within a database:

   ```
   SQL> create user aso identified externally;
   SQL> grant create session to aso;
   ```

3. Use the following commands to connect to the database as `sysdba` and dismount it:

   ```
   SQL> connect / as sysdba
   SQL> shutdown immediate;
   ```

4. Add the following lines to the `t_init1.ora` file:

   ```
   os_authent_prefix = ""
   ```

   > **Note:**
   >
   > Once the test is over, you need to revert the preceding changes made to the t_init1.ora file.

5. Use the following command to restart the database:

   ```
   SQL> startup pfile=?/work/t_init1.ora
   ```

6. Modify the `sqlnet.ora` file so that it contains only these lines:

   ```
   sqlnet.authentication_services = ( beq, radius)
   sqlnet.radius_authentication = <RADUIUS_SERVER_HOST_NAME>
   sqlnet.radius_authentication_port = 1812
   sqlnet.radius_authentication_timeout = 120
   sqlnet.radius_secret=/home/Jdbc/Security/radius/radius_key
   # logging (optional):
   trace_level_server=16
   trace_directory_server=/scratch/sqlnet/
   ```

7. Use the following command to verify that you can connect through SQL*Plus:

   ```
   >sqlplus 'aso/1234@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)
   (HOST=oracleserver.mydomain.com)(PORT=5221))
   (CONNECT_DATA=(SERVICE_NAME=orcl)))'
   ```

9.9.3 Code Example for Using RADIUS

This example demonstrates the new RADIUS authentication feature that is a part of Oracle Database 12*c* Release 1 (12.1) JDBC thin driver. You need to have a working setup, that is, a RADIUS server up and running, and an Oracle database server that is configured to use RADIUS authentication. You then need to change the URLs given in the example to compile and run it.

Example 9-7   Using RADIUS Authentication to Connect to the Database

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import oracle.net.ano.AnoServices;
public class RadiusJdbcDemo
{
  String url ="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)"+
    "(HOST=oracleserver.mydomain.com)(PORT=5221))(CONNECT_DATA=" +
    "(SERVICE_NAME=orcl)))";

  public static void main(String[] arv)
  {
    RadiusJdbcDemo radiusDemo = new RadiusJdbcDemo();
    try
    {
      radiusDemo.connect();
```

```
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }

  /*
   * This method attempts to logon to the database using the RADIUS
   * authentication protocol.
   *
   * It should print the following output to stdout:
   * -----------------------------------------------------
   * Authentication adaptor=RADIUS
   * User is:ASO
   * -----------------------------------------------------
   */
  void connect() throws SQLException
  {
    OracleDriver driver = new OracleDriver();
    Properties prop = new Properties();


prop.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_AUTHENTICATION_SERVICES,
      "("+AnoServices.AUTHENTICATION_RADIUS+")");
    // The user "aso" needs to be properly setup on the radius server with
    // password "1234".
    prop.setProperty("user","aso");
    prop.setProperty("password","1234");

    Connection conn  = driver.connect(url,prop);
    String auth = ((OracleConnection)conn).getAuthenticationAdaptorName();
    System.out.println("Authentication adaptor="+auth);
    printUserName(conn);
    conn.close();
  }


  void printUserName(Connection conn) throws SQLException
  {
    Statement stmt = null;
    try
    {
      stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("select user from dual");
      while(rs.next())
        System.out.println("User is:"+rs.getString(1));
      rs.close();
    }
    finally
    {
      if(stmt != null)
        stmt.close();
    }
  }
}
```

9.9.4 Support for Challenge-Response Authentication

The RADIUS challenge-response authentication is an interactive authentication, where the RADIUS server asks for a valid response to a displayed challenge. Starting from Oracle Database Release 23ai, the JDBC thin drivers support this authentication.

In challenge-response authentication, the first level of authentication is performed using the user name and the password. The RADIUS server then sends a challenge to the application, to which the application must respond. For handling the challenge, you must configure a handler in your application, which is responsible for producing the response, using a given hint. The hint is provided to the handler as a byte array and the value of the byte array is dependent on the configuration of the challenge-response authentication in the RADIUS server.

See Also:

Oracle Database Security Guide

You can configure the handler in the following two ways:

- Using the `oracle.net.radius_challenge_response_handler` connection property
- Using the `ConnectionBuilder.radiusChallengeResponseHandler` method

Using the oracle.net.radius_challenge_response_handler Connection Property

Use the `oracle.net.radius_challenge_response_handler` connection property to configure the fully qualified name of the class that handles the challenge. This handler class must implement the `java.util.function.Function<byte[], byte[]>` interface. The following example shows how to configure RADIUS challenge-response authentication using the `oracle.net.radius_challenge_response_handler` connection property.

```
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;

import java.sql.SQLException;
import java.util.Properties;
import java.util.function.Function;

public class RadiusExample {

  private static String USER = "myRadiusUser";
  private static String PASSWORD = "myRadiusPwd";
  private static String URL
="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)"+
      "(HOST=myserver.example.com)(PORT=5221))
(CONNECT_DATA=(SERVICE_NAME=orcl)))";

  public static void main(String a[]) {
    try {
      OracleDataSource ods  = new OracleDataSource();
      Properties prop = new Properties();
      prop.setProperty("oracle.net.authentication_services", "RADIUS");
      prop.setProperty("user", USER);
      prop.setProperty("password", PASSWORD);
      prop.setProperty("oracle.net.radius_challenge_response_handler",
"MyChallengeResponseHandler");
      ods.setConnectionProperties(prop);
      ods.setURL(URL);
      OracleConnection conn = (OracleConnection) ods.getConnection();
      System.out.println("Got Connection. Authentication adaptor="+
conn.getAuthenticationAdaptorName());
      conn.close();
    }
    catch(SQLException e) {
      e.printStackTrace();
    }

  }

  class MyChallengeResponseHandler implements Function<byte[], byte[]> {
    @Override
    public byte[] apply(byte[] hint) {
      // TODO: use the hint to produce the challenge response
      byte[] response = null;
      return response;
    }
  }

}
```

Using the ConnectionBuilder.radiusChallengeResponseHandler Method

You can configure the handler using the new `ConnectionBuilder.radiusChallengeResponseHandler` method. Use this method to specify the handler lambda / instance. The following example shows how to configure RADIUS challenge-response authentication using the `radiusChallengeResponseHandler` method:

```
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;

import java.sql.SQLException;
import java.util.Properties;

public class RadiusExample {

  private static String USER = "myRadiusUser";
  private static String PASSWORD = "myRadiusPwd";
  private static String URL
="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)"+
      "(HOST=myserver.example.com)(PORT=5221))
(CONNECT_DATA=(SERVICE_NAME=orcl)))";

  public static void main(String a[]) {
    try {
      OracleDataSource ods  = new OracleDataSource();
      Properties prop = new Properties();
      prop.setProperty("oracle.net.authentication_services", "RADIUS");
      prop.setProperty("user", USER);
      prop.setProperty("password", PASSWORD);
      ods.setConnectionProperties(prop);
      ods.setURL(URL);
      OracleConnection conn  = ods.createConnectionBuilder()
          .radiusChallengeResponseHandler(RadiusExample::getResponse).build();
      System.out.println("Got Connection. Authentication adaptor="+
conn.getAuthenticationAdaptorName());
      conn.close();
    }
    catch(SQLException e) {
      e.printStackTrace();
    }
  }

  private static byte[] getResponse(byte[] hint) {
    // TODO: use the hint to produce the challenge response
    byte[] response = null;
    return response;
  }

}
```

9.10 Support for FIPS with Native Network Encryption

Starting with Oracle Database Release 19c, the JDBC Thin drivers support the Federal Information Processing Standards (FIPS).

For enabling FIPS support, you should use `CONNECTION_PROPERTY_THIN_NET_SET_FIPS_MODE`. By default, this property is set to `false`. You must set it to `true`, to enable FIPS mode for native network encryption.

Note:

FIPS with TLS connection does not require this configuration.

See Also:

Oracle Database JDBC Java API Reference

Example 9-8 Enabling FIPS Mode in Your Application

The following example demonstrates how you can enable FIPS mode in your application:

```
import java.sql.*;
import java.util.*;

import oracle.jdbc.*;

import java.security.*;

import org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider;

public class TestFIPS {
  static String driver = "thin";
  static String protocol = "tcp";
  static String user = "scott";
  static String password = "<password>";
  static String url = "jdbc:oracle:" + driver +
":@(DESCRIPTION=(ADDRESS=(PROTOCOL=" + protocol + ")(HOST=" +
System.getenv("HOST") + ")(PORT=3484))
(CONNECT_DATA=(SERVICE_NAME=myservice.com))
(SECURITY=(ENCRYPTION_CLIENT=REQUESTED)))";

  public static void main(String[] args) throws Exception {

    System.setProperty("org.bouncycastle.fips.approved_only", "true");
    Provider bcFipsProvider = new BouncyCastleFipsProvider();
    Security.insertProviderAt(bcFipsProvider, 1);

    Properties props = new Properties();
    props.setProperty("user", user);
    props.setProperty("password", password);

props.setProperty(OracleConnection.CONNECTION_PROPERTY_THIN_NET_SET_FIPS_MODE,
 "true");
    show("DEBUG: Properties: " + props);

    try (Connection conn = DriverManager.getConnection(url, props);) {
      checkConn(conn);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static void checkConn(Connection conn) throws Exception {
    String sql = "select user from dual";
    try (Statement st = conn.createStatement();
         ResultSet rs = st.executeQuery(sql);){
      String user = "";
      while (rs.next()) {
        user = rs.getString(1);
      }
      show("Connect with user : " + user);
      String EncryptionAlgorithmName = ((OracleConnection)
conn).getEncryptionAlgorithmName();
      show("Connection EncryptionAlgorithmName : " + EncryptionAlgorithmName);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  private static void show(String msg) {
    System.out.println(msg);
  }

  private static void doSQL(Connection conn, String sql) throws SQLException {
    try (Statement stm = conn.createStatement()){
      stm.execute(sql);
    }
  }
}
```

9.11 Support for PEM in JDBC

Starting with Oracle Database Release 23ai, version 23.6, the JDBC thin driver supports PEM (Privacy Enhanced Mail) files, which can be used to store and transmit public key and certificate information.

You can use the `oracle.net.wallet_location` JDBC property to configure the JDBC driver with a PEM file. If the PEM file uses a password, then use the `oracle.net.wallet_password` property as well. The following are a few examples that show how to configure the driver:

```
oracle.net.wallet_location=file:/<path>/ewallet.pem
oracle.net.wallet_location=(SOURCE=(METHOD=FILE)(METHOD_DATA=(DIRECTORY=/
<path_to_directory>)))
oracle.net.wallet_location="data:;base64,<Base64_representation_of_the_bytes_i
n_ewallet.pem>"
```

Set the PEM file location in the connection string in either of the following ways:

- Using the TNS connection string:

  ```
  jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcps)(HOST=host)
  (PORT=5521))(CONNECT_DATA=
  (SERVICE_NAME=<service_name>))(SECURITY=(WALLET_LOCATION=/<mywallets>/
  ewallet.pem)))
  ```

- Using the EasyConnect URL:

  ```
  jdbc:oracle:thin:@tcps://host:port/<service_name>?wallet_location=/
  mywallets/ewallet.pem
  ```

  or

  ```
  jdbc:oracle:thin:@tcps://host:port/servicename?wallet_location=/mywallets/
  ```

9.11.1 Centralized PEM File Configuration

This section lists code snippets to describe how to centrally configure a PEM file, using unencrypted and encrypted PEM files.

With Unencrypted PEM Files

You can store the unencrypted PEM file, without making any modification to it, in a vault secret and configure the driver with the following JSON file:

```
{
  "connect_descriptor": "...",
  "user": "scott",
  "password": {
    "type": "ocivault",
    "value": "ocid1.vaultsecret.oc1.phx.amaaaaaxxxx",
```

```
      "authentication": {
        "method": "OCI_INSTANCE_PRINCIPAL"
      }
    },

    "wallet_location": {
      "type": "ocivault",
      "value": "ocid1.vaultsecret.oc1.phx.amaaaaY",
      "authentication": {
        "method": "OCI_INSTANCE_PRINCIPAL"
      }
    },
    "jdbc": {
      "oracle.jdbc.ReadTimeout": 1000,
      "defaultRowPrefetch": 20,
      "autoCommit": "false"
}}
```

With Encrypted PEM Files

You can store the encrypted PEM file, without making any modification to it, in a vault secret and configure the driver with the following JSN file:

```
{
  "connect_descriptor": "...",
  "user": "scott",
  "password": {
    "type": "ocivault",
    "value": "ocid1.vaultsecret.oc1.phx.amaaaaaaxxxx",
    "authentication": {
      "method": "OCI_INSTANCE_PRINCIPAL"
    }
  },
  "wallet_location": {
    "type": "ocivault",
    "value": "ocid1.vaultsecret.oc1.phx.amaaaaY",
    "authentication": {
      "method": "OCI_INSTANCE_PRINCIPAL"
    }
  },
  "wallet_password": {
    "type": "ocivault",
    "value": "ocid1.vaultsecret.oc1.phx.amabbbbaxxxx",
    "authentication": {
      "method": "OCI_INSTANCE_PRINCIPAL"
    }
  },
  "jdbc": {
    "oracle.jdbc.ReadTimeout": 1000,
    "defaultRowPrefetch": 20,
    "autoCommit": "false"
}}
```

9.12 Support for Secure External Password Store (SEPS)

For large-scale deployments where applications use password credentials to connect to databases, you can use a client-side Oracle wallet to store such credentials. An Oracle wallet is a secure software container that is used to store authentication and sign-in credentials.

Storing database password credentials in a client-side Oracle wallet eliminates the need to embed user names and passwords in application code, batch jobs, or scripts. This reduces the risk of exposing passwords in the scripts and application code, and simplifies maintenance because you do not need to change your code each time user names and passwords change. In addition, if you do not have to change the application code, then it also becomes easier to enforce password management policies for these user accounts.

You can store the credentials in the following ways:

- Using the connection descriptor:

  ```
  mkstore -wrl ./client_wallet -createCredential \(DESCRIPTION=\(ADDRESS=\
  (PROTOCOL=tcp\)
  \(HOST=<servername>\)\(PORT=5560\)\)\(CONNECT_DATA=\
  (SERVICE_NAME=<service_name>\)\)\) <user_name> <password>
  ```

  Where, the JDBC URL is in the following format:

  ```
  jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS(PROTOCOL=tcp)(HOST=<servername>)
  (PORT=5560))(CONNECT_DATA=(SERVICE_NAME=<service_name>)))
  ```

- Using the default keys:

  ```
  mkstore -wrl .\walletpath -createEntry
  oracle.security.client.default_username <user_name>
  mkstore -wrl .\walletpath -createEntry
  oracle.security.client.default_password <password>
  ```

- Using the `orapki` command-line interface:

  ```
  $ orapki secretstore create_entry -wallet <wallet> -pwd <password> -
  default -alias oracle.security.client.default_username -secret <username>

  $ orapki secretstore create_entry -wallet <wallet> -pwd <password> -
  default -alias oracle.security.client.default_password -secret <password>
  ```

You can set the `oracle.net.wallet_location` connection property to specify the wallet location. The JDBC driver can then retrieve the user name and password pair from this wallet.

Note:

When an Oracle wallet is opened in a JDBC application, for security reasons, the wallet file permissions are aligned to make it accessible only to the wallet owner (creator).

See Also:

- Oracle Database JDBC Java API Reference for more information about managing the secure external password store
- *Oracle Database Administrator's Guide* for more information about configuring your client to use secure external password store and for information about managing credentials in it