OpenDS

Configuring SSL and StartTLS

Version 0.1
March 2007

Contents

Overview

This document describes the process for configuring OpenDS to use SSL and StartTLS to secure communication between clients and the server. It also covers using secure communication in the client tools provided with the server.

Copyright and trademark information

Feedback

Please direct any comments or suggestions about this document to:
users@opends.dev.java.net

Acknowledgements

The general format of this document was based on the documentation template used by OpenOffice.org.

Modifications and Updates

Date	*Description of Change*
March 2007	Initial draft

Introduction

SSL (the Secure Sockets Layer) is a very common way of securing network communication. It is used by a number of different protocols, and is widely supported by clients. SSL provides a number of benefits, including:

- Confidentiality -- it encrypts the communication between the client and the server so that anyone able to observe the traffic won't actually be able to interpret it.

- Integrity -- it uses message authentication codes (MACs) to ensure that the original content was not altered in any way.

- Authentication -- the server will present its certificate to the client at the time the connection is established to verify its identity to the client, and the client may decide whether to trust that certificate. Further, the server may also request that the client present its own certificate for the server to verify.

SSL is essentially a connection-oriented protocol. Traditionally, whenever a client establishes a connection to the server in order to use SSL, the client begins the SSL negotiation process immediately and all communication over the course of the connection is secured. However, this is not an absolute requirement, and it is possible to create an encrypted tunnel over a connection that has already been used for clear-text communication. In LDAP, this is performed using the StartTLS extended operation.

In most cases, the primary visible difference between using an SSL-based connection and the StartTLS extended operation is the port that the client uses to connect to the server. For SSL-based connections, the server will have at least one listener which is dedicated to communicating with clients over SSL and will not attempt to perform any clear-text communication. When using the StartTLS extended operation, the client establishes a connection to the standard (non-secured) LDAP port and sends the StartTLS request in the clear, potentially after performing other operations in the clear. That same connection will then be converted to use an encrypted tunnel so that all subsequent communication on that connection will be protected. In addition to allowing clients to secure only a portion of their communication with the server, this is also convenient for administrators in that they only need to allow access to a single port that can be used for both clear-text and encrypted communication.

In recent years, SSL has been standardized using the name TLS (which stands for "Transport Layer Security"). However, in this document we will refer to it as "SSL" in order to avoid potential confusion between TLS and the StartTLS extended operation. OpenDS supports both the use of SSL-based connections and the StartTLS extended operation.

"Quick and Dirty" SSL Setup

OpenDS provides a number of configurable options for configuring and using SSL and StartTLS. This document attempts to describe them in detail, but it may be daunting for those who are unfamiliar with the technology or who just want to get up and running as quickly as possible for testing purposes. Therefore, this section will provide a rough step-by-step list of the steps that need to be performed in order to allow OpenDS to accept SSL-based connections using a self-signed certificate. The contents of each step should be described in more complete detail later in the document.

Assumptions:

- The Java `keytool` utility is in your path. If it is not, then you can add it to your path or provide the complete path to it when invoking the commands.

- Your current working directory is the server root directory. This is the directory containing the `setup` script and subdirectories like `bin`, `config`, and `lib`.

The process for getting OpenDS to accept SSL-based connections using a self-signed certificate is as follows:

1. Create a private key for the certificate with the command:

   ```
   $ keytool -genkey -alias server-cert -keyalg rsa \
     -dname "CN=dogmatic.central.sun.com,O=Sun Microsystems,C=US" \
     -keystore config/keystore -storetype JKS
   ```

   In this case, the value of the "`-dname`" argument should be changed so that it is suitable for your environment. The value of the CN attribute should be the fully-qualified name of the system on which the certificate is being installed, the value of the O attribute should be the name of the company or organization, and C should be the two-character country abbreviation. You will be interactively prompted for both a password to protect the contents of the keystore and a password to protect the private key. Both passwords should be the same.

2. Generate a self-signed certificate for the key with the command:

   ```
   $ keytool -selfcert -alias server-cert -validity 1825 \
     -keystore config/keystore -storetype JKS
   ```

   When you are prompted for the keystore password, enter the same password that you provided for the previous command.

3. Create a text file named `config/keystore.pin`. It should contain the password that you chose to protect the contents of the key store.

4. Export the public key for the certificate that you just created using the following command:

   ```
   $ keytool -export -alias server-cert \
     -file config/server-cert.txt -rfc -keystore config/keystore \
     -storetype JKS
   ```

5. Create a new trust store and import the server certificate into it with the following command:

   ```
   $ keytool -import -alias server-cert \
     -file config/server-cert.txt -keystore config/truststore \
     -storetype JKS
   ```

   Enter "`yes`" when you are prompted about whether or not you wish to trust the certificate.

6. Create an LDIF file with the following contents:

   ```
   dn: cn=JKS,cn=Key Manager Providers,cn=config
   changetype: modify
   replace: ds-cfg-key-manager-provider-enabled
   ds-cfg-key-manager-provider-enabled: true

   dn: cn=JKS,cn=Trust Manager Providers,cn=config
   changetype: modify
   replace: ds-cfg-trust-manager-provider-enabled
   ds-cfg-trust-manager-provider-enabled: true

   dn: cn=LDAPS Connection Handler,cn=Connection Handlers,cn=config
   changetype: modify
   replace: ds-cfg-connection-handler-enabled
   ds-cfg-connection-handler-enabled: true
   -
   replace: ds-cfg-listen-port
   ds-cfg-listen-port: 636
   ```

   If you wish to accept SSL-based connections on a port other than 636 (this is the standard LDAPS port, but you may not be able to use it if it's already taken or you're running as an unprivileged user), then simply change the `ds-cfg-listen-port` attribute in the LDIF to the desired value.

7. Apply the modifications in the LDIF file you just created using the `ldapmodify` command:

   ```
   $ bin/ldapmodify --port 389 --bindDN "cn=Directory Manager" \
     --filename /tmp/enable-ssl.ldif
   ```

   Alter the command line as necessary if your server is listening on a port other than 389, if you wish to apply the changes as a user other than "cn=Directory Manager", or if you placed the LDIF changes in a file other than `/tmp/enable-ssl.ldif`.

At this point, the server should now have a second listener created that will accept SSL-based client connections. You can test that with the `ldapsearch` command, like:

```
$ bin/ldapsearch --port 636 --useSSL --baseDN "" \
  --searchScope base "(objectClass=*)"
```

If all goes well, you will be prompted about whether you wish to trust the server's certificate, and upon entering "yes" the root DSE entry will be returned to you.

Key Manager Providers

Key manager providers are components which are ultimately responsible for providing access to the certificate that should be used by the server when performing SSL or StartTLS negotiation. OpenDS provides an interface for developing custom key manager providers, and it is provided with mechanisms for obtaining the necessary certificate from the following locations:

- A JKS key store, which is the default key store format used by JSSE
- A PKCS#12 file
- A PKCS#11 device, like a hardware security module or cryptographic accelerator

The remainder of this section will describe the process for interacting with and configuring the server to use these types of key stores.

Using the JKS Key Manager Provider

The JKS key store type is the default type of key store used by most JSSE implementations, and in many environments it will be the preferred key store type. In order to configure the server to use this key store type, it will be necessary to first obtain a JKS key store containing a valid certificate. The easiest ways to accomplish this are to either generate a self-signed certificate or to issue a certificate signing request to an existing CA and import the signed certificate.

All of the steps described here will require the use of the `keytool` utility, which is provided with the Java runtime environment. It is typically found in the "`bin`" directory below the root of the Java installation. For more information about using the `keytool` utility, see the official Java documentation.

Step 1: Generate the private key

Regardless of whether you wish to use a self-signed certificate or generate a certificate signing request, you need to first generate a private key, which can be done with the "`-genkey`" option. The arguments that may be of greatest interest for use with this option include:

- `-alias` *{alias}* -- Specifies the nickname that should be used to refer to the certificate in the key store. The default nickname used by OpenDS is "`server-cert`".
- `-keyalg` *{algorithm}* -- Specifies the algorithm that should be used to generate the private key. This should almost always be "`rsa`".
- `-dname` *{subject}* -- Specifies the subject to use for the certificate. The subject typically contains at least a "CN" attribute which is the fully-qualified name of the system on

which it will be installed, an "O" attribute that specifies the name of the organization (or company), and a "C" attribute that specifies the country in which it will be used.

- `-keystore` *{path}* -- Specifies the path to the key store file. The file will be created if it doesn't already exist. The default key store path used by OpenDS is "`config/keystore`".
- `-keypass` *{password}* -- Specifies the password that should be used to protect the private key in the key store. If this is not provided, then you will be interactively prompted for it.
- `-storepass` *{password}* -- Specifies the password that should be used to protect the contents of the key store. If this is not provided, then you will be interactively prompted for it. At the present time, OpenDS expects the password used for the "`-keypass`" and "`-storepass`" options to be the same.
- `-storetype` *{type}* -- Specifies the key store type that should be used. For the JKS key store, the value should always be "`JKS`".

The following example demonstrates the use of this command to create a private key:

```
$ keytool -genkey -alias server-cert -keyalg rsa \
  -dname "CN=dogmatic.central.sun.com,O=Sun Microsystems,C=US" \
  -keystore config/keystore -keypass password \
  -storetype JKS -storepass password
```

Step 2a: Self-sign the certificate

If the certificate is to be self-signed, then that can be accomplished with the"`-selfcert`" option. The most important arguments for use with this option include:

- `-alias` *{alias}* -- Specifies the nickname that should be used to refer to the certificate in the key store. This should be the same as the value used when creating the private key with the "`-genkey`" option.
- `-validity` *{days}* -- Specifies the length of time in days that the certificate should be valid. The default validity is 90 days.
- `-keystore` *{path}* -- Specifies the path to the key store file. The file will be created if it doesn't already exist.
- `-keypass` *{password}* -- Specifies the password that should be used to protect the private key in the key store. If this is not provided, then you will be interactively prompted for it.
- `-storepass` *{password}* -- Specifies the password that should be used to protect the contents of the key store. If this is not provided, then you will be interactively prompted for it.

- `-storetype` *{type}* -- Specifies the key store type that should be used. For the JKS key store, the value should always be "`JKS`".

The following example demonstrates the use of this command to generate a self-signed certificate:

```
$ keytool -selfcert -alias server-cert -validity 1825 \
  -keystore config/keystore -keypass password -storetype JKS \
  -storepass password
```

Step 2b: Use an external certificate authority

If the certificate is to be signed by an external certificate authority, then it is first necessary to generate a certificate signing request (CSR) using the "`-certreq`" option. This option may be used with the following arguments:

- `-alias` *{alias}* -- Specifies the nickname that should be used to refer to the certificate in the key store. This should be the same as the value used when creating the private key with the "`-genkey`" option.
- `-file` *{path}* -- Specifies the path to the file to which the CSR should be written. If this is not provided, then the request will be written to standard output.
- `-keystore` *{path}* -- Specifies the path to the key store file. The file will be created if it doesn't already exist.
- `-keypass` *{password}* -- Specifies the password that should be used to protect the private key in the key store. If this is not provided, then you will be interactively prompted for it.
- `-storepass` *{password}* -- Specifies the password that should be used to protect the contents of the key store. If this is not provided, then you will be interactively prompted for it.
- `-storetype` *{type}* -- Specifies the key store type that should be used. For the JKS key store, the value should always be "`JKS`".

The following example demonstrates the use of this command to obtain a certificate signing request:

```
$ keytool -certreq -alias server-cert -file /tmp/server-cert.csr \
  -keystore config/keystore -keypass password -storetype JKS \
  -storepass password
```

This CSR may then be submitted to a certificate authority to be signed. The method by which to do that, and the method for obtaining the signed certificate, may vary from one certificate authority to another.

Once you have the signed certificate from the CA, then it must be imported into the key store, which can be accomplished with the "`-import`" option. The most useful arguments with this option include:

- `-alias` *{alias}* -- Specifies the nickname that should be used to refer to the certificate in the key store. This should be the same as the value used when creating the private key with the "`-genkey`" option.
- `-file` *{path}* -- Specifies the path to the file containing the signed certificate. It should be in either the DER-encoded binary format or the base64-encoded ASCII format as described in RFC 1421.
- `-keystore` *{path}* -- Specifies the path to the key store file. The file will be created if it doesn't already exist.
- `-storepass` *{password}* -- Specifies the password that should be used to protect the contents of the key store. If this is not provided, then you will be interactively prompted for it.
- `-storetype` *{type}* -- Specifies the key store type that should be used. For the JKS key store, the value should always be "`JKS`".

The following example demonstrates the use of this command to import a signed certificate:

```
$ keytool -import -alias server-cert -file /tmp/server-cert.cert \
  -keystore config/keystore -storetype JKS -storepass password
```

Step 3: Configure the JKS key manager provider

Once you have a JKS key store containing a signed certificate (whether self-signed or signed by an external CA), you may configure the server to use that key store. This may be done by creating a key manager provider entry for that key store. OpenDS provides a default configuration entry that may be used as a template for this. The DN for that entry is "cn=JKS,cn=Key Manager Providers,cn=config". This entry must include the following attributes:

- `ds-cfg-key-manager-provider-class` -- This specifies the fully-qualified name of the Java class that contains the key manager provider implementation. For a JKS key store, this should be `org.opends.server.extensions.FileBasedKeyManagerProvider`.

- `ds-cfg-key-manager-provider-enabled` -- This indicates whether the server should attempt to make this key manager provider available for use by other components. In order for the server to use a certificate in a JKS key store, this must have a value of "`true`".

- `ds-cfg-key-store-type` -- This specifies the key store type. For the JKS key store, it should have a value of "`JKS`".

- `ds-cfg-key-store-file` -- This specifies the path to the key store file. This may be either an absolute path, or a path that is relative to the server root.

Further, it is necessary to provide the server with the PIN needed to access the contents of the key store. This may be specified using one of the following attributes:

- `ds-cfg-key-store-pin` -- This attribute may be used to directly specify the PIN needed to access the key store.

- `ds-cfg-key-store-pin-file` -- This attribute may be used to specify the path to a file containing the PIN needed to access the key store. This may be either an absolute path, or a path that is relative to the server root.

- `ds-cfg-key-store-pin-property` -- This attribute may be used to specify the name of a Java property that holds the PIN needed to access the key store.

- `ds-cfg-key-store-pin-environment-variable` -- This attribute may be used to specify the name of an environment variable that holds the PIN needed to access the key store.

At the present time, the PIN must be provided in clear-text.

Using the PKCS#12 Key Manager Provider

PKCS#12 is a standard format for storing certificate information, including private keys. OpenDS provides the ability to use a PKCS#12 file as a certificate key store, provided that it includes the private key for the certificate.

Because PKCS#12 is a common format for storing certificate information, it may be the case that you already have a certificate in this format, or the CA that you wish to use creates certificates in this form. In some cases, it may also be possible to convert an existing certificate into PKCS#12 format. For example, if you already have a certificate in an NSS certificate database (which is the format used by the Sun Java System Directory Server), then the NSS `pk12util` tool may be used to achieve this. The following is an example of using the `pk12util` tool to export a certificate named "server-cert" contained in the database `../../alias/slapd-config-key3.db` to a PKCS#12 file `/tmp/server-cert.p12`:

```
$ ./pk12util -n server-cert -o /tmp/server-cert.p12 \
  -d ../../alias -P "slapd-config-"
```

If you wish to create a new certificate in PKCS#12 form, you may use the same procedure described in the previous section for obtaining a certificate in a JKS key store. The only difference in the process is that you should use "`-storetype PKCS12`" instead of "`-storetype JKS`" when invoking the `keytool` commands. For example, to create a self-signed certificate in a PKCS#12 file, you could use the following commands:

```
$ keytool -genkey -alias server-cert -keyalg rsa \
  -dname "CN=dogmatic.central.sun.com,O=Sun Microsystems,C=US" \
  -keystore config/keystore.p12 -keypass password \
  -storetype PKCS12 -storepass password


$ keytool -selfcert -alias server-cert -validity 1825 \
  -keystore config/keystore.p12 -keypass password \
  -storetype PKCS12 -storepass password
```

As with JKS, OpenDS provides a template key manager provider configuration entry for use with PKCS#12 certificate files. That is the "cn=PKCS12,cn=Key Manager Providers,cn=config" entry, and it uses the same set of configuration attributes as the configuration entry for the JKS key manager provider. The only differences are that the value of the `ds-cfg-key-store-type` attribute must be "PKCS12" and the `ds-cfg-key-store-file` attribute should refer to the location of the PKCS#12 file rather than a JKS key store.

Using the PKCS#11 Key Manager Provider

PKCS#11 is a standard interface that may be used for interacting with devices capable of holding cryptographic information and/or performing cryptographic functions. The PKCS#11 interface has two common uses of interest for OpenDS:

- Cryptographic accelerators use this interface to allow products to offload their cryptographic processing to an external board (or in some cases, a special module inside the system's CPU, or a framework inside the OS kernel) which may provide better performance for those operations.

- Hardware security modules (HSMs) use this interface to provide a secure repository for storing key information. This significantly reduces the likelihood that sensitive key information will be exposed (and in some cases like devices with FIPS 140-2 certifications, even under extreme conditions) and helps protect the overall integrity of the secure communication mechanisms.

At present, the PKCS#11 support that OpenDS provides has only been tested and verified on systems running Solaris 10 and higher (on SPARC and x86/x64 systems), through the use of the Solaris cryptographic framework. Any device which plugs into the Solaris cryptographic framework should be supported in this manner. This includes the "softtoken" device which is simulated in software and therefore available on all systems supporting the Solaris cryptographic framework regardless of whether they have a hardware device providing PKCS#11 support.

If you do have a third-party PKCS#11 device installed in a Solaris system, then it is likely the case that the Solaris cryptographic framework is already configured to access that device. However, if you will simply be using the software token, or if you are running on a Sun Fire T1000 or T2000 system and wish to take advantage of the cryptographic processor included in the UltraSPARC-T1 CPU, then you will likely need to initialize the PKCS#11 interface. This should first be accomplished by choosing a PIN to use to for the certificate store, which can be done with the command:

```
$ pktool setpin
```

This will prompt you for the current passphrase. If you have not yet used the Solaris cryptographic framework, then the default passphrase will be "changeme". You will then be prompted twice for the new password. Note that this step should be done while logged in as system the user or role that will be used to run OpenDS, since each user may have a different set of certificates.

At this point, it should be possible to use the Java `keytool` utility to interact with the Solaris cryptographic framework through PKCS#11. This will work much in the same way as it does when working with JKS or PKCS#12 key stores, with the following exceptions:

- The value of the "`-keystore`" argument must be "NONE".

- The value of the "`-storetype`" argument must be "PKCS11".

- You should not use the "`-keypass`" argument, nor will the tool prompt you for that password interactively if you do not provide it.

- The value of the "`-storepass`" argument must be the passphrase that you chose when using the "`pktool setpin`" command. Alternately, if you do not provide this argument on the command line, then this is the password that you should enter when prompted.

For example, the following commands will use the PKCS#11 interface to generate a self-signed certificate through the Solaris cryptographic framework:

```
$ keytool -genkey -alias server-cert -keyalg rsa \
  -dname "CN=dogmatic.central.sun.com,O=Sun Microsystems,C=US" \
  -keystore NONE -storetype PKCS11 -storepass password


$ keytool -selfcert -alias server-cert -validity 1825 \
  -keystore NONE -storetype PKCS11 -storepass password
```

Once the certificate has been installed in the PKCS#11 key store, OpenDS must be configured to use that key store. The "cn=PKCS11,cn=Key Manager Providers,cn=config" entry provides a template for this. This entry should be treated much in the same way as the entry for the JKS and PKCS#12 key store providers, with the exception that the `ds-cfg-key-store-file` attribute should not be included. However, a PIN is still required, and may be provided either directly, in a PIN file, through a Java property, or through an environment variable.

Developing Custom Key Manager Providers

If none of the key manager providers offered with OpenDS is sufficient for your needs, you can create your own in order to obtain the key material from virtually anywhere you like. This can be done by creating a class that extends the `org.opends.server.api.KeyManagerProvider` superclass and implementing the methods that it contains. The source for the existing key manager providers may be found in the `org.opends.server.extensions` package and may be useful for reference purposes.

Trust Manager Providers

OpenDS uses trust manager providers to determine whether to trust a certificate that is presented to it. There are two primary use cases for trust manager providers:

- Whenever a client presents its own certificate to the Directory Server during the SSL or StartTLS negotiation process, potentially for use in SASL EXTERNAL authentication.
- Whenever the Directory Server attempts to establish an SSL-based connection to an external system, for example for the purpose of synchronization or for proxied or chained operations.

The role of the trust manager provider is to examine the certificate that is presented and determine whether to continue the negotiation process. It can help improve security whenever SSL or StartTLS is used by thwarting attempts to use forged certificates and foiling man-in-the-middle attacks.

The Basics of Certificate Trust Mechanisms

As mentioned above, trust managers serve an important role in the overall security of the system by ensuring that the peer (the system at the other end of the connection, whether it is an inbound connection from a client or an outbound connection to another server) is who it claims to be. The trust manager has no impact on the strength of the encryption, so only the Directory Server and its peer will be able to understand the communication, and any third-party observer will be unable to decipher the exchange. However, the trust manager is responsible for ensuring that the peer is who they claim to be so that confidential information is not inadvertently exposed to one peer masquerading as another.

There are a number of factors that may come into play when deciding whether a given peer certificate should be trusted. This section will describe some of the most common criteria that is taken into account during this process.

One of the simplest trust mechanisms is the validity period for the certificate. All certificates have a specific window during which they should be considered valid, bounded by "notBefore" and "notAfter" timestamps. If the current time is beyond the "notAfter" timestamp, then the certificate is expired and most trust managers will reject it. Similarly, certificates will also typically be rejected if the current time is before the "notBefore" timestamp. Most often, the "notBefore" timestamp is set to the time that the certificate was signed so it isn't an issue, but there are cases in which a certificate may be issued which is not immediately valid and in those cases, it is important to ensure that the peer is not granted access too early.

Another very important factor in deciding whether to trust a peer certificate is the peer certificate chain. Whenever one system presents its certificate to another, it actually doesn't present just its certificate, but a chain of certificates describing all entities involved in the process. Whenever a trust manager is attempting to determine whether to trust a peer, it will first look in its trust store to determine whether it contains the peer certificate. If that certificate is found, then the peer will be trusted (barring rejection for another reason, like being outside the validity period). If the peer's certificate is not found, then the trust manager will look at the next certificate in the chain, which will be the certificate that was used to sign the peer's certificate (also called the issuer certificate). If the trust store contains the issuer's certificate, then the server will trust that issuer certificate and will also implicitly trust any certificate that it has signed. This process will continue up the certificate chain (looking at the certificate that signed the issuer certificate, and so on) until one of the certificates is found in the trust store or until the root of the chain is reached (in which case, the root certificate will be self-signed and therefore will be its own issuer). If none of the certificates in the peer chain is contained in the trust store, then the peer's certificate will generally be rejected.

This process makes it much easier to manage an environment with a large number of certificates (e.g., one in which there are a large number of servers, or in which many clients may use SASL EXTERNAL authentication) because it is not necessary for the trust store to have each individual peer certificate. The trust store only needs to contain one of the certificates in the peer chain. For example, if all of the certificates that might legitimately be presented to the server were signed by the same issuer, then it will only be necessary to have that issuer's certificate in the trust store in order to implicitly trust any of the peers.

In some environments, there may be other elements taken into account when deciding to trust a peer certificate chain. For example, there may be a certificate revocation list (CRL) which contains a list of all of the certificates which have been revoked and should no longer be considered valid even if they are still within their validity period and were signed by a trusted issuer. This can be useful, for example, if the certificate belonged to an employee that has left the company, or if the private key for the certificate has been compromised. The online certificate status protocol (OCSP, as described in RFC 2560) also provides a similar mechanism, in which the trust manager may ask an OCSP server whether a given certificate is still valid. At the present time, OpenDS does not provide any support for using CRLs or OCSP when attempting to determine whether a peer certificate chain should be trusted.

Using the Blind Trust Manager Provider

The blind trust manager provider is a very simple provider that, as its name implies, blindly trusts any certificate that is presented to it. It doesn't look at the expiration date, who signed the certificate, the subject or alternate names, or any other form of criteria. The blind trust manager provider is provided as a form of convenience for testing purposes only and should never be used in a production server, especially one that is configured to allow SASL EXTERNAL authentication. If a client attempts to use SASL EXTERNAL in order to authenticate to the Directory Server using a certificate and the server blindly accepts any certificate that the client may present, then it is a trivial matter for the user to create their own self-signed certificate that allows them to masquerade as any user in the directory.

OpenDS provides a template entry that may be used to enable the blind trust manager provider. It is contained in the entry "cn=Blind Trust,cn=Trust Manager Providers,cn=config". It is disabled by default, but it may be enabled by changing the value of the "`ds-cfg-trust-manager-provider-enabled`" attribute to "`true`". It does not require any other configuration attributes.

Using the JKS Trust Manager Provider

Just as the JKS key store can be used to provide the key material for a key manager provider, it can also be used to provide information that may be used by trust manager providers. In general, using a JKS file as a trust store is similar to using it as a key store, but because there is no need to access private key information when used as a trust store there is generally no need for a PIN when accessing its contents. The primary factors used by the JKS trust manager provider when attempting to determine wether a given peer certificate chain is to be trusted are whether the peer certificate is within the validity period, and whether any certificate in the chain is contained in the trust store. If the peer certificate is not within the validity period, or none of the certificates in the peer certificate chain are contained in the trust store, then that peer certificate will be rejected by the JKS trust manager.

The `keytool` utility may be used to import certificates into a JKS trust store, through the "`-import`" option. The following arguments may be used with this option:

- `-alias` *{alias}* -- This specifies the nickname to give to the certificate in the trust store. Each certificate should be given a unique nickname, although is primarily for the purpose of managing the certificates in the trust store and has no impact on whether a given certificate will be trusted.

- `-file` *{path}* -- This specifies the path to the file containing the certificate to import. It may be in either DER format, or in the base64-encoded ASCII format described in RFC 1421.

- `-keystore` *{path}* -- This specifies the path to the file that should be used as the JKS trust store. For use as a JKS trust store in OpenDS, this is typically "`config/truststore`".

- `-storetype` *{type}* -- This specifies the format of the trust store file. For the JKS trust manager, this should always have a value of "`JKS`".

- `-storepass` *{password}* -- This specifies the password used to protect the contents of the trust store. If the trust store file does not yet exist, then this will be the password to assign to it that must be used for future interaction with it. If this option is not provided, then it will be interactively requested from the user.

Note that if the trust store does not yet exist, then this command will create it before importing the certificate. The following command provides an example of importing a certificate into a JKS trust store:

```
$ keytool -import -alias server-cert -file /tmp/cert.txt \
  -keystore config/truststore -storetype JKS -storepass password
```

The "cn=JKS,cn=Trust Manager Providers,cn=config" configuration entry provides a template for use with the JKS trust manager provider. The following are the most important configuration attributes in this entry:

- `ds-cfg-trust-manager-provider-enabled` -- This indicates whether the trust manager provider is enabled. It will not be available for use by other server components unless it has a value of "`true`".
- `ds-cfg-trust-store-type` -- This specifies the format of the trust store. For the JKS trust store provider, it should always have a value of "`JKS`".
- `ds-cfg-trust-store-file` -- This specifies the path to the trust store file, which is typically "`config/truststore`", although an alternate file may be used if desirable. It may be either an absolute path or a path that is relative to the server root.

Using the PKCS#12 Trust Manager Provider

The PKCS#12 trust manager provider is primarily useful if you already have the peer and/or issuer certificate(s) that will be used in a PKCS#12 file. If you do not have the certificates in this format, then it is recommended that you use the JKS trust manager provider instead, as the Java `keytool` utility does not currently support importing trusted certificates (i.e., those with just a public key and no private key information) into a PKCS#12 file.

If you do choose to use the PKCS#12 trust manager provider, the entry "cn=PKCS12,cn=Trust Manager Providers,cn=config" may be used to hold the configuration for this provider. The following configuration attributes are of interest when configuring this trust manager provider:

- `ds-cfg-trust-manager-provider-enabled` -- This indicates whether the trust manager provider is enabled. It will not be available for use by other server components unless it has a value of "`true`".
- `ds-cfg-trust-store-type` -- This specifies the format of the trust store. For the PKCS#12 trust store provider, it should always have a value of "`PKCS12`".

- `ds-cfg-trust-store-file` -- This specifies the path to the trust store file, which is typically "`config/truststore.p12`", although an alternate file may be used if desirable. It may be either an absolute path or a path that is relative to the server root.

It may also be necessary to provide a PIN to use to access the contents of the PKCS#12 file. In that case, exactly one of the following configuration attributes must be used to provide that password (at the present time, that password must be provided in clear text):

- `ds-cfg-trust-store-pin` -- This attribute may be used to directly specify the PIN needed to access the trust store.

- `ds-cfg-trust-store-pin-file` -- This attribute may be used to specify the path to a file containing the PIN needed to access the trust store. This may be either an absolute path, or a path that is relative to the server root.

- `ds-cfg-trust-store-pin-property` -- This attribute may be used to specify the name of a Java property that holds the PIN needed to access the trust store.

- `ds-cfg-trust-store-pin-environment-variable` -- This attribute may be used to specify the name of an environment variable that holds the PIN needed to access the trust store.

Developing Custom Trust Manager Providers

If none of the trust manager providers offered with OpenDS is sufficient for your needs, you can create your own by extending the `org.opends.server.api.TrustManagerProvider` abstract class. The trust manager providers mentioned earlier in this section all exist in the `org.opends.server.extensions` package, and their code may be useful in understanding what is involved in creating a trust manager provider.

Certificate Mappers

A certificate mapper is a component which examines a certificate presented by a client and maps it to the user in the directory that should be associated with that certificate. It is primarily used in the context of processing SASL EXTERNAL authentication, in which case the client wishes to authenticate to the server using its SSL certificate rather than a password or some other form of credentials.

This section will describe the certificate mappers provided with OpenDS and how they may be used.

Using the Subject Equals DN Certificate Mapper

The Subject Equals DN certificate mapper is a very simple certificate mapper that expects the subject of the client certificate to be exactly the same as the DN of the corresponding user entry. The configuration for this certificate mapper is held in the "cn=Subject Equals DN,cn=Certificate Mappers,cn=config" entry. While it may be very easy to use this certificate mapper because there are no configuration attributes associated with it, this mapper is also not suitable for many environments because certificate subjects and user DNs are often not the same.

Using the Subject Attribute to User Attribute Certificate Mapper

The Subject Attribute to User Attribute certificate mapper is one that attempts to map a client certificate to a user entry based on a set of attributes that they have in common. In particular, it takes the values of a specified set of attributes from the certificate subject and attempts to locate user entries containing those same values in a corresponding set of attributes.

The configuration for this certificate mapper may be found in the "cn=Subject Attribute to User Attribute,cn=Certificate Mappers,cn=config" entry. There are two configuration attributes that may be used in conjunction with this certificate mapper:

- `ds-cfg-certificate-subject-attribute-mapping` -- This is a multivalued attribute which is used to map attributes from the certificate subject to attributes in user entries. Values for this attribute should consist of the name of the attribute in the certificate subject followed by a colon and the name of the corresponding attribute in the user's entry. For example, the value "`e:mail`" will map the "`e`" attribute from the certificate subject to the "`mail`" attribute in user entries. At least one attribute mapping must be defined.

- `ds-cfg-certificate-user-base-dn` -- This is a multivalued attribute which is used to specify the set of base DNs below which the server should look for matching entries. If this is not present, then the server will search below all public naming contexts.

If there are multiple attribute mappings defined, then the server will combine them with an AND search. For example, if there are two mappings defined, "`cn:cn`" and "`e:mail`", and the server is presented with a certificate having a subject of "E=john.doe@example.com,CN=John Doe,O=Example Corp,C=US", then it will generate a search filter of "`(&(cn=John Doe)(mail=john.doe@example.com))`". Any attribute for which a mapping is defined but is not contained in the certificate subject will not be included in the generated search filter. Note that all attributes that might be used in generated search filters should have corresponding indexes in all backend databases that might be searched by this certificate mapper.

In order for the mapping to be successful, the generated search filter must match exactly one user in the directory (within the scope of the base DNs for the mapper). If there are no users that match the generated criteria, or if there are multiple matching users, then the mapping will fail.

Using the Subject DN to User Attribute Certificate Mapper

The Subject DN to User Attribute certificate mapper is one which attempts to establish a mapping by searching for the subject of the provided certificate in a specified attribute in user entries. In this case, it is necessary to ensure that user entries are populated with the subjects of the certificates associated with those users, but it is possible that this process could be automated in the future with a plugin that automatically identifies any certificates contained in a user entry and adds the subjects of those certificates to a separate attribute.

The configuration for this certificate mapper may be found in the "cn=Subject DN to User Attribute,cn=Certificate Mappers,cn=config" entry. There are two configuration attributes that may be used in conjunction with this certificate mapper:

- `ds-cfg-certificate-subject-attribute-type` -- This is a single-valued attribute whose value is the name of the attribute type that should contain the certificate subject in user entries. This attribute must be defined in the server schema, and it should be indexed for equality in all backends that might be searched.

- `ds-cfg-certificate-user-base-dn` -- This is a multivalued attribute which is used to specify the set of base DNs below which the server should look for matching entries. If this is not present, then the server will search below all public naming contexts.

Note that while there is no standard attribute for holding the subjects of the certificates that a user might hold, OpenDS does define a custom attribute type, `ds-certificate-subject-dn`, that can be used for this purpose. This attribute may be added to user entries along with the `ds-certificate-user` auxiliary object class. It is a multivalued attribute, and if a user has multiple certificates then it should contain the subjects for each of them as separate values.

In order for the mapping to be successful, the certificate mapper must match exactly one user (within the scope of the base DNs for the mapper). If there are no matching entries found, or if there are multiple matches, then the mapping will fail.

Using the Fingerprint Certificate Mapper

The Fingerprint Certificate Mapper is one which attempts to establish a mapping by searching for the MD5 or SHA1 fingerprint of the provided certificate in a specified attribute in user entries. In this case, it is necessary to ensure that user entries are populated with the certificate fingerprints (in standard hexadecimal notation with colons separating the individual bytes, like "07:5A:AB:4B:E1:DD:E3:05:83:C0:FE:5F:A3:E8:1E:EB"). In the future, this process could be automated by a plugin that automatically identifies any certificates contained in user entries and adds the fingerprints of those certificates to the appropriate attribute.

The fingerprint certificate mapper is configured in the "cn=Fingerprint Mapper,cn=Certificate Mappers,cn=config" configuration entry. There are three configuration attributes that may be used in conjunction with this certificate mapper:

- `ds-cfg-certificate-fingerprint-attribute-type` -- This is a single-valued attribute whose value is the name of the attribute type that should contain the certificate fingerprint in user entries. This attribute must be defined in the server schema, and it should be indexed for equality in all backends that might be searched.

- `ds-cfg-certificate-fingerprint-algorithm` -- This specifies which digest algorithm should be used to calculate certificate fingerprints. The value should be either "`MD5`" or "`SHA1`".

- `ds-cfg-certificate-user-base-dn` -- This is a multivalued attribute which is used to specify the set of base DNs below which the server should look for matching entries. If this is not present, then the server will search below all public naming contexts.

Note that while there is no standard attribute for holding certificate fingerprints, OpenDS does define a custom attribute type, `ds-certificate-fingerprint`, that can be used for this purpose. This attribute may be added to user entries along with the `ds-certificate-user` auxiliary object class. It is a multivalued attribute, and if a user has multiple certificates then it should contain the fingerprints for each of them as separate values.

In order for the mapping to be successful, the certificate mapper must match exactly one user (within the scope of the base DNs for the mapper). If there are no matching entries found, or if there are multiple matching entries, then the mapping will fail.

Developing Custom Certificate Mappers

If none of the certificate mappers provided with OpenDS are suitable for your environment, then you may develop your own by extending the `org.opends.server.api.CertificateMapper` class. All of the certificate mappers described in this section are implemented in the `org.opends.server.extensions` package and may be used for reference purposes.

Configuring SSL and StartTLS for LDAP and JMX

Once the Directory Server has been configured with at least one key manager provider and at least one trust manager provider enabled, you can enable SSL and/or StartTLS for the connection handlers.

Configuring the LDAP and LDAPS Connection Handlers

The LDAP connection handler is responsible for managing all communication with clients using the LDAP protocol. By default, it does not use any form of security for protecting that communication, but it can be configured to use SSL, or also to allow the use of the StartTLS extended operation.

OpenDS provides two template configuration entries that may be used for this purpose. The "cn=LDAP Connection Handler,cn=Connection Handlers,cn=config" entry is enabled by default and is used to perform unencrypted LDAP communication, but can be easily configured to also support the StartTLS extended operation. The "cn=LDAPS Connection Handler,cn=Connection Handlers,cn=config" entry is disabled but the default configuration is set up for SSL-based communication.

The following configuration attributes are of particular interest for the LDAP connection handler:

- `ds-cfg-connection-handler-enabled` -- This indicates whether the connection handler is enabled.
- `ds-cfg-listen-port` -- This specifies the port number to use when communicating with the Directory Server through this connection handler. The standard port to use unencrypted LDAP communication (or LDAP using StartTLS) is 389, and the standard port for SSL-encrypted LDAP is 636, although it may be desirable or necessary to change this in some environments (e.g., if the standard port is already in use, or if you are running on a UNIX system as a user without sufficient privileges to bind to a port below 1024).
- `ds-cfg-ssl-client-auth-policy` -- This specifies how the connection handler should behave with regard to requesting a client certificate during the SSL or StartTLS negotiation process. If the value is "`optional`", then the server will request that the client present its own certificate but will not still accept the connection even if the client

does not provide one. If the value is "`required`", then the server will request that the client present its own certificate and will reject any connection in which the client does not do so. If the value is "`disabled`", then the server will not ask the client to present its own certificate.

- `ds-cfg-ssl-cert-nickname` -- This specifies the nickname of the certificate that the server should present to clients during SSL or StartTLS negotiation. This is primarily useful for the case in which there are multiple certificates in the key store and you wish to explicitly specify which should be used for that listener instance.

- `ds-cfg-key-manager-provider-dn` -- This specifies the DN of the configuration entry for the key manager provider that should be used by the connection handler to obtain the key material for the SSL or StartTLS negotiation.

- `ds-cfg-trust-manager-provider-dn` -- This specifies the DN of the configuration entry for the trust manager provider that should be used by the connection handler to decide whether to trust client certificates presented to it.

- `ds-cfg-use-ssl` -- This indicates whether the connection handler should use SSL to encrypt communication with the clients. If SSL is enabled, then non-SSL communication will not be available for that connection handler instance.

- `ds-cfg-allow-start-tls` -- This indicates whether the connection handler should allow the use of the StartTLS extended operation. This may not be used if SSL is enabled.

In order to enable StartTLS support in the "cn=LDAP Connection Handler,cn=Connection Handlers,cn=config" entry, it is first necessary to provide appropriate values for the `ds-cfg-key-manager-provider-dn` and `ds-cfg-trust-manager-provider-dn` attributes, and then the `ds-cfg-allow-start-tls` attribute may be set to "true".

In order to enable the "cn=LDAPS Connection Handler,cn=Connection Handlers,cn=config" connection handler for SSL-based communication, it is first necessary to ensure that the configured key manager provider and trust manager provider values are correct, and then the `ds-cfg-connection-handler-enabled` attribute may be set to "`true`".

Enabling SSL in the JMX Connection Handler

The JMX connection handler may be used to communicate with clients using the JMX (Java Management eXtensions) protocol. This protocol does not support the use of StartTLS to allow both encrypted and unencrypted communication over the same port, but it may be configured to accept only unencrypted JMX or only SSL-encrypted JMX communication.

The "cn=JMX Connection Handler,cn=Connection Handlers,cn=config" entry provides the server's default configuration for communicating over JMX. In order to enable SSL for this connection handler, the following configuration attributes will be of interest:

- `ds-cfg-key-manager-provider-dn` -- This specifies the DN of the configuration entry for the key manager provider that should be used to obtain the key material for the SSL negotiation.
- `ds-cfg-ssl-cert-nickname` -- This specifies the nickname (or alias) of the certificate that should be presented to clients.
- `ds-cfg-use-ssl` -- This indicates whether the connection handler should use SSL to communicate with clients.

Configuring SASL EXTERNAL Authentication

The EXTERNAL SASL mechanism is used to allow a client to authenticate itself to the Directory Server using information provided outside of what is strictly considered LDAP communication. In OpenDS, this is currently restricted to authentication using a client certificate presented to the server during SSL or StartTLS negotiation, and it is at present only available for LDAP communication.

Configuring the LDAP Connection Handler to Allow SASL EXTERNAL

In order for the Directory Server to be able to map the client certificate to a user entry, it will first be necessary to ensure that the connection handler is properly configured to handle client certificates. This should be done through the following two configuration attributes:

- `ds-cfg-ssl-client-auth-policy` -- This specifies whether the Directory Server will prompt the client to present its own certificate during the SSL or StartTLS negotiation process. In order to support SASL EXTERNAL authentication, the value must be either "`optional`" or "`required`". If the value is "`disabled`", then clients will not be prompted to provide a certificate and therefore it will not be available for authentication.

- `ds-cfg-trust-manager-provider-dn` -- This specifies the DN of the trust manager provider that will be used to determine whether the Directory Server trusts the validity of the client certificate. If the Directory Server does not trust the client certificate, then the SSL or StartTLS negotiation will fail and it will not be possible for the client to request SASL EXTERNAL authentication. On the other hand, if the Directory Server trusts illegitimate client certificates, then it may be possible for malicious users to forge certificates and impersonate any user in the directory. In most cases, it is recommended that the JKS or PKCS12 trust manager provider be used and the corresponding trust store loaded only with the issuer certificates that will be used to sign client certificates.

Configuring the EXTERNAL SASL Mechanism Handler

The actual logic of processing SASL EXTERNAL bind requests is handled by the SASL mechanism handler defined in the "cn=EXTERNAL,cn=SASL Mechanisms,cn=config" entry. This entry provides the following configuration attributes:

- `ds-cfg-sasl-mechanism-handler-class` -- This specifies the fully-qualified name of the Java class that provides the logic for the SASL mechanism handler. For the EXTERNAL mechanism, this should always be `org.opends.server.extensions.ExternalSASLMechanismHandler`.

- `ds-cfg-sasl-mechanism-handler-enabled` -- This indicates whether the EXTERNAL SASL mechanism will be enabled for use in the server. If you do not wish to allow clients to use SASL EXTERNAL authentication, then you can change its value to "`false`".

- `ds-cfg-certificate-mapper-dn` -- This specifies the DN of the configuration entry for the certificate mapper that should be used to map client certificates to user entries.

- `ds-cfg-client-certificate-validation-policy` -- This specifies whether the server should attempt to locate the client certificate in the user's entry after establishing a mapping. If the value is "`always`" then the authentication will succeed only if the mapped user's entry contains the certificate presented by the client. If the value is "`ifpresent`" (which is the default value) and the user's entry contains one or more certificates, then the authentication will succeed only if one of those certificates matches the one presented by the client. If the value is "`ifpresent`" and the user's entry does not contain any certificates, then the authentication will still succeed based on the fact that it would have been accepted by the trust manager and mapped by the certificate mapper. If the value is "`never`" then the server will not attempt to match the certificate to a value in the user's entry even if that entry contains one or more certificates.

- `ds-cfg-certificate-attribute` -- This specifies the name of the attribute that should hold user certificates to be examined if the ds-cfg-client-certificate-validation-policy attribute has a value of either "`always`" or "`ifpresent`".

Testing SSL and StartTLS with ldapsearch

The `ldapsearch` utility included with OpenDS can be a very useful tool for testing that the server is properly configured to support SSL and StartTLS. It includes a number of options that are well-suited for testing in a number of different scenarios. This section will describe how it can be used to test SSL and StartTLS communication, and how to perform SASL EXTERNAL authentication.

Note that the same process may also be used with many of the other client tools provided with OpenDS, including `ldapmodify`, `ldapcompare`, and `ldapdelete`.

Applicable Command Line Arguments

The following command-line arguments are of particular interest when using the `ldapsearch` tool to communicate via SSL or StartTLS:

- "`-h` *{address}*" or "`--host` *{address}*" -- This is used to specify the address of the Directory Server system to which you wish to connect. If no value is specified, it will use the IPv4 loopback address (127.0.0.1).

- "`-p` *{port}*" or "`--port` *{port}*" -- This is used to specify the port number on which the Directory Server is listening for connections. If no value is specified, it will use the standard unencrypted LDAP port (389).

- "`-Z`" or "`--useSSL`" -- This is used to indicate that the client should use SSL to secure the communication with the Directory Server. If this option is used, then the value specified for the port argument must be one on which the server is listening for SSL-based connections (the standard LDAPS port is 636).

- "`-q`" or "`--startTLS`" -- This is used to indicate that the client should use the StartTLS extended operation to secure the communication with the Directory Server. If this option is used, then the value specified for the port argument must be the one on which the server is listening for clear-text LDAP connections (although the port argument is not required if the server is listening on the default LDAP port of 389).

- "`-r`" or "`--useSASLExternal`" -- This is used to indicate that the client should use SASL EXTERNAL to authenticate to the Directory Server. If this option is used, then you must also provide a key store path.

- "`-X`" or "`--trustAll`" -- This is used to indicate that the client should blindly trust any certificate that the Directory Server may present. It should not be used in conjunction with the argument used to specify the trust store path.

- "`-K` *{path}*" or "`--keyStorePath` *{path}*" -- This is used to specify the path to the key store that should be used if the client is to present a certificate to the Directory Server (e.g., when using SASL EXTERNAL authentication). This should be the path to a JKS key store.

- "`-W` *{password}*" or "`--keyStorePassword` *{password}*" -- This is used to specify the PIN required to access the contents of the key store. This should not be used in conjunction with the key store password file argument.

- "`--keyStorePasswordFile` *{path}*" -- This is used to specify the path to a file containing the PIN required to access the contents of the key store. This should not be used in conjunction with the key store password argument.

- "`-N` *{nickname}*" or "`--certNickname` *{nickname}*" -- This is used to specify the nickname, or alias, of the certificate that the client should present to the Directory Server. The key store path argument must also be provided. If no nickname is given, then the client will pick the first acceptable client certificate that it finds in the key store.

- "`-P` *{path}*" or "`--trustStorePath` *{path}*" -- This is used to specify the path to the JKS trust store file that the client should use when determining whether to trust the certificate presented by the Directory Server. If this argument is not given and the trust all option is not given, then any certificate presented to the client will be displayed and the user will be prompted about whether to trust it.

- "`--trustStorePassword` *{password}*" -- This is used to specify the password needed to access the trust store contents. In most cases, no trust store password will be required. This should not be used in conjunction with the trust store password file option.

- "`--trustStorePasswordFile` *{path}*" -- This is used to specify the path to a file containing the password needed to access the trust store contents. In most cases, no trust store password will be required. This should not be used in conjunction with the trust store password option.

- "`-E`" or "`--reportAuthzID`" -- This is used to indicate that the Directory Server should include the authorization identity of the authenticated user in the bind response. This is useful when performing SASL authentication to determine the user to which the client certificate (or other form of SASL credentials if a mechanism other than EXTERNAL was used) was mapped.

Testing SSL

The following demonstrates the use of `ldapsearch` to communicate with a Directory Server using LDAP over SSL:

```
$ bin/ldapsearch --host directory.example.com --port 636 \
  --useSSL --baseDN "" --searchScope base "(objectClass=*)"
```

In this case, no trust store was specified, and the trust all option was also not given. Therefore, when the server presents its certificate to the client, then the user will be prompted about whether that certificate should be trusted. The entire sequence may look something like:

```
$ bin/ldapsearch --host directory.example.com --port 636 \
  --useSSL --baseDN "" --searchScope base "(objectClass=*)"
The server is using the following certificate:
    Subject DN:  CN=directory.example.com, O=Example Corp, C=US
    Issuer DN:  CN=directory.example.com, O=Example Corp, C=US
    Validity:  Fri Mar 02 16:48:17 CST 2007 through Thu May 31
17:48:17 CDT 2007
Do you wish to trust this certificate and continue connecting to
the server?
Please enter "yes" or "no":  yes
dn:
objectClass: ds-rootDSE
objectClass: top
```

If the client simply wants to always trust any certificate that the server presents without being prompted, then the trust all argument may be provided, like:

```
$ bin/ldapsearch --host directory.example.com --port 636 \
  --useSSL --trustAll --baseDN "" --searchScope base \
  "(objectClass=*)"
```

If the client has a trust store and wants to use that to determine whether to trust the server certificate, then the trust store path argument may also be given, like:

```
$ bin/ldapsearch --host directory.example.com --port 636 \
  --useSSL --trustStorePath client.truststore --baseDN "" \
  --searchScope base "(objectClass=*)"
```

Testing StartTLS

The process for using StartTLS with the `ldapsearch` utility is almost identical to the process for using SSL. The only differences are that you should use the port on which the server is listening for unencrypted LDAP requests, and you should indicate that StartTLS should be used instead of

SSL (that is, use "`--startTLS`" instead of "`--useSSL`"). The following example is the equivalent of the first example given for using SSL with `ldapsearch` except that it uses StartTLS to secure the communication:

```
$ bin/ldapsearch --host directory.example.com --port 389 \
  --startTLS --baseDN "" --searchScope base "(objectClass=*)"
```

This applies for all of the other examples given -- simply change the port number from the LDAPS port to the LDAP port, and replace the "`--useSSL`" option with "`--startTLS`".

Testing SASL EXTERNAL Authentication

SASL EXTERNAL authentication may be used in conjunction with either SSL or StartTLS. The primary differences are that it will be necessary to provide a key store that contains the client certificate, the PIN required to access the contents of that key store, and a flag indicating that the client should use SASL EXTERNAL authentication. The following example demonstrates sample usage for such a command:

```
$ bin/ldapsearch --host directory.example.com --port 636 \
  --useSSL --keyStorePath /path/to/client.keystore \
  --keyStorePasswordFile /path/to/client.keystore.pin \
  --useSASLExternal --baseDN "" --searchScope base \
  "(objectClass=*)"
```

When using SASL EXTERNAL authentication, it is also often useful to ask the server to return the authorization identity to ensure that the authentication is being performed as the correct user. The following demonstrates an example of this process (note the value reported on the line beginning with the "`#`" character):

```
$ bin/ldapsearch --host directory.example.com --port 636 \
  --useSSL --keyStorePath /path/to/client.keystore \
  --keyStorePasswordFile /path/to/client.keystore.pin \
  --useSASLExternal --reportAuthzID --baseDN "" \
  --searchScope base "(objectClass=*)"
# Bound with authorization ID dn:uid=test.user,dc=example,dc=com
dn:
objectClass: ds-rootDSE
objectClass: top
```

Common Development and Distribution License, Version 1.0

Unless otherwise noted, all components of the OpenDS Directory Server, including all source, configuration, and documentation, are released under the Common Development and Distribution License (CDDL), Version 1.0. The full text of that license is as follows: