Security

This guide helps you to reduce risk and mitigate threats to directory service security.

Threats

Understand security threats.

Authentication

Enforce secure authentication.

Cryptographic Keys

Manage certificates and keys.

Connections

Secure network connections.

Passwords

Store and manage passwords.

Data Encryption

Protect data on disk.

IMPORTANT

IMPORTANT

A guide to securing directory services can go wrong for many reasons, including at least the following:

- The author fails to understand or to properly explain the subject.
- The reader fails to understand or to act on what is written.
- Bugs exist in the directory's security-related features.

The authors of this guide aim to understand directory security features and issues before attempting to explain how to manage them.

The reader would do well to gain grounding in securing services and systems, and in applying and designing processes that prevent or mitigate threats, before reading the guide with a critical eye, and a grain of salt. This is not a guide to getting started with security.

ForgeRock® Identity Platform serves as the basis for our simple and comprehensive Identity and Access Management solution. We help our customers deepen their relationships with their customers, and improve the productivity and connectivity of their employees and partners. For more information about ForgeRock and about the platform, see https://www.forgerock.com.

The ForgeRock® Common REST API works across the platform to provide common ways to access web resources and collections of resources.

Threats

Review common threats to directory services, which you can mitigate by following the instructions in this guide.

Insecure Client Applications

Directory services make a good, central, distributed store for identity data and credentials.

Standard access protocols, and delegated access and data management mean you might not know which client applications use your services. Some client applications may behave insecurely:

- Prevent insecure connections.

  Require that applications connect with LDAPS and HTTPS, and restrict the protocol versions and cipher suites for negotiating secure connections to those with no known vulnerabilities. For details, see Secure Connections.

- Accept only secure management of sensitive data.

  Nothing in LDAPv3 or HTTP prevents a client application from sending credentials and other sensitive account data insecurely. You can configure the directory to discourage the practice, however.
- Encourage secure authentication.

  For details, see Authentication Mechanisms.
- Encourage best practices for client applications, such as scrubbing input to avoid injection vulnerabilities.

  For details, see Client Best Practices.

Client Applications

Client applications can misuse directory services. They may be poorly built or incorrectly configured, and waste server resources. If your organization owns the client, help the owner fix the problem.

Misuse can be intentional, as in a denial-of-service attack. Protect the directory service to limit attacks.

Unreasonable requests from client applications include the following:

- Unindexed searches that would require the directory to evaluate all entries.

  Unindexed searches are not allowed by default for normal accounts, adjustable with the `unindexed-search` privilege.

  The access log records attempts to perform unindexed searches.
- Excessive use of overly broad persistent searches, particularly by clients that do not process the results quickly.

  Review requests before setting ACIs to grant access to use the persistent search control. Alternatively, let client applications read the external change log.
- Extremely large requests, for example, to update directory entries with large values.

  By default, requests larger than 5 MB are refused. The setting is per connection handler, `max-request-size` .
- Requests that make excessive demands on server resources.

  Set resource limits. See Resource Limits.
- Requests to read entire large group entries only to check membership.

  Encourage client applications to read the `isMemberOf` attribute on the account instead.

Poor Password Management

Despite efforts to improve how people manage passwords, users have more passwords than ever before, and many use weak passwords. You can use identity and access management services to avoid password proliferation, and you can ensure the safety of passwords that you cannot eliminate.

As a central source of authentication credentials, directory services provide excellent password management capabilities. DS servers have flexible password policy settings, and a wide range of safe password storage options. Be sure that the passwords stored in your directory service are appropriately strong and securely stored.

For details, see Passwords.

Manage passwords for server administration securely as well. Passwords supplied to directory server tools can be provided in files, interactively, through environment variables, or as system property values. Choose the approach that is most appropriate and secure for your deployment.

Make sure that directory administrators manage their passwords well. To avoid password proliferation for directory administrators, consider assigning administration privileges and granting access to existing accounts for delegated administrative operations. For details, see Administrative Roles and Access Control.

Misconfiguration

With the power to administer directory services comes the responsibility to make sure the configuration is correct. Misconfiguration can arise from bad or mistaken configuration decisions, and from poor change management.

Bad configuration decisions can result in problems such as the following:

- A particular feature stops working.

  Depending on the configuration applied, features can stop working in obvious or subtle ways.

  For example, suppose a configuration change prevents the server from making LDAPS connections. Many applications will no longer be able to connect, and so the problem will be detected immediately. If the configuration change simply allows insecure TLS protocol versions or cipher suites for LDAPS connections, some applications will negotiate insecure TLS, but they will appear to continue to work properly.
- Access policy is not correctly enforced.

  Incorrect parameters for secure connections and incorrect ACIs can lead to overly permissive access to directory data, and potentially to a security breach.

- The server fails to restart.

  Although failure to start a server is not directly a threat to security, it can affect dependent identity and access management systems.

  Generally a result of editing the server configuration LDIF incorrectly, this problem can usually be avoided by using configuration tools. A server that started correctly saves a copy of the configuration in the `var/config.ldif.startok` file. You can compare this with the `config/config.ldif` file if the server fails to restart.

To guard against bad configuration decisions, implement good change management:

- For all enabled features, document why they are enabled and what your configuration choices mean.

  This implies review of configuration settings, including default settings that you accept.
- Validate configuration decisions with thorough testing.

  For details, see Tests.
- Maintain a record of your configurations, and the changes applied.

  For example, use a filtered directory audit log. Use version control software for any configuration scripts and to record changes to configuration files.

  Make sure you also maintain a record of external changes on the system, such as changes to operating system configuration, and updates to software such as the JVM that introduce security changes.
- Strongly encourage owners of applications that change ACIs, collective attributes, and similar settings in directory data to also follow good change management practices.

Unauthorized Access

Data theft can occur when access policies are too permissive, and when the credentials to gain access are too easily cracked. It can also occur when the data is not protected, when administrative roles are too permissive, and when administrative credentials are poorly managed.

To protect against unauthorized access over the network, see the suggestions in Insecure Client Applications, Poor Password Management, and Access Control.

To protect against unauthorized access by administrators, see the suggestions in Data Encryption, and Administrative Roles.

Poor Risk Management

Threats can arise when plans fail to account for outside risks.

Develop appropriate answers to at least the following questions:

- What happens when a server or an entire data center becomes unavailable?
- How do you remedy a serious security issue in the service, either in the directory service software or the underlying systems?
- How do you validate mitigation plans and remedial actions?
- How do client applications work when the directory service is offline?

  If client applications require always-on directory services, how do your operations ensure high availability, even when a server or data center goes offline?

For a critical directory service, you must test both normal, expected operation, and disaster recovery.

Security Features

This short introduction provides an overview of DS security features.

Encryption and Digests

When DS servers must store sensitive data, and file permissions alone are not sufficient, they use encryption and digests:

- *Encryption* turns source data into a reversible code. Good design makes it extremely hard to recover the data from the code without the decryption key.

  Encryption uses keys and cryptographic algorithms to convert source data into encrypted codes and back again. Given the decryption key and the details of the algorithm, converting an encrypted code back to source data is straightforward, though it can be computationally intensive.

  DS software does not implement its own versions of all encryption algorithms. Instead, it often relies on cryptographic algorithms provided by the underlying JVM. DS servers do manage access to encryption keys, however. An important part of server configuration concerns key management.

  DS servers use encryption to protect data and backup files on disk. They can encrypt password values when you configure a reversible storage scheme. Another important use of encryption is to make network connections secure.
- A *digest* (also called a hash) is a non-reversible code generated from source data using a one-way *hash function*. (A hash function is one that converts input of arbitrary size into output of fixed size.) Good one-way hash design design makes it

effectively impossible to retrieve the source data even if you have access to the hash function.

The hash function makes it simple to test whether a given value matches the original. Convert the value into a digest with the same hash function. If the new digest matches the original digest, then the values are also identical.

DS server use digests to store hashed passwords, making the original passwords extremely hard to recover. They also use digests for authentication and signing.

In DS software, two types of encryption keys are used:

1. *Symmetric keys* , also called secret keys, because they must be kept secret.

   A single symmetric key is used for both encryption and decryption.
2. *Asymmetric key pairs* , consisting of a sharable public key and a secret private key.

   Either key can be used for encryption and the other for decryption.

Connection Management

DS servers manage incoming client connections using *connection handlers*. Each connection handler is responsible for accepting client connections, reading requests, and sending responses. Connection handlers are specific to the protocol and port used. For example, a server uses one connection handler for LDAPS and another for HTTPS.

The connection handler configuration includes optional security settings. When you configure a handler, specify a *key manager provider* and a *trust manager provider*:

- The key manager provider retrieves the server certificate when negotiating a secure connection.

  A key manager provider is backed by a *keystore* , which stores the server's key pairs.
- The trust manager provider retrieves trusted certificates, such as CA certificates, to verify trust for a certificate presented by a peer when setting up a secure connection.

  A trust manager provider is backed by a keystore that contains trusted certificates, referred to as a *truststore* when used in this way.

DS servers support file-based keystores and PKCS#11 security tokens, such as HSMs.

Always use secure connections when allowing access to sensitive information. For details, see Secure Connections.

Cryptographic Key Management

DS servers use cryptographic mechanisms for more than setting up secure connections:

- Encrypted backup files must be decrypted when restored.
- Passwords can be protected by encryption rather than hashing, although this is not recommended.
- Database backends can be encrypted for data confidentiality and integrity.

For all operations where data is stored in encrypted form, all replicas must be trusted to access the secret key.

Trust between servers depends on a public key infrastructure. This type of infrastructure is explained in more detail in Public Key Infrastructure.

Replication requires trust between the servers. Trust enables servers to secure network connections, and to share symmetric keys securely. Servers encrypt symmetric keys with a shared master key, and store them in replicated data. When a server needs the symmetric key for decryption or further encryption, it decrypts its copy with the master key.

The component that provides a common interface for cryptographic functions is called the DS Crypto Manager.

You can configure the following Crypto Manager features with the `dsconfig` command:

- Protection for symmetric keys.
- The alias of the shared master key to use for protecting secret keys.
- Cipher key lengths, algorithms, and other advanced properties.

Authentication

Authentication is the act of confirming the identity of a principal, such as a user, application, or device. The main reason for authentication is that authorization decisions are based on the identity of the principal.

Servers should require authentication before allowing access to any information that is not public.

Authentication mechanisms depend on the access protocol. HTTP has a number of mechanisms, such as HTTP Basic. LDAP has other mechanisms, such as anonymous bind and external SASL. For details on supported mechanisms, see Authentication Mechanisms.

Authorization

Authorization is the act of determining whether to grant a principal access to a resource.

DS servers authorize access based on these mechanisms:

- *Access control instructions* (ACI)

  Access control instructions provide fine-grained control over LDAP operations permitted for a principal.

  ACIs can be replicated.
- Administrative *privileges*

  Privileges control access to administrative tasks, such as backup and restore operations, making changes to the configuration, and other tasks.

  Privileges can be replicated.
- *Global access control policies*

  Global access control policies provide coarse-grained access control for proxy servers, where the lack of local access to directory data makes ACIs a poor fit.

For details about ACIs and global access control policies with proxy servers, see Access Control.

For details about privileges, see Administrative Roles.

Monitoring and Logging

You must monitor deployed services for evidence of threats and other problems. Interfaces for monitoring include the following:

- Remote monitoring facilities that clients applications can access over the network.

  These include JMX and SNMP connection handlers, and the monitor backend that is accessible over LDAP and HTTP.
- Alerts to notify administrators of significant problems or notable events over JMX or by email.
- Account status notifications to send users alerts by email, or to log error messages when an account state changes.
- Logging facilities, including local log files for access, debugging, entry change auditing, and errors. ForgeRock Common Audit event handlers support local logging and sending access event messages to a variety of remote logging and reporting systems.

For details, see Monitoring.

Operating Systems

When you deploy Directory Services software, secure the host operating system. The suggestions that follow are not exhaustive. Familiarize yourself with the specific recommendations for the host operating systems you use.

System Updates

Over the lifetime of a directory services deployment, the operating system might be subject to vulnerabilities. Some vulnerabilities require system upgrades, whereas others require only configuration changes. All updates require proactive planning and careful testing.

For the operating systems used in production, put a plan in place for avoiding and resolving security issues. The plan should answer the following questions:

- How does your organization become aware of system security issues early?

  This could involve following bug reports, mailing lists, forums, and other sources of information.
- How do you test security fixes, including configuration changes, patches, service packs, and system updates?

  Validate the changes first in development, then in one or more test environments, then in production in the same way you would validate other changes to the deployment.
- How do you roll out solutions for security issues?

  In some cases, fixes might involve both changes to the service, and specific actions by those who use the service.
- What must you communicate about security issues?
- How must you respond to security issues?

Software providers often do not communicate what they know about a vulnerability until they have a way to mitigate or fix the problem. Once they do communicate about security issues, the information is likely to become public knowledge quickly. Make sure that you can expedite resolution of security issues.

To resolve security issues quickly, make sure you are ready to validate any changes that must be made. When you validate a change, check the fix resolves the security issue. Validate that the system and DS software continue to function as expected in all the ways they are used.

Disable Unused Features

By default, operating systems include many features, accounts, and services that DS software does not require. Each optional feature, account, and service on the system brings a risk of additional vulnerabilities. To reduce the surface of attack, enable only required features, system accounts, and services. Disable or remove those that are not needed for the deployment.

The features needed to run and manage DS software securely include the following:

- A Java runtime environment, required to run DS software.
- Software to secure access to service management tools; in particular, when administrators access the system remotely.
- Software to secure access for remote transfer of software updates, backup files, and log files.
- Software to manage system-level authentication, authorization, and accounts.
- Firewall software, intrusion-detection/intrusion-prevention software.
- Software to allow auditing access to the system.
- System update software to allow updates that you have validated previously.
- If required for the deployment, system access management software such as SELinux.
- If the DS server sends email alerts locally, mail services.
- If you use SNMP with DS servers, an SNMP agent.
- Any other software that is clearly indispensable to the deployment.

Consider the minimal installation options for your operating system, and the options to turn off features.

Consider configuration options for system hardening to further limit access even to required services.

For each account used to run a necessary service, limit the access granted to the account to what is required. This reduces the risk that a vulnerability in access to one account affects multiple services across the system.

Make sure that you validate the operating system behavior every time you deploy new or changed software. When preparing the deployment and when testing changes, maintain a full operating system with DS software that is not used for any publicly available services, but only for troubleshooting problems that might stem from the system being *too* minimally configured.

Administrative Access

Limit access to the system by protecting network ports and reducing access granted to administrative accounts. This further reduces the attack surface and reduces the advantage to be gained from exploiting a vulnerability.

DS servers listen for protocols listed in the following table. When protecting network ports, you must open some to remote client applications, and for replication. If administrators can connect over SSH, you can restrict access to the administrative port to the localhost only.

Protocols	Ports(1)	Active by Default?	Description
LDAP	`389`, `1389`	No	Port for insecure LDAP requests and for StartTLS requests to enable a secure connection. The reserved LDAP port number is `389`. If LDAP is used, leave this port open to client applications.
LDAPS	`636`, `1636`	No	Port for secure LDAPS requests. The standard LDAPS port number is `636`. If LDAPS is used, leave this port open to client applications.
HTTP, HTTPS	`80` / `8080`, `443` / `8443`	No	Port for HTTP client requests, such as RESTful API calls. The standard HTTP port number is `80`. The standard HTTPS port number is `443`. If HTTP or HTTPS is used, leave this port open to client applications. For production deployments, use HTTPS instead of HTTP.
Administration	`4444`	Yes	Port for administrative requests, such as requests from the **`dsconfig`** command. Initial setup secures access to this port.
Replication	`8989`	No	Port for replication requests, using the DS-specific replication protocol. If replication is used, leave this port open to other replicas. For production deployments, secure access to this port.

Protocols	Ports[1]	Active by Default?	Description
JMX	`1689`	No	Port for Java Management Extensions requests (`1689`), and JMX RMI requests. The default setting for the JMX RMI port is `0` , meaning the service chooses a port of its own. This can be configured using the JMX connection handler `rmi-port` setting If used in production deployments, secure access to this port.
SNMP	`161` , `162`	No	Reserved ports are `161` for regular SNMP requests, and `162` for traps. If used in production deployments, secure access to these ports.

(1) You choose actual port numbers at setup time.

When setting up system accounts to manage directory services:

- Set up a separate DS system account for the server.
- Prevent other system accounts from accessing DS files.
- Configure the system to prevent users from logging in as the DS system account user.
- Configure the system to restrict the DS account to server management operations.

System Audits

DS logs provide a record of directory service events, but they do not record system-level events. Use the auditing features of the host operating system to record access that is not recorded in DS logs.

System audit logs make it possible to uncover system-level security policy violations, such as unauthorized access to DS files. Such violations are not recorded in DS logs or monitoring information.

Also consider how to prevent or at least detect tampering. A malicious user violating security policy is likely to try to remove evidence of how security was compromised.

Java Updates

Security updates are regularly released for the Java runtime environment. Plan to deploy Java security updates to systems where you run DS software:

1. If the DS server relies on any CA certificates that were added to the Java runtime environment truststore, `$JAVA_HOME/Home/lib/security/cacerts`, for the previous update, add them to the truststore for the update Java runtime environment.
2. Edit the `default.java-home` setting in the `config/java.properties` file to use the new path.

   The setting should reflect the updated Java home:

   ```
   default.java-home=/path/to/updated/java/jre
   ```

   When you set up DS servers, the path to the Java runtime environment is saved in the configuration. The server continues to use that Java version until you change the configuration.
3. Restart the DS server to use the updated Java runtime environment:

   ```
   $ stop-ds --restart
   ```

Gateway Security

The DS DSML and DS REST to LDAP gateways run as web applications in containers like Apache Tomcat. Security settings depend on the container, and the gateway configuration files.

Container Security Settings

Security settings are covered in the documentation for supported web application containers. The documentation to use depends on the web application container.

For example, the Apache Tomcat 9 documentation includes the following:

- For instructions on setting up HTTPS, see SSL/TLS Configuration HOW-TO.
- For other security-related settings, see Security Considerations.

DSML Settings

Make sure the web application container protects traffic to the gateway with HTTPS.

Review the following settings DSML gateway settings:

`ldap.port`
: Use an LDAP port that supports StartTLS or LDAPS.

  Using StartTLS or LDAPS is particularly important if the gateway ever sends credentials over LDAP.

`ldap.usessl`
: If `ldap.usestarttls` is not used, set this to `true`.

`ldap.usestarttls`
: If `ldap.usessl` is not used, set this to `true`.

`ldap.trustall`
: Make sure this is set to `false`.

`ldap.truststore.path`
: Set this to a truststore with the appropriate certificate(s) for remote LDAP servers.

`ldap.truststore.password`
: If `ldap.truststore.path` is set, and the truststore requires a password, set this appropriately.

REST to LDAP Settings

Make sure the web application container protects traffic to the gateway with HTTPS.

Review the following settings in the gateway configuration file, `config.json`:

`security/keyManager`
: If the LDAP server expects client authentication for TLS, set this to access the gateway's keystore.

`security/trustManager`
: Set this to a truststore with the appropriate certificate(s) for remote LDAP servers.

`ldapConnectionFactories/bind/connectionSecurity`
: Use `ssl` or `startTLS`.

`ldapConnectionFactories/bind/sslCertAlias`
: If the LDAP server expects client authentication for TLS, set this to access the gateway's certificate alias.

`ldapConnectionFactories/primaryLdapServers/port`
: Use an LDAP port that supports StartTLS or LDAPS.

  Using StartTLS or LDAPS is particularly important if the gateway ever sends credentials over LDAP.

`authorization/resolver`
: Check the `endpointUrl` of the resolver to make sure that OAuth 2.0 tokens are sent over HTTPS.

For details on settings, see REST to LDAP Reference.

Server Security

Secure DS server installations as outlined below.

Server Account

Do not run DS servers as the system superuser (root) or Windows Administrator.

Run the server under its own account, and use system capabilities to let the server account:

- Access privileged ports, such as `389`, `443`, and `636`, as required.
- Read and write to server files, and execute server commands.
- Log in to the local system.

Use system capabilities to:

- Allow administrator users to run commands as the server user.
- Allow other user to run commands, such as the **`ldapsearch`** command.
- Prevent other users from reading configuration files.

  On UNIX, set a umask such as `027` to prevent users in other groups from accessing server files.

  On Windows, use file access control to do the same.

Encryption

By default, only passwords are protected (hashed rather than encrypted). Encrypt other content to protect your data:

Backend files
: To encrypt entries and index content, see Data Encryption.

Backup files

Backup files are always encrypted. The server uses its Crypto Manager configuration to determine how to encrypt the backup files, and which HMAC algorithm to use to sign and verify backup file integrity. Backup file permissions depend on the UNIX umask or Windows file access control settings.

Changelog files

To encrypt change log files, see Encrypt External Changelog Data.

LDIF exports

When you use the **`export-ldif`** command, encrypt the LDIF output.

File Permissions

Many DS server file permissions depend on the software distribution, not the UNIX file mode creation mask. For example, the server commands are generally executable by all users, but only the server user can read PIN code files.

The following table recommends file permission settings:

Setting	Impact
`umask` of `027`	A UNIX **`umask`** setting of `027` for the server account prevents members of other groups from reading files, and listing keystore contents. Members of the server user's group can still read the files. Use this setting when other processes read the files to process them independently. For example, other processes might copy backup files to a remote system, or parse the logs to look for particular patterns.
`umask` of `077`	A UNIX **`umask`** setting of `077` for the server account prevents members of the server user's group from reading files, and listing keystore contents. This setting can be useful when no other processes need access to server files. Other users can still run commands delivered with the server.

Setting	Impact
`log-file-permissions`	This setting applies to DS-native file-based log publishers on UNIX systems. It does not apply to Common Audit file-based log publishers. Its value is a UNIX mode string. The impact of the setting is independent of the server user's **`umask`** setting. The default for file-based log publishers is `600`. A value of `640` allows only the user read/write access to the logs.
Windows NTFS ACLs	On Windows systems, set folder ACLs on the NTFS volume where the server files are installed. Apply folder permissions that are inherited by all old and new files. Consider setting ACLs on at least the following folders: • The backup folder, by default `/path/to/opendj/bak`. • The configuration folder, `/path/to/opendj/config`. • The logs folder, by default `/path/to/opendj/logs`.

Disable Unused Features

Use the **`status`** command to check which connection handlers are enabled.

Disable any unused connection handlers with the **`dsconfig set-connection-handler-prop --set enabled:false`** command.

Log Settings

By default, DS servers write log messages on error and when the server is accessed. Access logs tend to be much larger than error logs.

The default DS server log levels and rotation and retention policies facilitate troubleshooting while preventing the logs from harming performance or filling up the disk. If you change log settings for more advanced troubleshooting, reset the log configuration to conservative settings when you finish.

Password Management

Make sure you keep passwords secret in production.

In Configuration Files

By default, DS servers keystore passwords in configuration files with `.pin` extensions. These files contain the cleartext, randomly generated passwords. Keep the PIN files readable and writable only by the user running the server.

Alternatively, configure the server to store keystore passwords in environment variables or Java properties. Key Manager Provider and Trust Manager Provider settings let you make this change.

In Command Arguments

DS commands supply credentials for any operations that are not anonymous. Password credentials can be supplied as arguments, such as the `--bindPassword password` option shown in the documentation.

In production, do not let the password appear in commands. Omit the `--bindPassword` option to provide the password interactively:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --baseDN uid=admin \
 "(&)" \
 userPassword

Password for user 'uid=admin':
dn: uid=admin
userPassword: {PBKDF2}10000:<hash>
```

Notice that the password appears neither in the shell history, nor in the terminal session.

When using scripts where the password cannot be supplied interactively, passwords can be read from files. For example, the `--bindPassword:file file` option takes a file that should be readable only by the user running the command.

In Directory Data

By default, DS servers hash passwords before storing them. DS servers support many password storage schemes.

Password policies define password storage schemes, and characteristics of valid passwords. Configure your policies to use strong password storage, and to prevent users from using weak passwords or reusing passwords.

Cryptographic Keys

DS servers use cryptographic keys for encryption, signing, and securing network connections.

Deployment Keys

A *deployment key* is a random string generated by DS software. A *deployment key password* is a secret string at least 8 characters long that you choose. The two are a pair. You must have a deployment key's password to use the key.

Each deployment requires a *single, unique deployment key and its password*. DS uses the pair to:

- Protect the keys to encrypt and decrypt backup files and directory data.
- Generate the TLS key pairs to protect secure connections, unless you provide your own.

Store your deployment key and password in a safe place, and reuse them when configuring other servers in the same deployment.

The DS `setup` and `dskeymgr` commands use the pair to generate the following:

- (Required) A shared master key for the deployment.

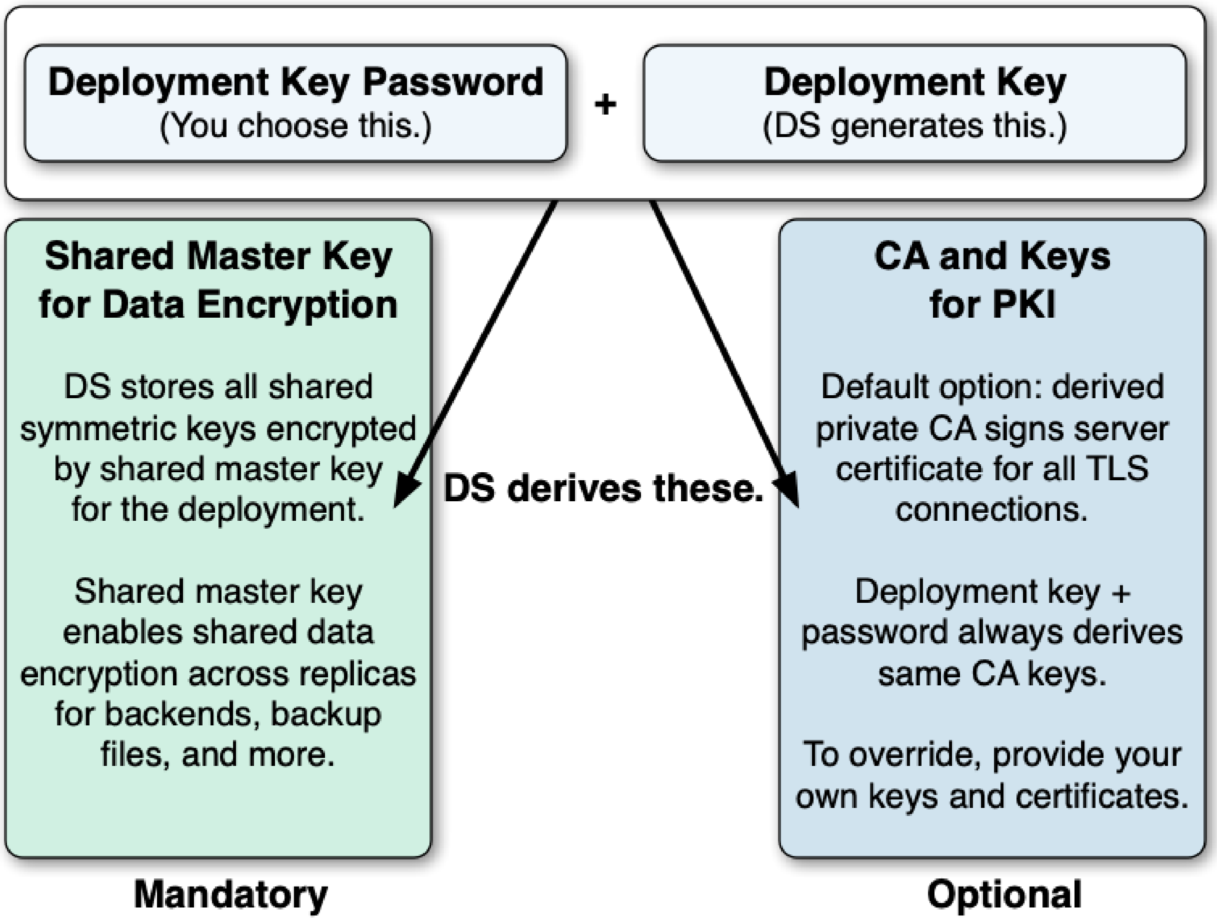
  DS replicas share secret keys for data encryption and decryption. DS servers encrypt backend data, backup files, and passwords, and each replica must be able to decrypt data encrypted on another peer replica.

  To avoid exposing secret keys, DS servers encrypt secret keys with a shared master key. DS software uses a deployment key and its password to derive the master key.
- (Optional) A private PKI for trusted, secure connections.

  A PKI serves to secure network connections from clients and other DS servers. The PKI is a trust network, requiring trust in the CA that signs public key certificates.

Building a PKI can be complex. You can use self-signed certificates, but you must distribute each certificate to each server and client application. You can pay an existing CA to sign certificates, but that has a cost, and leaves control of trust with a third party. You can set up a CA or certificate management software, but that can be a significant effort and cost. As a shortcut to setting up a private CA, DS software uses deployment keys and passwords.

DS software uses the deployment key and its password to generate key pairs without storing the CA private key.


Deployment Key Password
(You choose this.)
+
Deployment Key
(DS generates this.)
Shared Master Key for Data Encryption
DS stores all shared symmetric keys encrypted by shared master key for the deployment.
Shared master key enables shared data encryption across replicas for backends, backup files, and more.
DS derives these.
CA and Keys for PKI
Default option: derived private CA signs server certificate for all TLS connections.
Deployment key + password always derives same CA keys.
To override, provide your own keys and certificates.
Mandatory
Optional

- Initially, you start without a deployment key.

  To generate a new deployment key, either:

  - Use the `dskeymgr create-deployment-key` command to create a new key prior to installation.
  - Run the `setup` command for the first time without specifying a deployment key. The command displays a deployment key in its output.

  You provide the deployment key password when generating the deployment key or setting up the server.

- DS software uses the deployment key and password together to generate a CA key pair.

Every time you use the same deployment key and password, you get the same CA key.

Protect the deployment key password with the same care you would use to protect the CA private key.

- DS software generates a server key pair used for secure communications in a new PKCS#12 keystore.
- DS software signs the server certificate with the CA key.
- DS software derives the shared master key for protecting secret keys, storing the master key in the PKCS#12 keystore.
- DS software writes the CA certificate to the PKCS#12 keystore.
- DS software discards the CA private key temporarily held in memory.

You can use the `dskeymgr` command with an existing deployment key and password to add keys to keystores, or to export them in PEM file format.

This private CA alternative, using a deployment key and password instead, is not appropriate for signing server certificates in some situations:

- External applications you do not control must be able to trust the server certificates.

  In this case, use server certificates signed by a well-known CA.
- Your deployment requires high security around CA private keys.

  If the CA private key needs to be stored in an HSM that it never leaves, you cannot achieve the same level of security with a deployment key and password. The deployment key and password must be provided to sign a certificate, and cannot remain secure in an HSM. Furthermore, the CA private key used to sign the certificate is present in memory during the operation.

Secret Keys and Key Pairs

DS software uses two types of cryptographic keys:

- *Symmetric keys* (also known as secret keys)
- *Asymmetric key pairs* (public/private key pairs)

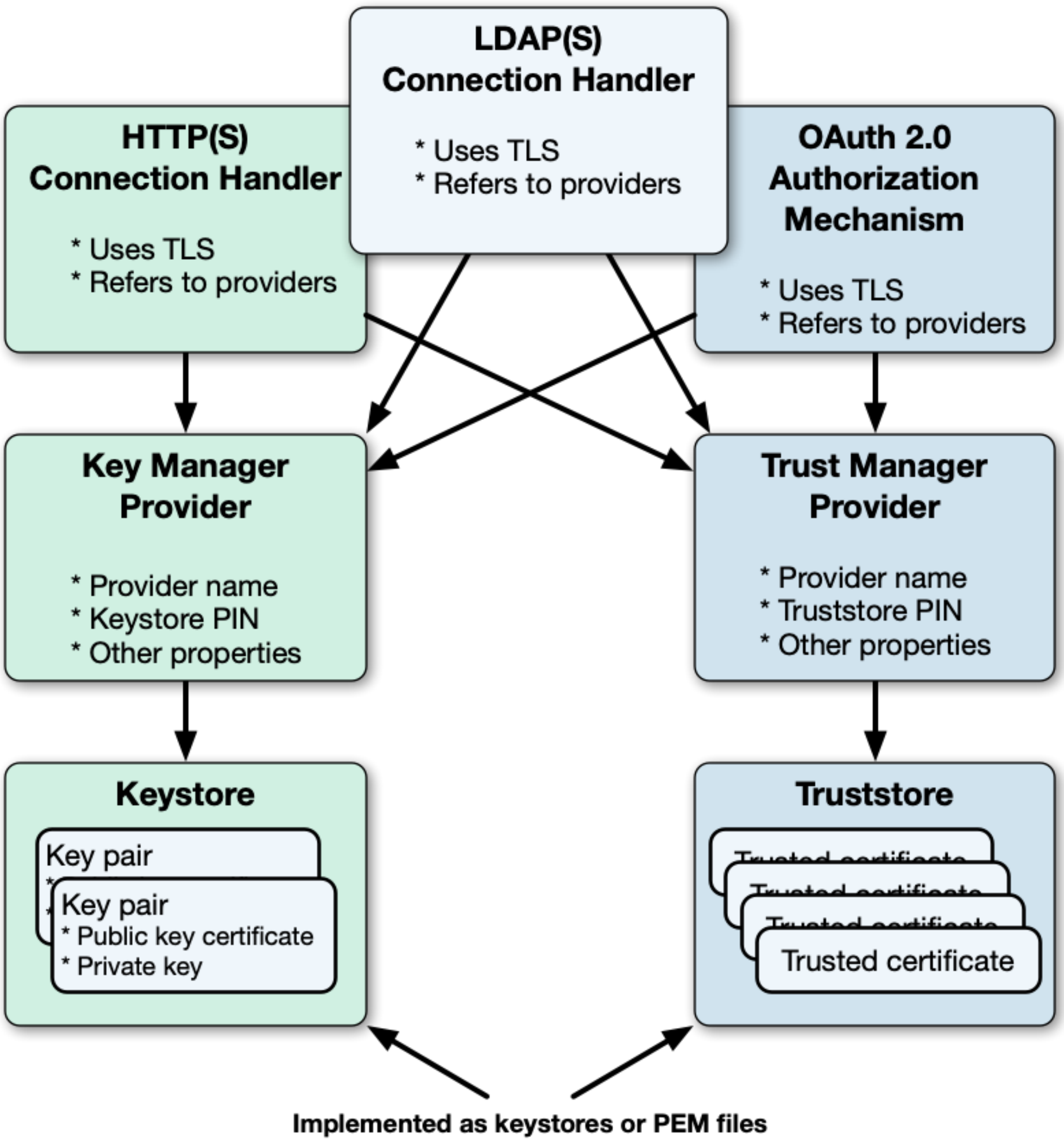
	Symmetric (Secret) Keys	**Asymmetric Key Pairs**
Content	Single key, such as a random array of bits.	Pair of keys, one public, the other private.

	Symmetric (Secret) Keys	Asymmetric Key Pairs
Encryption	A single key serves to encrypt and to decrypt.	Data encrypted with a public key can only be decrypted with the private key, and vice versa.
Generation	Easier to generate, can be a random array of bits.	Harder to generate a matched pair of keys.
Speed	Faster.	Slower.
Distribution	Must be kept secret. Each party must have a copy. Secure channels must be established to exchange secret keys.	Public key can be shared with any party. Private key must be kept secret by owner. No secure channel is required to distribute public keys. Proving that a public key is valid and belongs to the issuer requires a trust network, such as a public key infrastructure (PKI).

	Symmetric (Secret) Keys	Asymmetric Key Pairs
Uses	Encrypting shared data. DS servers use secret keys for data confidentiality, and for encrypted backup files.	• Public key encryption: Encrypt a message with a public key; only the private key owner can decrypt the message. • Digital signatures: Sign a message with the private key; any party can verify the signature with the public key. DS software uses public/private key pairs to establish secure connections. DS servers can use public keys to authenticate clients.

Asymmetric Keys

DS software stores asymmetric key pairs and trusted certificates in keystores or PEM files. In the DS server configuration, key manager providers reference Java keystores or PEM files. (Except for some ForgeRock Common Audit event handlers that manage their own keystores.) Trust manager providers also reference Java keystores or PEM files. Components that access keys reference key manager providers for their certificates and private keys, and trust manager providers for trusted certificates. This enables, but does not require, reuse:



DS servers use keystores or PEM files for server keys and trusted certificates. By default, each server stores keys in a file-based keystore, `/path/to/opendj/config/keystore`. The cleartext password is stored in a `keystore.pin` file next to the keystore file. This password serves as the password for the keystore and each private key.

The password for the keystore and for private keys must be the same. DS servers do not support using different passwords for the keystore and private keys.

By default, the `keystore` file holds these keys based on the deployment key and password:

- The CA certificate
- The shared master key
- A key pair for secure communications

Symmetric Keys

DS servers encrypt data using symmetric keys. Servers store symmetric keys, encrypted with the shared master key, with the data they encrypt. As long as servers have the same shared master key, any server can recover symmetric keys needed to decrypt data.

Symmetric keys for (deprecated) reversible password storage schemes are the exception to this rule. When you configure a reversible password storage scheme, enable the `adminRoot` backend, and configure a replication domain for `cn=admin data`.

Public Key Infrastructure

A public key infrastructure (PKI) is a set of procedures, roles, and policies required to enable parties to trust each other. A PKI makes it possible to trust that a public key belongs to its owner by enabling the following steps:

1. Each party in the PKI trusts one or more *certificate authorities* (CAs).

   Trusting a CA means trusting that it owns its public key, that it maintains its private key secret, and that it only signs another party's certificate according to standard operating procedures.

   The decision to trust a CA is a prerequisite for other operations, such as negotiating secure connections.

   Trusting a CA equates to storing a trusted copy of the CA's certificate. The trusted copy is used to verify CA signatures.
2. Other trusted parties get their keys certified in one of the following ways:
   - A party wanting to use public key cryptography requests a CA-signed certificate for its key pair:
     - The owner generates a key pair with a *public key* to share and a *private key* to keep secret.

       This key pair is generated for the public key *subject* (or owner).
     - The owner makes a certificate signing request to a trusted CA. The request includes the public key.
     - The CA verifies that the party making the request is indeed the party making the request. If so, the CA signs the certificate request, resulting in the signed public key certificate. The CA responds to the owner with the signed certificate as the response to the request.

       Notice that the certificate is a digital document that certifies ownership of the public key. The certificate includes the public key and other information, such as the validity period, who the subject (owner) is, and the digital signature of the *issuer* CA who signed the certificate.

    - A party registers for an account with a service provider that uses certificate-based authentication.

      The service provider, acting as a CA, issues a key pair including a certificate to the account owner over a secure channel. The service provider stores a copy of the certificate with the owner's account.

3. The owner stores the signed certificate, and shares it when necessary with other parties for public key encryption and signature verification.

   It stores the private key in a safe manner and never shares it.

4. Another party wanting to trust the certificate must verify that the certificate is valid.

   Certificate verification involves checking certificate information such as the following:

    - Whether the current time is in the range of the validity dates.
    - Whether the owner's subject identifier in the certificate matches some externally verifiable attribute of the owner, such as the DNS record of the host FQDN or the owner's email address.
    - Whether the certificate has been signed by a trusted party.

      A public CA does not sign certificates with its root certificate directly. Instead, the CA issues signing certificates to itself, and uses them to sign other certificates. Trust is verified for the certificate chain, whereby the root certificate signature on the signing certificate makes it possible to trust the signing certificate. The signing certificate signature on the owner's certificate makes it possible to trust the owner's certificate.

    - Whether the party providing the certificate with the public key can prove that is has the corresponding private key.

      For example, the verifier supplies a nonce for the party to sign with the private key, and verifies the signature with the public key in the certificate.

5. Ultimately, the chain of verification must:

    - End by determining that the issuer's signature is valid and trusted, and that the party providing the certificate is authenticated.
    - Fail at some point, in which case, the signature cannot be trusted.

      This does not mean that the certificate is invalid. It does mean, however, that the party that wants to use the public key cannot be certain that it belongs to the owner, and so, cannot trust the public key.

      This can happen when the party trying to use the public key does not have a means to trust the issuer who signed the certificate. For example, it has no trusted copy of the issuer’s certificate.

Trusted Certificates

Secure connections between server and client applications are based on TLS. TLS depends on the use of digital certificates. By default, DS servers present their certificates when establishing a secure connection. This process fails if the client cannot trust the server certificate.

By default, DS client tools prompt you if they do not recognize the server certificate. DS servers have no one to prompt, so they refuse to accept a connection with an untrusted certificate. For ease of use, your deployment should enable secure connections without user interaction.

Automating trust is based on configuring applications to trust the CAs who sign the certificates. To achieve this, operating systems, JVMs, and web browsers ship with many trusted public CA certificates. On one hand, this prevents end users from having to understand PKI before using secure connections. On the other hand, it also introduces some risk. When public CAs are installed by default, the user must trust:

- The software distribution mechanism used to obtain the original software and subsequent updates.
- The software distributor to vet each CA and make sure the CA remains worthy of trust.
- The CAs to perform their CA duties correctly.
- The whole process to be safe from serious bugs.

Stronger security requires that you take more control. You can do this by using a private CA to distribute keys used for private communications.

Use	Private CA	Public CA	Rationale
Private connections	✓		You control both parties making the connection.

Use	Private CA	Public CA	Rationale
Public connections		✓	Your service publishes information over HTTPS or LDAPS to unknown end user clients. Your service connects as a client to public HTTPS or LDAPS services.
Mutual TLS	✓		When your private CA signs the client certificate, store certificate information for authentication on the client's entry in the directory.
Replication	✓		Replication messages are private to your service.
Service administration	✓		Service administration is private to your service.

Key Management

Update Keys

When you update keys, DS servers load them and begin using them for new connections. This section covers common rotation operations, such as renewing and replacing keys.

Renew a TLS Certificate

1. Choose the appropriate method to renew an expiring certificate:
   a. If you set up the server with a deployment key, renew it with the **dskeymgr** command:

   ```
   $ dskeymgr \
    create-tls-key-pair \
    --deploymentKey $DEPLOYMENT_KEY \
    --deploymentKeyPassword password \
    --keyStoreFile /path/to/opendj/config/keystore \
    --keyStorePassword:file
   /path/to/opendj/config/keystore.pin \
    --hostname localhost \
    --hostname ds.example.com \
    --subjectDn CN=DS,O=ForgeRock
   ```

   For more command options, refer to dskeymgr. The default validity for the certificate is one year.

   b. If you use a CA, renew the CA signature:
      - Create a certificate signing request.
      - Have the request signed by the CA you use.
      - Import the signed certificate from the CA reply.

   c. If you used a self-signed certificate, sign it again, and distribute the new certificate as appropriate.

   Have each server and client application that trusted the old certificate import the renewed certificate.

   For details, see Trust a Server Certificate.

Replace a TLS Key Pair

1. Choose the appropriate method to replace or rotate a key pair used for secure communications:
   a. If the certificate depends on a deployment ID and password, use the **dskeymgr** command:

   ```
   $ dskeymgr \
    create-tls-key-pair \
    --deploymentKey $DEPLOYMENT_KEY \
   ```

```
 --deploymentKeyPassword password \
 --keyStoreFile /path/to/opendj/config/keystore \
 --keyStorePassword:file
/path/to/opendj/config/keystore.pin \
 --hostname localhost \
 --hostname ds.example.com \
 --subjectDn CN=DS,O=ForgeRock
```

The default validity for the certificate is one year.

b. If you provided your own keys in the server keystore, do the following:

- Generate a new key pair in the server keystore with a new alias.
- If you use a CA, get the certificate signed by the CA.
- If you use a self-signed certificate, distribute the certificate for import by all servers and clients that trust your server.
- Once the old key pair is no longer used anywhere, delete the keys using the **`keytool -delete`** command with the old alias.

Retire Secret Keys

You do not need to retire secret keys for the following, because encryption is only performed once per key:

- Backup uses a new secret key for each file.
- Confidential replication changelog backends use a new secret key for each file.

You can retire secret keys for confidential database backends:

1. Change the cipher-key-length property for the backend.

   Each time you change the setting, the server generates a new secret key. After you retire the key, DS servers will only use that key for decryption, not for encryption.

Replace Deployment Keys

Follow these steps if you must:

- Renew an expiring deployment key-based CA.

  The default validity period is 10 years.
- Replace a lost or compromised deployment key password.

1. Generate a new deployment key with a new password:

   ```
   $ dskeymgr \
    create-deployment-key \
    --outputFile new.deployment.key \
    --deploymentKeyPassword password
   ```

   For more command options, refer to dskeymgr. The default validity for the deployment key is 10 years.

2. On each server, add the new shared master key:

   ```
   $ dskeymgr \
    export-master-key-pair \
    --alias new-master-key \
    --deploymentKey "$(<new.deployment.key)" \
    --deploymentKeyPassword password \
    --keyStoreFile /path/to/opendj/config/keystore \
    --keyStorePassword:file
   /path/to/opendj/config/keystore.pin
   ```

   Servers continue to use the existing shared master key to decrypt existing symmetric keys. Do not overwrite a shared master key that is already in use.

3. On each server that uses the deployment key for PKI, add the new CA certificate:

   ```
   $ dskeymgr \
    export-ca-cert \
    --alias new-ca-cert \
    --deploymentKey "$(<new.deployment.key)" \
    --deploymentKeyPassword password \
    --keyStoreFile /path/to/opendj/config/keystore \
    --keyStorePassword:file
   /path/to/opendj/config/keystore.pin
   ```

   Also distribute the new CA certificate to any client applications that rely on the old CA certificate.

4. On each server that uses the deployment key for PKI, renew the key pair used for secure communications.

   Before completing this step, make sure you have added the new CA certificate *on all servers*. Any peer servers missing the new CA certificate will not trust the new keys:

```
$ dskeymgr \
 create-tls-key-pair \
 --deploymentKey "$(<new.deployment.key)" \
 --deploymentKeyPassword password \
 --keyStoreFile /path/to/opendj/config/keystore \
 --keyStorePassword:file
/path/to/opendj/config/keystore.pin \
 --hostname localhost \
 --hostname ds.example.com \
 --subjectDn CN=DS,O=ForgeRock
```

For more command options, refer to dskeymgr. The default validity for the certificate is one year.

Also renew any client application key pairs that were generated using the old deployment key and password.

5. On each server, update the master key alias to use the new key:

```
$ dsconfig \
 set-crypto-manager-prop \
 --set master-key-alias:new-master-key \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

6. Stop trusting the old CA certificate by removing all references to it.

   For example, remove the old CA certificate from all client and server truststores.

7. When installing a new server after replacing deployment keys:
   - Use the new deployment key to set up the server, and do not start the server at setup time.
   - Copy the keystore and PIN from an existing server, overwriting the existing keystore and PIN.

     This adds the older shared master key to the new server's keystore.
   - Renew the local key pair used for secure communications using the new deployment key.

- Start the server.

Use New Keys

Generate a Key Pair (Wildcard Certificate)

A wildcard certificate uses a `*` to replace the top-level subdomain in the subject FQDN. It can list domains in the subject alternative domain list:

1. Generate a key pair with a wildcard certificate:

   ```
   $ dskeymgr \
    create-tls-key-pair \
    --deploymentKey $DEPLOYMENT_KEY \
    --deploymentKeyPassword password \
    --hostname localhost \
    --hostname "*.example.com" \
    --subjectDn CN=DS,CN=Example,CN=com \
    --keyStoreFile /path/to/opendj/config/keystore \
    --keyStorePassword:file
   /path/to/opendj/config/keystore.pin
   ```

   For more command options, refer to dskeymgr. The default validity for the certificate is one year.

Generate a Key Pair (CA-Signed Certificate)

1. Generate a server key pair in the existing server keystore.

   Many client applications will check that the server's DNS name in the certificate matches the server hostname.

   Find the hostname for the server in the **`status`** command output. When creating the key pair, set it as a `DNSName` in the certificate's `SubjectAlternativeName` list. If the server can respond on multiple FQDNs, use multiple subject alternative names.
2. Create a certificate signing request `.csr` file for the generated certificate.
3. Have the CA sign the request in the `.csr` file.

   See the instructions from your CA on how to provide the request.

The CA returns the signed certificate, for example, in a `.crt` file.

4. If you have set up your own CA and signed the certificate, or are using a CA whose certificate is not included in the Java runtime environment, import the CA certificate into the DS keystore and truststore so that it can be trusted.

   For an example command, see Trust a CA Certificate.

5. Import the signed certificate, such as the `.crt` file, from the CA reply into the keystore where you generated the key pair.
6. If you use a CA certificate that is not known to clients, such as a CA that you set up yourself rather than a well-known CA, import the CA certificate into the client application truststore. For an example command, see Trust a CA Certificate.

   Otherwise, the client application cannot trust the signature on the server certificate.

Generate a Key Pair (Self-Signed Certificate)

1. Generate a server key pair in the existing server keystore.

   The certificate is considered self-signed, because the issuer DN and subject DN are the same.

   Many client applications will check that the server's DNS name in the certificate matches the server hostname.

   Find the hostname for the server in the **`status`** command output. When creating the key pair, set it as a `DNSName` in the certificate's `SubjectAlternativeName` list.

Use an Alternative Keystore Type

To use an alternative keystore implementation, start with a different keystore type when generating the keypair.

IMPORTANT

When generating a key pair for TLS with the **`keytool`** command, set the `-keyalg` option to `EC` or `RSA` for compatibility with TLSv1.3.

The `-keyalg DSA` option is not compatible with TLSv1.3.

1. Use one of the keystore types supported by the Java runtime environment:

Java Keystore

The basic Java keystore type is `JKS`:

```
$ keytool \
 -genkeypair \
 -keyalg EC \
 -alias new-keys \
 -ext "san=dns:ds.example.com" \
 -dname "CN=ds.example.com,O=Example Corp,C=FR" \
 -keystore /path/to/new-keystore.jks \
 -storetype JKS \
 -storepass:env KEYSTORE_PASSWORD \
 -keypass:env KEYSTORE_PASSWORD
```

This is the keystore type if you do not specify a `-storetype` option.

Java Cryptography Extension Keystore

The `JCEKS` type implements additional Java cryptography extensions and stronger protection for private keys:

```
$ keytool \
 -genkeypair \
 -keyalg EC \
 -alias new-keys \
 -ext "san=dns:ds.example.com" \
 -dname "CN=ds.example.com,O=Example Corp,C=FR" \
 -keystore /path/to/new-keystore.jceks \
 -storetype JCEKS \
 -storepass:env KEYSTORE_PASSWORD \
 -keypass:env KEYSTORE_PASSWORD
```

PKCS#11 device

A PKCS#11 device, such as an HSM, can be used as a keystore.

For details, refer to PKCS#11 Hardware Security Module.

PKCS#12 Keystore

The `PKCS12` type lets you use a PKCS#12 format file. This is the default for DS servers. It is a standard format and is interoperable with other systems that do not use Java:

```
$ keytool \
 -genkeypair \
 -keyalg EC \
 -alias new-keys \
 -ext "san=dns:ds.example.com" \
 -dname "CN=ds.example.com,O=Example Corp,C=FR" \
 -keystore /path/to/new-keystore \
 -storetype PKCS12 \
 -storepass:env KEYSTORE_PASSWORD \
 -keypass:env KEYSTORE_PASSWORD
```

2. After setting up an alternate keystore type, make sure that you configure:
   - A key manager provider to open the correct keystore with the correct credentials.
   - Any components using the key manager provider to use the correct certificate alias.

Add a New Keystore

These steps demonstrate adding a new PKCS#12 keystore with existing server keys. Follow these steps if you have existing keys in a PKCS#12 keystore:

1. Create a Key Manager Provider that references your keystore:

```
$ dsconfig \
 create-key-manager-provider \
 --provider-name MyKeystore \
 --type file-based \
 --set enabled:true \
 --set key-store-file:/path/to/keystore \
 --set key-store-pin:password \
 --set key-store-type:PKCS12 \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

2. For each configuration object that needs to use the keys in the keystore, update the `key-manager-provider` and `ssl-cert-nickname` properties:

```
$ dsconfig \
 set-connection-handler-prop \
 --handler-name LDAPS \
 --set key-manager-provider:MyKeystore \
 --set ssl-cert-nickname:ssl-key-pair \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

Many configuration objects use key manager providers. For a full list, see the page of configuration properties that start with K, and review the `key-manager-provider` properties links.

Do not update the key manager provider for the Crypto Manager. The Crypto Manager needs access to the shared master key.

Generate a Test Deployment Key

1. Use the **`dskeymgr`** command to generate a deployment key for testing.

   This example records the deployment key in a file:

   ```
   $ dskeymgr \
    create-deployment-key \
    --deploymentKeyPassword password \
    --outputFile test.deployment.key
   ```

   If you do not specify the deployment key password or file containing the password, the tool prompts you for the password interactively.

   Use this deployment key when setting up test servers. The test servers trust each others' certificates.

Trust Certificates

Trust a Client Certificate

These steps let the DS server trust a self-signed client application certificate:

NOTE

NOTE

If you control the application, issue the client a certificate from a private CA instead.

For an example, see Certificate-Based Authentication.

1. Import the self-signed client certificate.

   The following example imports the client certificate into the default truststore:

```
$ keytool \
 -import \
 -trustcacerts \
 -alias myapp-cert \
 -file myapp-cert.pem \
 -keystore /path/to/opendj/config/keystore \
 -storepass:file /path/to/opendj/config/keystore.pin \
 -storetype PKCS12 \
 -noprompt

Certificate was added to keystore
```

   If the truststore was provided during the setup process, specify the truststore and password.

Trust a Server Certificate

These steps let a client trust the DS server.

If the server certificate was signed by a well-known CA, the client may not have to configure any further trust.

To trust a certificate signed using a deployment key and password, get the CA certificate. Either:

- Copy the server's truststore file and PIN.
- Export the CA certificate as a PEM format file, as described in PEM Format Keys.

For an unknown CA or a self-signed server certificate, follow these steps:

1. Export the CA certificate from its truststore or the server certificate from the server keystore.

You can export the certificate in PEM format using the **`keytool -exportcert -rfc`** command.

Some client applications can use the PEM format directly.

2. If the client uses a Java truststore, import the certificate into the client truststore.

   You can import the server certificate using the **`keytool -import -trustcacerts`** command.

Trust a CA Certificate

These steps let the DS server trust a CA. If the CA's certificate is included with Java, you can let the server use the JVM truststore.

If the CA is not well-known, you can import the trusted CA certificate into a truststore:

1. Import the certificate as a CA certificate.

   The following example imports the CA certificate into the default truststore:

```
$ keytool \
 -import \
 -trustcacerts \
 -alias my-ca-cert \
 -file ca.pem \
 -keystore /path/to/opendj/config/keystore \
 -storepass:file /path/to/opendj/config/keystore.pin \
 -storetype PKCS12 \
 -noprompt

Certificate was added to keystore
```

   If the truststore was provided during the setup process, specify the truststore and password.

2. If no trust manager provider is configured to access the truststore, create one.

   For reference, see create-trust-manager-provider.

   Also, configure connection handlers to use the new trust manager provider.

Add a New Truststore

These steps demonstrate adding a new PKCS#12 truststore with existing trusted certificates. Follow these steps if you have existing certificates in a PKCS#12 truststore:

1. Create a Trust Manager Provider that references your truststore:

```
$ dsconfig \
 create-trust-manager-provider \
 --provider-name MyTruststore \
 --type file-based \
 --set enabled:true \
 --set trust-store-file:/path/to/truststore \
 --set trust-store-pin:password \
 --set trust-store-type:PKCS12 \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

2. For each configuration object that needs to trust the certificates in the truststore, update the `trust-manager-provider` property:

```
$ dsconfig \
 set-connection-handler-prop \
 --handler-name LDAPS \
 --set trust-manager-provider:MyTruststore \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

   Many configuration objects use trust manager providers. For a full list, see the page of configuration properties that start with T, and review the `trust-manager-provider` properties links.

PEM Format Keys

DS servers can read keys and trusted certificates from files that contain keys in Privacy-Enhanced Mail (PEM) format:

1. Choose or create a directory for PEM format keys.

   This example uses `/path/to/opendj/pem`:

   ```
   $ mkdir -p /path/to/opendj/pem
   ```

2. Use the **`dskeymgr`** command to generate PEM format keys.

   - Add a master key using the deployment key and password, even if you have your own CA and server keys:

     ```
     $ dskeymgr \
      export-master-key-pair \
      --deploymentKey $DEPLOYMENT_KEY \
      --deploymentKeyPassword password \
      --outputFile /path/to/opendj/pem/master-key.pem
     ```

   - Add a trusted CA certificate.

     This example exports a deployment key CA certificate:

     ```
     $ dskeymgr \
      export-ca-cert \
      --deploymentKey $DEPLOYMENT_KEY \
      --deploymentKeyPassword password \
      --outputFile /path/to/opendj/pem/ca-cert.pem
     ```

   - Add server keys.

     This example exports a server key pair based on a deployment key:

     ```
     $ dskeymgr \
      create-tls-key-pair \
      --deploymentKey $DEPLOYMENT_KEY \
      --deploymentKeyPassword password \
      --hostname localhost \
      --hostname ds.example.com \
      --subjectDn CN=DS,O=ForgeRock \
      --outputFile /path/to/opendj/pem/ssl-key-pair.pem
     ```

For more command options, refer to dskeymgr. The default validity for the certificate is one year.

3. Allow only the user running the server to read any PEM files that contain private keys.

   The keys are not encrypted, so you must protect the PEM files. For example, if the server runs with user ID `opendj`, restrict access to that user:

   ```
   $ sudo chown opendj /path/to/opendj/pem/*.pem && \
     sudo chmod 600 /path/to/opendj/pem/master-key.pem && \
     sudo chmod 600 /path/to/opendj/pem/ssl-key-pair.pem
   ```

4. Configure a PEM key manager provider for the master key and server keys:

   ```
   $ dsconfig \
    create-key-manager-provider \
    --provider-name PEM \
    --type pem \
    --set enabled:true \
    --set pem-directory:/path/to/opendj/pem \
    --hostname localhost \
    --port 4444 \
    --bindDN uid=admin \
    --bindPassword password \
    --usePkcs12TrustStore /path/to/opendj/config/keystore \
    --trustStorePassword:file
   /path/to/opendj/config/keystore.pin \
    --no-prompt
   ```

5. Configure a PEM trust manager provider for trusted certificates, such as CA certificates:

   ```
   $ dsconfig \
    create-trust-manager-provider \
    --provider-name PEM \
    --type pem \
    --set enabled:true \
    --set pem-directory:/path/to/opendj/pem \
    --hostname localhost \
    --port 4444 \
    --bindDN uid=admin \
    --bindPassword password \
    --usePkcs12TrustStore /path/to/opendj/config/keystore \
    --trustStorePassword:file
   ```

```
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

6. Configure other components to use the new providers by setting their `key-manager-provider` and `trust-manager-provider` properties.

   When using PEM keys, the alias or `ssl-cert-nickname` property is filename of the key. In this example:

   - The master key alias is `master-key.pem`.
   - The CA certificate alias is `ca-cert.pem`.
   - The TLS keys alias is `ssl-key-pair.pem`.

Notice that the PEM key manager provider, and the PEM trust manager provider share the same `pem-directory`. This works because the key manager provider loads key pairs, and the trust manager provider loads trusted certificates. When the PEM files change, the server regularly reloads the files. For details, see Pem Key Manager Provider and Pem Trust Manager Provider.

PKCS#11 Hardware Security Module

DS servers support key management using a PKCS#11 token store. The PKCS#11 standard defines a cryptographic token interface, a platform-independent API for storing keys in an HSM, for example.

IMPORTANT

IMPORTANT

DS servers use an HSM only to hold asymmetric key pairs and, optionally, CA certificates.

Since the CA certificate holds the CA's *public* key, ForgeRock recommends storing it in a separate, file-based keystore or PEM file, not in the HSM.

The asymmetric key pairs include the server's TLS keys, and the shared master key for the deployment.

DS servers use the shared master key to wrap symmetric (secret) keys. DS servers store secret keys in the data they encrypt. Therefore, DS servers do not use the HSM for secret keys.

You can store the master key, technically an asymmetric key pair, on the HSM as long as the HSM supports importing the master key. Generate the master key (pair) into a PKCS#12 keystore, or PEM file, and import the keys into the HSM.

If you store the shared master key in an HSM, the HSM *must share the same master key with all DS servers in the deployment*. Each server unwraps secret keys with the master key. Without access to the shared master key, DS servers cannot read each others' encrypted data.

Using a PKCS#11 device for storing DS keys involves:

- Storing the keys on the PKCS#11 device.

  How you store keys in a device such as an HSM depends on the device. For details, see the documentation for your device.

- Creating the DS PKCS11 key manager provider to access the device.

  The DS server accesses a PKCS#11 device using a PIN. The PIN can be provided in the server configuration, in a separate file, or as the value of an environment variable or Java system property. The PIN must be stored as a cleartext value, so take care to protect it in production environments.

- Configuring other components to use the key manager provider.

  For example, DS connection handlers and OAuth 2.0 authorization mechanisms requiring mutual TLS can reference the key manager provider in their configurations.

The following procedures demonstrate how to use the SoftHSM PKCS#11 software device for evaluation, development, and testing:

Prepare the HSM Simulator

The procedures uses the `sun.security.pkcs11.SunPKCS11` provider implementation with SoftHSM. If you use a different Java implementation, see the documentation for details on how to use PKCS#11 devices with your JVM:

1. Install SoftHSM, including the configuration and the `SOFTHSM2_CONF` environment variable.

   For details, see the SoftHSM documentation, using the following hints:

   - Make sure you can write tokens to SoftHSM:

     ```
     $ cat $SOFTHSM2_CONF

     # SoftHSM v2 configuration file

     # You must be able to write to the token dir when
     initializing the device:
     directories.tokendir = /path/to/softhsm/tokens
     objectstore.backend = file

     # ERROR, WARNING, INFO, DEBUG
     log.level = INFO
     ```

   - Keep track of the PINs that you enter when initializing the device:

     ```
     $ softhsm2-util --init-token --slot 0 --label "My token
     1"

     *** SO PIN (4-255 characters) ***
     Please enter SO PIN:
     Please reenter SO PIN:
     *** User PIN (4-255 characters) ***
     Please enter user PIN:
     Please reenter user PIN:
     The token has been initialized.
     ```

     The SO PIN is to reinitialize the token.

     The user PIN is the one the DS server needs to access the device.

2. Generate a key pair on the device:

   - To use the Java **`keytool`** command with the device, create a PKCS#11 configuration file that is used by the security provider implementation:

```
$ cat /path/to/softhsm/hsm.conf

name = SoftHSM
library =
/path/to/softhsm/2.0.0/lib/softhsm/libsofthsm2.so
slot = 0
attributes(generate, *, *) = {
   CKA_TOKEN = true
}
attributes(generate, CKO_CERTIFICATE, *) = {
   CKA_PRIVATE = false
}
attributes(generate, CKO_PUBLIC_KEY, *) = {
   CKA_PRIVATE = false
}
attributes(*, CKO_SECRET_KEY, *) = {
   CKA_PRIVATE = false
}
```

Notes regarding the configuration file:

- The format is described in the Java PKCS#11 Reference Guide.
- The `library` must point to the SoftHSM `libsofthsm2.so` library.
- The `slot` must be one used when initializing the device.

◦ Using the configuration file, generate the key pair.

The following example generates a key pair with the alias `server-cert`:

```
$ keytool \
 -genkeypair \
 -alias server-cert \
 -keyalg EC \
 -keysize 2048 \
 -ext "san=dns:ds.example.com" \
 -dname "CN=ds.example.com,O=Example Corp,C=FR" \
 -keystore NONE \
 -storetype PKCS11 \
 -providerClass sun.security.pkcs11.SunPKCS11 \
 -providerArg /path/to/softhsm/hsm.conf

Enter keystore password:
```

The keystore password is the user PIN.

- Self-sign the public key certificate:

  ```
  $ keytool \
   -selfcert \
   -alias server-cert \
   -keystore NONE \
   -storetype PKCS11 \
   -providerClass sun.security.pkcs11.SunPKCS11 \
   -providerArg /path/to/softhsm/hsm.conf

  Enter keystore password:
  ```

  The keystore password is the user PIN.

  Using a CA-signed cert is similar, but not shown here.

Create a PKCS#11 Key Manager Provider

1. Make sure you have the plain text PIN.

   With SoftHSM, this is the user PIN set when initializing the slot where you stored the keys.
2. Make sure that the Java environment can find SoftHSM with its configuration.

   For example, add a provider definition by using an extra Java security properties file as in the following example. In this example, 10 security providers are already defined in the system `java.security` file. The configuration must use the next available key, which is `security.provider.11`:

   ```
   # Define the additional security provider in the extra
   file:
   $ cat /path/to/opendj/config/java.security

   # Security provider for accessing SoftHSM:
   security.provider.11=sun.security.pkcs11.SunPKCS11
   /path/to/softhsm/hsm.conf

   # Use the extra file when starting the DS server:
   $ grep java.security
   /path/to/opendj/config/java.properties

   start-ds.java-args=-server -
   ```

```
Djava.security.properties=/path/to/opendj/config/java.secu
rity

# Restart the DS server so the changes take effect:
$ stop-ds --restart
```

3. Create the PKCS#11 key manager provider configuration.

   The following example creates a provider for SoftHSM with a protected PIN file:

```
$ touch /path/to/opendj/config/softhsm.pin

$ chmod 600 /path/to/opendj/config/softhsm.pin

$ vi /path/to/opendj/config/softhsm.pin

# Add the user PIN on the first and only line in the file,
and save your work.
$ dsconfig \
 create-key-manager-provider \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --provider-name SoftHSM \
 --type pkcs11 \
 --set enabled:true \
 --set key-store-pin:"&
{file:/path/to/opendj/config/softhsm.pin}" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

   DS key manager providers also support storing the PIN in the configuration, in an environment variable, or in a Java property.

Use the PKCS#11 Key Manager Provider

1. Set a connection handler or authorization mechanism to use the PKCS#11 key manager provider.

   The following example configures the LDAPS connection handler to use the SoftHSM provider:

```
$ dsconfig \
 set-connection-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name LDAPS \
 --set listen-port:1636 \
 --set enabled:true \
 --set use-ssl:true \
 --set key-manager-provider:SoftHSM \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

2. Verify that the secure connection negotiation works with the HSM configured:

```
$ ldapsearch \
 --hostname ds.example.com \
 --port 1636 \
 --useSSL \
 --baseDN dc=example,dc=com \
 "(uid=bjensen)" \
 cn

The server is using the following certificate:
    Subject DN:  CN=ds.example.com, O=Example Corp, C=FR
    Issuer DN:  CN=ds.example.com, O=Example Corp, C=FR
    Validity:  <validity-period>
Do you wish to trust this certificate and continue
connecting to the server?
Please enter "yes" or "no": yes

dn: uid=bjensen,ou=People,dc=example,dc=com
cn: Barbara Jensen
cn: Babs Jensen
```

Secure Connections

Securing connections depends on PKI and asymmetric, public/private key pairs. For details, see Cryptographic Keys.

About Secure Connections

Incoming connections are from clients to the directory. To match port numbers with protocols, see Administrative Access.

Outgoing connections are from the directory to another service.

Recommendations For Incoming Connections

Protocol	Recommendations
Administration	DS servers use an Administration Connector for connections from administration tools. Leave the Administration Connector configured to use SSL/TLS unless you are certain the connections are already secured by some other means.
DSML	DSML support is available through the DS DSML gateway. Use HTTPS to protect client connections.
HTTP	HTTP connections that are not protected by SSL/TLS use cleartext messages. When you permit insecure connections, you cannot prevent client applications from sending sensitive data. For example, a client could send unprotected credentials in an HTTP Authorization header. Even if the server were to reject the request, the credentials would already be leaked to any eavesdroppers. HTTP could be allowed instead of HTTPS with anonymous connections if only public information is exposed, and no client applications send credentials or other sensitive information. Configure the HTTP connection handler to use only the default HTTP Anonymous authorization mechanism.

Protocol	Recommendations
HTTPS	Prefer HTTPS for secure connections over HTTP. When using an HTTP connection handler, use HTTPS to protect client connections. Some client applications require a higher level of trust, such as clients with additional privileges or access. Client application deployers might find it easier to manage public keys as credentials than to manage user name/password credentials. Client applications can use SSL client authentication. When using DS REST to LDAP gateway, use HTTPS to protect client connections.
JMX	Secure JMX access with the SSL/TLS-related properties, such as `use-ssl` and others.
LDAP	LDAP connections that are not protected by SSL/TLS use cleartext messages. When you permit insecure connections, you cannot prevent client applications from sending sensitive data. For example, a client could send unprotected credentials in an LDAP simple bind request. Even if the server were to reject the request, the credentials would already be leaked to any eavesdroppers. If all the LDAP applications are under your control, make sure that the only "insecure" requests are anonymous binds, SASL binds, or StartTLS requests.
LDAPS	Prefer LDAPS for secure connections, or make sure that applications use StartTLS after establishing an insecure LDAP connection and before performing other operations. Some client applications require a higher level of trust, such as clients with additional privileges or access. Client application deployers might find it easier to manage public keys as credentials than to manage user name/password credentials. Client applications can use SSL client authentication. See Certificate-Based Authentication.

Protocol	Recommendations
Replication	Replication is required in all but a few deployments. If *any of the following are true*, replication is required: • Client applications require highly available access to critical services, such as authentication and updates. • Client applications have specific quality of service requirements. • Client applications use the directory service to share common data. • Directory service downtime, either planned or unplanned, can lead to lost organizational or business opportunities. • Backup operations must be performed while the service is online. • Update and upgrade operations must be performed while the service is online. • Load sometimes exceeds the service that a single server can provide. • Global directory services must be available at more than one location. Configure replication to use secure connections.
SNMP	Secure SNMP access with settings for `security-level` and related properties.

Protocol	Recommendations
SSH	DS administration tools can connect securely. If the firewall is configured to prevent remote access to the administration connector port, then use a secure third-party tool to access the system remotely. A recommended choice for UNIX and Linux systems is Secure Shell (SSH). The user account for running directory services should not be the same user account for connecting remotely. Instead, connect as a another user who can then assume the role of the directory services account. The following example demonstrates this approach: `# Log in to ds.example.com:` `me@my-laptop $ ssh user@ds.example.com` `user@ds.example.com's password:` `# Logged in to ds.example.com as user.` `Last login: ... from ...` `# Run dsconfig interactively as opendj:` `user@ds.example.com $ sudo -i -u opendj dsconfig` Secure Copy (SCP) uses SSH to transfer files securely. SCP is an appropriate protocol for copying backup data, for example.

Recommendations For Outgoing Connections

Client	Recommendations
Common Audit event handlers	Configure ForgeRock Common Audit event handlers to use HTTPS or TLS when connecting to external log services.
DSML gateway	The DS DSML gateway connects to remote LDAP directory servers. Use LDAP and StartTLS or LDAPS to protect the connections.
OAuth 2.0-based HTTP authorization mechanisms	HTTP authorization can be based on OAuth 2.0, where DS servers act as resource servers, and make requests to resolve OAuth 2.0 tokens. Use HTTPS to protect the connections to OAuth 2.0 authorization servers.

Client	Recommendations
Pass-Through Authentication	When DS servers use pass-through authentication, they connect to remote LDAP directory servers for authentication. Use LDAP with StartTLS or LDAPS to protect the connections.
Proxy Requests	A DS directory proxy server can connect to remote directory servers with a bind DN and bind password. Use LDAP with StartTLS or LDAPS to protect requests to remote directory servers.
Replication	Configure replication to use secure connections.
REST to LDAP gateway	The DS REST to LDAP gateway connects to remote LDAP directory servers. Use LDAP with StartTLS or LDAPS to protect the connections.
SMTP account notification and alert handlers	DS servers can send email account notifications and alerts. Use TLS to protect the connections.

Require HTTPS

You can configure an HTTPS port, and no HTTP port, at setup time, or later, as described below. For details on configuring the DS gateway applications to use HTTPS, see your web container documentation.

Set the HTTPS Port

At setup time use the `--httpsPort` option.

Later, follow these steps to set up an HTTPS port:

1. Create an HTTPS connection handler.

   The following example sets the port to `8443` and uses the default server certificate:

```
$ dsconfig \
 create-connection-handler \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name HTTPS \
 --type http \
 --set enabled:true \
 --set listen-port:8443 \
 --set use-ssl:true \
 --set key-manager-provider:PKCS12 \
 --set trust-manager-provider:"JVM Trust Manager" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

If the key manager provider has multiple key pairs that DS could use for TLS, where the secret key was generated with the same key algorithm, such as `EC` or `RSA`, you can specify which key pairs to use with the `--set ssl-cert-nickname:`*`server-cert`* option. The *server-cert* is the certificate alias of the key pair. This option is not necessary if there is only one server key pair, or if each secret key was generated with a different key algorithm.

2. Enable the HTTP access log.

   a. The following command enables JSON-based HTTP access logging:

   ```
   $ dsconfig \
    set-log-publisher-prop \
    --hostname localhost \
    --port 4444 \
    --bindDN uid=admin \
    --bindPassword password \
    --publisher-name "Json File-Based HTTP Access Logger"
   \
    --set enabled:true \
    --no-prompt \
    --usePkcs12TrustStore /path/to/opendj/config/keystore
   \
    --trustStorePassword:file
   /path/to/opendj/config/keystore.pin
   ```

   b. The following command enables HTTP access logging:

   ```
   $ dsconfig \
    set-log-publisher-prop \
    --hostname localhost \
    --port 4444 \
    --bindDN uid=admin \
   ```

```
 --bindPassword password \
 --publisher-name "File-Based HTTP Access Logger" \
 --set enabled:true \
 --no-prompt \
 --usePkcs12TrustStore /path/to/opendj/config/keystore
\
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin
```

3. If the deployment requires SSL client authentication, set the properties `ssl-client-auth-policy` and `trust-manager-provider` appropriately.
4. After you set up an HTTPS port, enable an HTTP endpoint.

   For details, see Configure HTTP User APIs, or Use Administrative APIs.

Require LDAPS

You can configure an LDAPS port, and no LDAP port, at setup time, or later, as described below.

Set the LDAPS port

At setup time, use the `--ldapsPort` option.

Later, follow these steps to set up an LDAPS port:

1. Configure the server to activate LDAPS access:

```
$ dsconfig \
 set-connection-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name LDAPS \
 --set enabled:true \
 --set listen-port:1636 \
 --set use-ssl:true \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

2. If the deployment requires SSL client authentication, set the `ssl-client-auth-policy` and `trust-manager-provider` properties appropriately.

Disable LDAP

Configure the server to disable insecure LDAP access:

```
$ dsconfig \
 set-connection-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name LDAP \
 --set enabled:false \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt
```

Message-Level Security

Server protocols like LDAP, HTTP, JMX, and replication rely on transport layer security to protect connections. Configure connection handlers for TLS or StartTLS, and use access control to enforce secure communications.

- For directory servers, see `ssf` in ACI Subjects. Set the security strength factor to achieve a balance; for example, with 128 or 256. If set too low, the server and client can negotiate a connection that is not secure. If set too high, some clients might not be able to connect.
- For directory proxy servers, use a `connection-minimum-ssf` setting that enforces use of transport layer security, such as 128 or 256.

When negotiating connection security, the server and client must use a common security protocol and cipher suite. To update the security protocols and cipher suites, see TLS Settings.

Transport Layer Security

When a client and server set up an HTTPS or LDAPS connection, they use a protocol to establish the security. They then encapsulate HTTP or LDAP messages in the secure protocol. The process depends on digital certificates. The process uses client certificates differently for mutual authentication and for SASL EXTERNAL binds.

A connection that uses TLS, a protocol based on the SSL protocol, is a connection that is private and reliable. Communications on the connection are kept private by being encrypted with a symmetric key for the session that only the client and server know. Communications are reliable because their integrity is checked with a message authentication code (MAC).

The TLS protocol is independent of the application protocol. TLS encapsulates application level protocols like HTTP and LDAP. The client and server negotiate the secure connection before any messages are sent using the application protocol.

When a client and server set up a secure connection, they negotiate a session with the TLS handshake protocol. During the handshake the server and client use asymmetric keys, their public key certificates and associated private keys, to authenticate (prove their identities).

The default configuration for DS servers sends the server certificate during the handshake. The client is not required to send its certificate. This lets the client authenticate the server when negotiating security. It does not let the server authenticate the client at this stage. This avoids requiring clients, such as browsers, to manage keys and certificate signatures.

For client applications that are part of the software infrastructure, rather than end user applications, managing keys and certificate signatures can be a better choice than managing passwords. Such clients present their certificates during the handshake, letting the server authenticate the client. When both the server and client present certificates during the handshake, this is known as *mutual authentication*.

In TLS v1.2, for example, a successful initial handshake has the client and server do the following:

1. Exchange supported algorithms and random values.
2. Exchange cryptographic information to agree on an initial secret.
3. Exchange certificates and cryptographic information to permit authentication.
4. Generate a symmetric key for the sessions with the initial secret and the random values exchanged.
5. Set the security parameters for the session.
6. Verify that each party uses the same security parameters, and that the handshake was not tampered with.

The handshake completes before any HTTP or LDAP messages are sent over the connection. HTTP authentications and LDAP binds happen after the secure connection has been established.

DS support for TLS relies on the Java implementation. DS support for LDAP authentication is part of the DS server. This is an important distinction:

- When a client application presents its certificate for TLS mutual authentication, the JVM checks the certificate independently of the client application's directory entry.

  When securing the transport layer with mutual authentication, the client certificate must be trusted.

- When the client application binds to the DS server with its certificate, the server checks that the certificate presented matches the certificate stored in the client's directory entry. A certificate mapper finds the directory entry based on the client certificate.

Client Certificate Validation

A first step in establishing a secure connection involves validating the certificate presented by the other party. Part of this is trusting the certificate. The certificate identifies the client or server and the signing certificate. The validating party checks that the other party corresponds to the one identified by the certificate, and checks that the signature can be trusted. If the signature is valid, and the signing certificate is trusted, then the certificate can be trusted.

Certificates can be revoked after they are signed. Therefore, the validation process can involve checking whether the certificate is still valid. This can be done with the Online Certificate Status Protocol (OCSP) or Certificate Revocation Lists (CRLs). OCSP is a newer solution that provides an online service to handle the revocation check for a specific certificate. CRLs are potentially large lists of user certificates that are no longer valid or that are on hold. A CRL is signed by the CA. The validating party obtains the CRL and checks that the certificate being validated is not listed. For a brief comparison, see OCSP: Comparison to CRLs. A certificate can include links to contact the OCSP responder or to the CRL distribution point. The validating party can use these links to check whether the certificate is still valid.

In both cases, the CA who signed the certificate acts as the OCSP responder or publishes the CRLs. When establishing a secure connection with a client application, the server relies on the CA for OCSP and CRLs. This is the case even when the DS server is the repository for the CRLs.

DS directory services are logical repositories for certificates and CRLs. For example, DS servers can store CRLs in a `certificateRevocationList` attribute:

```
dn: cn=My CA,dc=example,dc=com
objectClass: top
objectClass: applicationProcess
objectClass: certificationAuthority
cn: My CA
authorityRevocationList;binary: Base64-encoded ARL
```

```
cACertificate;binary:: Base64-encoded CA certificate
certificateRevocationList;binary:: Base64-encoded CRL
```

Replicate the CRL for high availability. (The ARL in the entry is like a CRL, but for CA certificates.)

Despite being a repository for CRLs, the DS server does check client certificates with its CRLs directly. Instead, when negotiating a secure connection, the server depends on the JVM security configuration. The JVM configuration governs whether validation uses OCSP, CRLs, or both. The JVM relies on system properties that define whether to use the CRL distribution points defined in certificates, and how to handle OCSP requests. These system properties can be set system-wide in `$JAVA_HOME/lib/security/java.security`. The JVM handles revocation checking without the DS server's involvement. For details, see *Support for the CRL Distribution Points Extension*, and *Appendix C: On-Line Certificate Status Protocol (OCSP) Support* in the Java PKI Programmer's Guide.

After a connection is negotiated, the client can bind with its certificate using SASL EXTERNAL authentication. For details, see Certificate-Based Authentication.

OCSP and obtaining CRLs depend on network access to the CA. If DS servers or the DSML or REST to LDAP gateways run on a network where the CA is not accessible, and the deployment requires OSCP or checking CRLs for client application certificates, then you must provide some alternative means to handle OCSP or CRL requests. Configure the JVM to use a locally available OCSP responder, for example. If the solution depends on CRLs, regularly update the CRLs in the directory with downloaded copies of the CA CRLs.

Restrict Client Access

Use a server's global configuration properties, to restrict how clients access the server. These global configuration settings are per server, and are not replicated:

bind-with-dn-requires-password

Whether the server rejects simple bind requests containing a DN but no password.

Default: `true`

To change this setting use the following command:

```
$ dsconfig \
 set-global-configuration-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --set bind-with-dn-requires-password:false \
 --no-prompt \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin
```

max-allowed-client-connections

Restricts the number of concurrent client connections to this server.

Default: 0, meaning no limit is set.

To set a limit of 64K use the following command:

```
$ dsconfig \
 set-global-configuration-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --set max-allowed-client-connections:65536 \
 --no-prompt \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin
```

allowed-client

Restrict which clients can connect to the server.

restricted-client

Restrict the number of concurrent connections per client.

unauthenticated-requests-policy

This setting can take the following values:

`reject`

Reject requests (other than bind or StartTLS requests) received from a client:

- Who has not yet authenticated.
- Whose last authentication was unsuccessful.
- Whose last authentication attempt used anonymous authentication.

`allow-discovery`

Like `reject`, but allows unauthenticated base object searches of the root DSE.

This setting supports applications that read the root DSE to discover server capabilities, and applications that target the root DSE for keep-alive heartbeats.

`allow` (default)

Allow all unauthenticated requests, subject to privileges and access control.

Although this is the default setting, all setup profiles except `ds-evaluation` use `allow-discovery`.

To allow anonymous binds, use the following command:

```
$ dsconfig \
 set-global-configuration-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --set unauthenticated-requests-policy:allow \
 --no-prompt \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin
```

return-bind-error-messages

Does not restrict access, but prevents a server from returning extra information about why a bind failed, as that information could be used by an attacker. Instead, the information is written to the server errors log.

Default: `false`.

To have the server return additional information about why a bind failed, use the following command:

```
$ dsconfig \
 set-global-configuration-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --set return-bind-error-messages:true \
 --no-prompt \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin
```

TLS Settings

To negotiate a secure connection, the server and client must agree on a common protocol and cipher suite. Otherwise, they fail to establish a secure connection.

By default, DS servers use only security protocols and cipher suites considered secure at the time of release. DS servers do, however, support all the security protocols and cipher suites provided by the JVM. For details, see the documentation for the JVM, such as the JDK Providers Documentation for the *The SunJSSE Provider*.

Researchers continue to find vulnerabilities in protocols and cipher suites. If a server supports vulnerable protocols or cipher suites, clients can use them. Attackers can then exploit the vulnerabilities. You might therefore need to restrict the list of accepted protocols and cipher suites at any time.

List Protocols and Cipher Suites

1. To list the protocols and cipher suites that DS servers accept, read the root DSE attributes, `supportedTLSProtocols and supportedTLSCiphers:`

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --baseDN "" \
 --searchScope base \
 "(objectclass=*)" \
 supportedTLSCiphers supportedTLSProtocols
```

A `supportedTLSCiphers` name, such as `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384` for use with TLSv1.2, identifies the key attributes of the cipher suite:

TLS
Specifies the protocol, in this case TLS.

ECDHE_RSA
Specifies the key exchange algorithm used to determine how the client and server authenticate during the handshake phase.

The algorithm in this example uses an elliptic curve variant of the Diffie-Hellman key exchange. In `ECDHE_RSA`, the server signs the ephemeral ECDH public key in the `ServerKeyExchange` message with its RSA private key.

Ideally the server certificate would have a `KeyUsage` extension with only the `digitalSignature` bit set to prevent it being used for encryption.

WITH_AES_256_GCM

Specifies the bulk encryption algorithm, including the key size or initialization vectors.

This example specifies the Advanced Encryption Standard (AES) with 256-bit key size and Galois/Counter Mode (GCM) block cipher mode.

SHA384

Specifies the message authentication code algorithm used to create the message digest, which is a cryptographic hash of each block in the message stream.

In this example, the SHA-2 hash function, SHA-384, is used.

A `supportedTLSProtocols` name identifies the protocol and version, such as `TLSv1.2` or `TLSv1.3`.

Cipher suites compatible with TLSv1.3 do not include the key exchange algorithm. In TLSv1.3, the signature algorithm and key exchange are negotiated separately.

Restrict Protocols and Cipher Suites

You can limit the protocols and cipher suites that DS servers accept by setting the properties, `ssl-protocol` and `ssl-cipher-suite` on the appropriate components.

This following settings derive from the server-side TLS recommendations published by the Mozilla Operations Security team at the time of this writing. Recommendations evolve. Make sure you use current recommendations when configuring security settings:

1. For each cipher suite key algorithm to support, create a key pair using the supported key algorithm.

   The following example adds a key pair to the default PKCS#12 keystore:

   ```
   $ keytool \
    -genkeypair \
    -alias ssl-key-pair-ec \
    -keyalg EC \
    -ext "san=dns:ds.example.com" \
    -dname "CN=ds.example.com,O=Example Corp,C=FR" \
    -keystore /path/to/opendj/config/keystore \
    -storetype PKCS12 \
    -storepass:file /path/to/opendj/config/keystore.pin \
    -keypass:file /path/to/opendj/config/keystore.pin
   ```

2. On the components you use, explicitly set the supported protocols and cipher suites.

   The following example adjusts settings for the LDAP and LDAPS connection handlers:

```
$ dsconfig \
 set-connection-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name LDAP \
 --add ssl-protocol:TLSv1.3 \
 --add ssl-cipher-suite:TLS_AES_128_GCM_SHA256 \
 --add ssl-cipher-suite:TLS_AES_256_GCM_SHA384 \
 --no-prompt \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin

$ dsconfig \
 set-connection-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name LDAPS \
 --add ssl-protocol:TLSv1.3 \
 --add ssl-cipher-suite:TLS_AES_128_GCM_SHA256 \
 --add ssl-cipher-suite:TLS_AES_256_GCM_SHA384 \
 --no-prompt \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin
```

3. Use the appropriate settings for your connection handlers:

```
$ dsconfig \
 set-connection-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name LDAP \
 --set enabled:true \
 --set listen-port:1389 \
 --set allow-start-tls:true \
 --set key-manager-provider:PKCS12 \
 --set trust-manager-provider:PKCS12 \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt

$ dsconfig \
 set-connection-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name LDAPS \
 --set listen-port:1636 \
 --set enabled:true \
 --set use-ssl:true \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

Restrict Protocols For Command-Line Tools

You can specify which protocol versions to allow when command-line tools negotiate secure connections with LDAP servers.

The command-line tools depend on a system property, `org.opends.ldaps.protocols`. This property takes a comma-separated list of protocols. The default is constructed from the list of all protocols the JVM supports, removing protocol names starting with `SSL`. For example, if support is enabled in the JVM for versions 1.0, 1.1, and 1.2 of the TLS protocol, then the default is `"TLSv1,TLSv1.1,TLSv1.2"`.

1. Restrict the protocols to use by setting the property in one of the following ways:

   a. Set the property by editing the `java-args` for the command in `config/java.properties`.

For example, to restrict the protocol to TLS v1.2 when the **status** command negotiates a secure administrative connection, edit the corresponding line in `config/java.properties`:

```
status.java-args=-Xms8m -client -
Dorg.opends.ldaps.protocols=TLSv1.2
```

b. Set the property at runtime when running the command.

The following example restricts the protocol to TLS v1.2 when the **status** command negotiates a secure administrative connection:

```
$ export OPENDJ_JAVA_ARGS="-
Dorg.opends.ldaps.protocols=TLSv1.2"

$ status \
 --bindDn uid=admin \
 --bindPassword password \
 --hostname localhost \
 --port 4444 \
 --usePkcs12TrustStore /path/to/opendj/config/keystore
\
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin
```

Authentication Mechanisms

Authentication is the process of verifying who is requesting access to a resource. The user or application making the request presents credentials, making it possible to prove that the requester is who they claim to be. The goal is to authorize access to directory resources depending on the confirmed identity of the user or application making the request.

LDAP is a stateful protocol, where the client sets up and maintains a connection with the server, potentially performing many operations or long-lived operations before disconnecting from the LDAP server. One of the LDAP operations, a *bind*, authenticates the client to the server. A bind is the first operation that a client performs after setting up a connection. Clients can bind again on the same connection to reauthenticate.

At the transport layer, DS servers support SSL and TLS protocols with mutual client authentication. This level of authentication is useful to properly secure connections. The authentication at this level is handled by the underlying JVM. The client identity verified at this level is not available to the DS server for the purpose of fine-grained authorization. For fine-grained authorization, you need LDAP authentication.

DS servers support multiple authentication mechanisms for LDAP operations:

Simple bind (name/password) authentication

The client application presents a bind DN/password combination. The server checks that the password matches the password on the entry with the specified bind DN.

This mechanism involves sending the credentials over the network. Always use secure connections at the transport layer when you expect simple LDAP binds. You can configure either or both LDAPS and StartTLS, depending on what the client applications support.

For additional information, see Simple Binds.

Anonymous authentication

Simple bind authentication without credentials.

Anonymous authentication lets the server make authorization decisions when the client identity is unknown.

Allow anonymous authentication for publicly readable resources, such as root DSE attributes. Configure access control to let anonymous and other users read them.

For additional information, see Anonymous Access.

SASL authentication

Simple Authentication and Security Layer (SASL) is a framework, rather than a single method. DS servers provide handlers for a number of SASL mechanisms, including strong authentication choices like the External SASL mechanism handler for certificate-based authentication, and the GSSAPI SASL mechanism handler for use with Kerberos v5 systems.

Certificate-based authentication is well-suited for applications where distributing keys is easier than protecting passwords. For additional information, see Certificate-Based Authentication.

GSSAPI-based authentication is useful for interoperation with Kerberos. For additional information, see Authenticate With Kerberos.

Authentication with proxied authorization

The client application binds with its credentials, and uses proxied authorization to perform operations as another user.

Client applications can use another means to authenticate the user before requesting proxied authorization. For details, see Proxied Authorization.

Pass-through authentication to another LDAP directory

The client application binds with its credentials, and another LDAP directory service handles the authentication. This is known as *pass-through authentication*.

Pass-through authentication is useful when credentials are stored in a remote directory service, and the DS directory service stores part of the user profile. For details, see Pass-Through Authentication.

DS servers and DS REST to LDAP gateway support multiple HTTP authentication mechanisms.

The identity from the HTTP request is mapped to an LDAP account for use in authorization decisions, so the mechanisms are known as authorization mechanisms. Their configuration is described in Configure HTTP Authorization. The following authorization mechanisms are available:

HTTP Basic authorization

The client application sends an HTTP request that uses HTTP Basic authentication.

The client application can alternatively send an HTTP request with username and password headers.

You configure DS software to map the HTTP username to an LDAP DN. The result is like a simple name/password bind.

This mechanism involves sending the credentials over the network. Always use secure connections at the transport layer when you allow HTTP Basic.

Anonymous authorization

The client application sends an HTTP request without authenticating.

You configure DS software either to map the HTTP request to anonymous authentication at the LDAP level, or to bind at the LDAP level as a specific user.

OAuth 2.0 authorization

The client application sends an HTTP request bearing an OAuth 2.0 access token that includes at least one scope whose value makes it possible to determine the user identity.

You configure DS software to resolve the access token, and to map the user identity from the scope to an LDAP account.

This mechanism involves sending the credentials over the network. Always use secure connections at the transport layer for OAuth 2.0 authorization.

Anonymous Access

In LDAP, an *anonymous bind* is a bind operation using simple authentication with an empty DN and an empty password. DS servers apply access controls (ACIs) to let anonymous clients access only fully public information. Examples include information about the directory server in the root DSE, and LDAP schema definitions.

By default, DS servers disable anonymous access to directory data. ACIs take a user DN subject, `ldap:///anyone`, that matches anonymous and authenticated users.

When a client accesses the directory over HTTP, anonymous operations can be mapped either to a specific user identity or to an anonymous user (default). This is set using the HTTP Anonymous authorization mechanism for the HTTP endpoint.

Simple Binds

In LDAP, a *simple bind* is name/password authentication. The client application presents a bind DN/password combination. The DS server checks that the password matches the password on the entry with the specified bind DN.

The LDAP connection transport layer must be secure for a simple bind. Otherwise, eavesdroppers can read the credentials.

DS servers provide two alternatives to secure the connection for a simple bind, both of which depend on certificates and public key infrastructure:

LDAPS
: To support LDAP over SSL and TLS, DS servers have a separate connection handler that listens for traffic on a port dedicated to secure connections.

LDAP with StartTLS
: DS servers support using the StartTLS extended operation on an insecure LDAP port. With StartTLS, the client initiates the connection on the LDAP port, and then negotiates a secure connection.

Pass-Through Authentication

Pass-Through Authentication (PTA), a remote LDAP service to determine the response to an authentication request. A typical use case for PTA involves passing authentication through to Active Directory for users coming from Microsoft Windows systems.

About PTA

You use PTA when the credentials for authenticating are stored in a remote directory service. In effect, the DS server redirects the bind operation against a remote LDAP server.

The method a server uses to redirect the bind depends on the mapping from the user entry in the DS server to the corresponding user entry in the remote directory. DS servers provide you several choices to set up the mapping:

- When both the local entry in the DS server and the remote entry in the other server have the same DN, you do not have to set up the mapping. By default, the DS server redirects the bind with the original DN and password from the client application.
- When the local entry in the DS server has an attribute holding the DN of the remote entry, you can specify which attribute holds the DN. The DS server redirects the bind on the remote server using the DN value.
- When you cannot get the remote bind DN directly, you need an attribute and value on the DS entry that corresponds to an identical attribute and value on the remote server. In this case, you also need the bind credentials for a user who can search for the entry on the remote server. The DS server performs a search for the entry using the matching attribute and value, and then redirects the bind with the DN from the remote entry.

You configure PTA as an authentication policy that you associate with a user's entry in the same way that you associate a password policy with a user's entry. Either a user has an authentication policy for PTA, or the user has a local password policy.

Set Up PTA

When setting up PTA, you need to know define:

- Which remote server(s) to redirect binds to
- How you map user entries in the DS server to user entries in the remote directory

Secure Connections

When performing PTA, you protect communications between the DS server and the authenticating server. When you test secure connections with a CA that is not well-known, make sure the authentication server's certificate is trusted by the DS server.

If the authentication server's CA is well-known or already trusted by the DS server, you can skip these steps:

1. Export the CA or server certificate from the authentication server.

   How you perform this step depends on the authentication directory server.
2. Record the hostname used in the certificate.

   You use the hostname when configuring the secure connection.
3. Import the trusted authentication server certificate into the DS server's keystore.

Configure a PTA Policy

Configure authentication policies with the **`dsconfig`** command. Notice that authentication policies are part of the server configuration, and therefore not replicated:

1. Set up an authentication policy for PTA to the authentication server:

   ```
   $ dsconfig \
    create-password-policy \
    --hostname localhost \
    --port 4444 \
    --bindDN uid=admin \
    --bindPassword password \
    --policy-name "PTA Policy" \
    --type ldap-pass-through \
    --set primary-remote-ldap-server:remote-
   server.example.com:1636 \
    --set mapped-attribute:uid \
    --set mapped-search-base-dn:"dc=example,dc=com" \
    --set mapping-policy:mapped-search \
    --set use-ssl:true \
    --usePkcs12TrustStore /path/to/opendj/config/keystore \
    --trustStorePassword:file
   /path/to/opendj/config/keystore.pin \
    --no-prompt
   ```

   The policy shown here maps identities with this password policy to identities under `dc=example,dc=com` on the authentication server. Users must have the same `uid` values on both servers. This policy uses LDAPS between the DS server and the authentication server.

2. Check that your policy has been added to the list:

   ```
   $ dsconfig \
    list-password-policies \
    --hostname localhost \
    --port 4444 \
    --bindDN uid=admin \
    --bindPassword password \
    --property use-ssl \
    --usePkcs12TrustStore /path/to/opendj/config/keystore \
    --trustStorePassword:file
   /path/to/opendj/config/keystore.pin \
    --no-prompt
   ```

```
Password Policy         : Type               : use-ssl
------------------------:--------------------:--------
Default Password Policy : password-policy    : -
PTA Policy              : ldap-pass-through  : true
Root Password Policy    : password-policy    : -
```

Use PTA To Active Directory

The steps below demonstrate how to set up PTA to Active Directory. The information that follows will help you make sense of the steps.

Entries on the DS side use `uid` as the naming attribute, and entries also have `cn` attributes. Active Directory entries use `cn` as the naming attribute. User entries on both sides share the same `cn` values. The mapping between entries therefore uses `cn`.

Consider a deployment where the DS account with `cn=LDAP PTA User` and DN `uid=ldapptauser,ou=People,dc=example,dc=com` corresponds to an Active Directory account with DN `CN=LDAP PTA User,CN=Users,DC=internal,DC=forgerock,DC=com`. The steps below enable the user with `cn=LDAP PTA User` on the DS server to authenticate through Active Directory:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
 --bindPassword bribery
 --baseDN dc=example,dc=com \
 uid=ldapptauser \
 cn

dn: uid=ldapptauser,ou=People,dc=example,dc=com
cn: LDAP PTA User$ ldapsearch \
 --hostname ad.example.com \
 --port 636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --baseDN "CN=Users,DC=internal,DC=forgerock,DC=com"  \
 --bindDN
"cn=administrator,cn=Users,DC=internal,DC=forgerock,DC=com"  \
 --bindPassword password \
 "(cn=LDAP PTA User)" \
 cn

dn: CN=LDAP PTA User,CN=Users,DC=internal,DC=forgerock,DC=com
cn: LDAP PTA User
```

The DS server must map the `uid=ldapptauser,ou=People,dc=example,dc=com` entry to the Active Directory `CN=LDAP PTA User,CN=Users,DC=internal,DC=forgerock,DC=com` entry. In order to do the mapping, the DS server must search for the user in Active Directory, using the `cn` value that it recovers from its own entry for the user. Active Directory does not allow anonymous searches, so part of the authentication policy configuration consists of the administrator DN and password the DS server uses to bind to Active Directory to search.

Finally, before setting up the PTA policy, make sure the DS server can connect to Active Directory over a secure connection to avoid sending passwords in the clear.

1. Export the certificate from the Windows server.
   - Select start > All Programs > Administrative Tools > Certification Authority, then right-click the CA and select Properties.
   - In the General tab, select the certificate and select View Certificate.
   - In the Certificate dialog, select the Details tab, then select Copy to File.
   - Use the Certificate Export Wizard to export the certificate to a file, such as `windows.cer` .
2. Copy the exported certificate to the system running the DS server.
3. Import the server certificate into the DS keystore:

   ```
   $ keytool \
    -importcert \
    -alias ad-cert \
    -keystore /path/to/opendj/config/keystore \
    -storepass:file /path/to/opendj/config/keystore.pin \
    -storetype PKCS12 \
    -file ~/Downloads/windows.cer \
    -noprompt

   Certificate was added to keystore
   ```

   At this point, the DS server can connect securely to Active Directory.
4. Set up an authentication policy for DS users to authenticate to Active Directory:

```
# Create a trust manager provider to access the Active
Directory certificate:
$ dsconfig \
 create-trust-manager-provider \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --provider-name PKCS12 \
 --type file-based \
 --set enabled:true \
 --set trust-store-type:PKCS12 \
 --set trust-store-file:config/keystore \
 --set trust-store-pin:"&{file:config/keystore.pin}" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt

# Use the trust manager provider in the PTA policy:
$ dsconfig \
 create-password-policy \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --type ldap-pass-through \
 --policy-name "AD PTA Policy" \
 --set primary-remote-ldap-server:ad.example.com:636 \
 --set mapped-attribute:cn \
 --set mapped-search-base-
dn:"CN=Users,DC=internal,DC=forgerock,DC=com" \
 --set mapped-search-bind-
dn:"cn=administrator,cn=Users,DC=internal,DC=forgerock,DC=
com" \
 --set mapped-search-bind-password:password \
 --set mapping-policy:mapped-search \
 --set trust-manager-provider:PKCS12 \
 --set use-ssl:true \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

5. Assign the authentication policy to a test user:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: uid=ldapptauser,ou=People,dc=example,dc=com
changetype: modify
add: ds-pwp-password-policy-dn
ds-pwp-password-policy-dn: cn=AD PTA Policy,cn=Password
Policies,cn=config
EOF
```

6. Check that the user can bind using PTA to Active Directory:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --baseDN dc=example,dc=com \
 --bindDN uid=ldapptauser,ou=People,dc=example,dc=com \
 --bindPassword password \
 "(cn=LDAP PTA User)" \
 userpassword cn

dn: uid=ldapptauser,ou=People,dc=example,dc=com
cn: LDAP PTA User
```

Notice that to complete the search, the user has authenticated with a password to Active Directory. No `userpassword` value is present in the DS service.

Assign PTA Policies

You assign authentication policies in the same way as you assign password policies, by using the `ds-pwp-password-policy-dn` attribute.

NOTE

NOTE

Although you assign the PTA policy using the same attribute as for password policy, the authentication policy is not in fact a password policy. Therefore, the user with a PTA policy does not have the operational attribute `pwdPolicySubentry` .

Assign a PTA Policy To a User

Users depending on PTA no longer need a local password policy, as they no longer authenticate locally.

Examples in the following procedure work for this user, whose entry is as shown below. Notice that the user has no `userPassword` attribute. The user's password on the authentication server is `password` :

```
dn: uid=ptaUser,ou=People,dc=example,dc=com
uid: ptaUser
objectClass: top
objectClass: person
objectClass: organizationalperson
objectClass: inetorgperson
uid: ptaUser
cn: PTA User
sn: User
```

This user's entry on the authentication server has `uid=ptaUser` . The PTA policy performs the mapping to find the user entry in the authentication server:

1. Give an administrator access to update a user's password policy:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: ou=People,dc=example,dc=com
changetype: modify
add: aci
aci: (targetattr = "ds-pwp-password-policy-dn")
 (version 3.0;acl "Allow Kirsten Vaughan to assign
password policies";
```

```
 allow (all) (userdn =
"ldap:///uid=kvaughan,ou=People,dc=example,dc=com";)
EOF
```

2. Update the user's `ds-pwp-password-policy-dn` attribute:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
 --bindPassword bribery << EOF
dn: uid=ptaUser,ou=People,dc=example,dc=com
changetype: modify
replace: ds-pwp-password-policy-dn
ds-pwp-password-policy-dn: cn=PTA Policy,cn=Password
Policies,cn=config
EOF
```

3. Check that the user can authenticate through to the authentication server:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --baseDN dc=example,dc=com \
 --bindDN uid=ptaUser,ou=People,dc=example,dc=com \
 --bindPassword chngthspwd \
 "(uid=ptaUser)" \
 1.1

dn: uid=ptaUser,ou=People,dc=example,dc=com
```

Assign a PTA Policy To a Group

Examples in the following steps use the PTA policy defined previously. The administrator's entry is present on the authentication server:

1. Give an administrator the privilege to write subentries, such as those used for password policies:

   ```
   $ ldapmodify \
    --hostname localhost \
    --port 1636 \
    --useSsl \
    --usePkcs12TrustStore /path/to/opendj/config/keystore \
    --trustStorePassword:file
   /path/to/opendj/config/keystore.pin \
    --bindDN uid=admin \
    --bindPassword password << EOF
   dn: uid=kvaughan,ou=People,dc=example,dc=com
   changetype: modify
   add: ds-privilege-name
   ds-privilege-name: subentry-write
   EOF
   ```

   Notice here that the directory superuser, `uid=admin`, assigns privileges. Any administrator with the `privilege-change` privilege can assign privileges. However, if the administrator can update administrator privileges, they can assign themselves the `bypass-acl` privilege. Then they are no longer bound by access control instructions, including both user data ACIs and global ACIs. For this reason, do not assign the `privilege-change` privilege to normal administrator users.

2. Create a subentry for a collective attribute that sets the `ds-pwp-password-policy-dn` attribute for group members' entries:

   ```
   $ ldapmodify \
    --hostname localhost \
    --port 1636 \
    --useSsl \
    --usePkcs12TrustStore /path/to/opendj/config/keystore \
    --trustStorePassword:file
   /path/to/opendj/config/keystore.pin \
    --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
    --bindPassword bribery << EOF
   dn: cn=PTA Policy for Dir Admins,dc=example,dc=com
   objectClass: collectiveAttributeSubentry
   objectClass: extensibleObject
   objectClass: subentry
   objectClass: top
   cn: PTA Policy for Dir Admins
   ```

```
ds-pwp-password-policy-dn;collective: cn=PTA
Policy,cn=Password Policies,cn=config
subtreeSpecification: { base "ou=People",
specificationFilter
  "(isMemberOf=cn=Directory
Administrators,ou=Groups,dc=example,dc=com)"}
EOF
```

The `base` entry identifies the branch that holds administrator entries.

3. Check that the DS server has applied the policy.
   - Make sure you can bind as the user on the authentication server:

     ```
     $ ldapsearch \
      --hostname remote-server.example.com \
      --port 1636 \
      --useSsl \
      --usePkcs12TrustStore /path/to/opendj/config/keystore
     \
      --trustStorePassword:file
     /path/to/opendj/config/keystore.pin \
      --bindDN "uid=kvaughan,ou=People,dc=example,dc=com" \
      --bindPassword bribery \
      --baseDN "dc=example,dc=com" \
      "(uid=kvaughan)" \
      1.1

     dn: uid=kvaughan,ou=People,dc=example,dc=com
     ```

   - Check that the user can authenticate through to the authentication server from the DS server:

     ```
     $ ldapsearch \
      --hostname localhost \
      --port 1636 \
      --useSsl \
      --usePkcs12TrustStore /path/to/opendj/config/keystore
     \
      --trustStorePassword:file
     /path/to/opendj/config/keystore.pin \
      --bindDN "uid=kvaughan,ou=people,dc=example,dc=com" \
      --bindPassword bribery \
      --baseDN dc=example,dc=com \
      "(uid=kvaughan)" \
      1.1
     ```

```
dn: uid=kvaughan,ou=People,dc=example,dc=com
```

Certificate-Based Authentication

One alternative to simple binds with user name/password combinations consists of storing a digital certificate on the LDAP entry, and using the certificate as credentials during the bind. You can use this mechanism, for example, to let applications bind without using passwords.

By setting up a secure connection with a certificate, the client is in effect authenticating to the server. The server must close the connection if it cannot trust the client certificate. However, the certificate presented when establishing a secure connection does not authenticate the client. The secure connection is established by the JVM at the transport layer, independently of the LDAP protocol.

When binding with a certificate, the client must request the SASL external mechanism. The DS server maps the certificate to the client's entry in the directory. When it finds a matching entry, and the entry contains a certificate, the DS server can check whether the certificate in the entry matches the certificate from the request. It depends on the `certificate-validation-policy` setting of the SASL external handler. On success, the server sets the authorization identity for the connection, and the bind is successful.

For the whole process of authenticating with a certificate to work smoothly, the DS server and the client must trust each others' certificates, and the DS server must be configured to map the certificate to the client entry.

Add Certificate Attributes to the Client Entry

Before a client tries to bind to DS servers using a certificate, create a certificate, and add appropriate certificate attributes to the client's entry.

The `ds-evaluation` setup profile includes the entry, `cn=My App,ou=Apps,dc=example,dc=com`, used in these examples. The client key store password is stored in a `MY_KEYSTORE_PIN` environment variable:

1. Create a certificate using the DN of the client entry as the subject DN.

   This example uses the **`dskeymgr`** command to generate a key pair. The certificate is signed by the private CA based on the deployment key used when setting up DS servers. Servers set up with the same deployment key trust the CA, and so can trust the client's certificate:

```
$ dskeymgr \
 create-tls-key-pair \
 --deploymentKey $DEPLOYMENT_KEY \
 --deploymentKeyPassword password \
 --alias myapp-cert \
 --subjectDn "cn=My App,ou=Apps,dc=example,dc=com" \
 --keyStoreFile /path/to/opendj/my-keystore \
 --keyStorePassword $MY_KEYSTORE_PIN
```

For more command options, refer to dskeymgr. The default validity for the certificate is one year.

2. Make note of the certificate SHA-256 fingerprint.

   Later in this procedure you update the client application entry with the SHA-256 fingerprint, referred to henceforth as `SHA265_FINGERPRINT`:

```
$ keytool \
 -list \
 -v \
 -alias myapp-cert \
 -keystore /path/to/opendj/my-keystore \
 -storepass $MY_KEYSTORE_PIN | awk '/SHA256:/{print $2}'

SHA256_FINGERPRINT
```

3. Modify the entry to add attributes related to the certificate.

   For example, add the client certificate fingerprint on the `ds-certificate-fingerprint` attribute. This example uses the SHA-256 fingerprint, which is the default for the fingerprint certificate mapper.

   To require that the certificate is issued by a known CA, use the `ds-certificate-issuer-dn` attribute. Use this to verify the certificate issuer whenever multiple CAs are trusted in order to prevent impersonation. Different CAs can issue certificates with the same subject DN, but not with the same issuer DN. You must also specify the issuer attribute in the certificate mapper configuration, as shown below.

   To map the certificate subject DN to an attribute of the entry, use the `ds-certificate-subject-dn` attribute.

   The following entry demonstrates all these attributes. Save the entry in a file `addcert.ldif` in order to edit the fingerprint:

```
dn: cn=My App,ou=Apps,dc=example,dc=com
changetype: modify
add: objectclass
objectclass: ds-certificate-user
-
add: ds-certificate-fingerprint
ds-certificate-fingerprint: SHA256_FINGERPRINT
-
add: ds-certificate-issuer-dn
ds-certificate-issuer-dn: CN=Deployment
key,O=ForgeRock.com
-
add: ds-certificate-subject-dn
ds-certificate-subject-dn: CN=My App, OU=Apps, DC=example,
DC=com
```

Replace the certificate fingerprint with the actual fingerprint before adding the certificate:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=kvaughan,ou=People,dc=example,dc=com \
 --bindPassword bribery \
 addcert.ldif
```

4. Check your work:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --baseDN dc=example,dc=com \
 "(cn=My App)"

dn: cn=My App,ou=Apps,dc=example,dc=com
ds-certificate-fingerprint: SHA256_FINGERPRINT
```

```
objectClass: person
objectClass: inetOrgPerson
objectClass: organizationalPerson
objectClass: ds-certificate-user
objectClass: top
ds-certificate-issuer-dn: CN=My App, OU=Apps, DC=example,
DC=com
ds-certificate-subject-dn: CN=My App, OU=Apps, DC=example,
DC=com
cn: My App
sn: App
```

Trust a Third-Party Client Certificate

TIP

If you control the client application, use your private CA to sign its certificate. This can be done as demonstrated in Add Certificate Attributes to the Client Entry. You can then skip this procedure.

To trust the client certificate, a trust manager must be able to trust the signing certificate (issuer). If the client certificate is self-signed or signed by an unknown CA, you must update the server's truststore:

1. Export the self-signed certificate or the CA certificate:

   ```
   $ keytool \
    -export \
    -alias myapp-cert \
    -keystore /path/to/opendj/my-keystore \
    -storepass:env MY_KEYSTORE_PIN \
    -keypass:env MY_KEYSTORE_PIN \
    -file /path/to/opendj/myapp-cert.crt

   Certificate stored in file </path/to/opendj/myapp-
   cert.crt>
   ```

   The command is similar for a CA certificate.

2. Import the exported, trusted certificate into a server truststore.

   The following examples use the default server keystore and PIN:

   a. The following example imports the self-signed certificate exported in Step 1:

```
$ keytool \
 -import \
 -alias myapp-cert \
 -file /path/to/opendj/myapp-cert.crt \
 -keystore /path/to/opendj/config/keystore \
 -storetype PKCS12 \
 -storepass:file /path/to/opendj/config/keystore.pin \
 -noprompt

Certificate was added to keystore
```

b. The following example imports an exported CA certificate in a file called `ca.crt` :

```
$ keytool \
 -import \
 -alias ca-cert \
 -file ca.crt \
 -keystore /path/to/opendj/config/keystore \
 -storetype PKCS12 \
 -storepass:file /path/to/opendj/config/keystore.pin \
 -noprompt

Certificate was added to keystore
```

Configure Certificate Mappers

DS servers use certificate mappers during binds to establish a mapping between a client certificate and the entry with the certificate. DS servers have the following certificate mappers:

Fingerprint Certificate Mapper

Looks for the certificate fingerprint in an attribute of the entry (default: `ds-certificate-fingerprint`).

Subject Attribute To User Attribute Mapper

Looks for a match between an attribute of the certificate subject and an attribute of the entry (default: match `cn` in the certificate to `cn` on the entry, or match `emailAddress` in the certificate to `mail` on the entry).

Subject DN to User Attribute Certificate Mapper

Looks for the certificate subject DN in an attribute of the entry (default: `ds-certificate-subject-dn`).

Subject Equals DN Certificate Mapper

Looks for an entry whose DN matches the certificate subject DN.

The following steps demonstrate how to use the Fingerprint Mapper default algorithm of SHA-256:

1. List the certificate mappers to retrieve the correct name:

```
$ dsconfig \
 list-certificate-mappers \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt

Certificate Mapper                   : Type
: enabled
------------------------------------:---------------------
-----------------:--------
Fingerprint Mapper                   : fingerprint
: true
Subject Attribute to User Attribute : subject-attribute-
to-user-attribute : true
Subject DN to User Attribute         : subject-dn-to-user-
attribute        : true
Subject Equals DN                    : subject-equals-dn
: true
```

2. Examine the current configuration:

```
$ dsconfig \
 get-certificate-mapper-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --mapper-name "Fingerprint Mapper" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt

Property               : Value(s)
----------------------:------------------------------------
---------------------
enabled                : true
fingerprint-algorithm : sha256
fingerprint-attribute : ds-certificate-fingerprint
issuer-attribute       : The certificate issuer DN will not
be verified.
user-base-dn           : The server performs the search in
all public naming
                       : contexts.
```

3. Change the configuration as necessary.
4. Set the External SASL Mechanism Handler to use the appropriate certificate mapper (default: Subject Equals DN):

```
$ dsconfig \
 set-sasl-mechanism-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name External \
 --set certificate-mapper:"Fingerprint Mapper" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

Authenticate With the Client Certificate

Instead of providing a bind DN and password as for simple authentication, use the SASL EXTERNAL authentication mechanism, and provide the certificate. As a test with example data, you can try an anonymous search, then try with certificate-based authentication.

Before you try this example, make sure the DS server is set up to accept StartTLS from clients, and that you have set up the client certificate as described above. The password for the client key store is stored in a `MY_KEYSTORE_PIN` environment variable.

Also, if the DS server uses a certificate for StartTLS that was signed by a private CA, reference a truststore containing the CA certificate. In this example, the DS server uses a keystore with the CA certificate, and the client uses the keystore as its truststore.

Notice that the DS server does not allow an anonymous user to modify its description:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin <<
EOF
dn: cn=My App,ou=Apps,dc=example,dc=com
changetype: modify
replace: description
description: New description

EOF

# The LDAP modify request failed: 50 (Insufficient Access Rights)
# Additional Information:  The entry cn=My
App,ou=Apps,dc=example,dc=com cannot be modified due to
insufficient access rights
```

After the client binds successfully, it can modify its description:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --saslOption mech="EXTERNAL" \
 --certNickName myapp-cert \
 --keyStorePath /path/to/opendj/my-keystore \
 --keyStorePassword $MY_KEYSTORE_PIN <<EOF
dn: cn=My App,ou=Apps,dc=example,dc=com
changetype: modify
replace: description
description: New description

EOF

# MODIFY operation successful for DN cn=My
App,ou=Apps,dc=example,dc=com
```

You can also try the same test with other certificate mappers.

This example uses the fingerprint mapper:

```
$ dsconfig \
 set-sasl-mechanism-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name External \
 --set certificate-mapper:"Fingerprint Mapper" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt

$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --saslOption mech="EXTERNAL" \
 --certNickName myapp-cert \
 --keyStorePath /path/to/opendj/my-keystore \
 --keyStorePassword $MY_KEYSTORE_PIN <<EOF
dn: cn=My App,ou=Apps,dc=example,dc=com
changetype: modify
replace: description
description: New description

EOF

# MODIFY operation successful for DN cn=My
App,ou=Apps,dc=example,dc=com
```

This example uses the subject attribute to user attribute mapper:

```
$ dsconfig \
 set-sasl-mechanism-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name External \
 --set certificate-mapper:"Subject Attribute to User Attribute" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt

$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --saslOption mech="EXTERNAL" \
 --certNickName myapp-cert \
 --keyStorePath /path/to/opendj/my-keystore \
 --keyStorePassword $MY_KEYSTORE_PIN <<EOF
dn: cn=My App,ou=Apps,dc=example,dc=com
changetype: modify
replace: description
description: New description

EOF

# MODIFY operation successful for DN cn=My
App,ou=Apps,dc=example,dc=com
```

This example uses the subject DN to user attribute mapper:

```
$ dsconfig \
 set-sasl-mechanism-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name External \
 --set certificate-mapper:"Subject DN to User Attribute" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt

$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --saslOption mech="EXTERNAL" \
 --certNickName myapp-cert \
 --keyStorePath /path/to/opendj/my-keystore \
 --keyStorePassword $MY_KEYSTORE_PIN <<EOF
dn: cn=My App,ou=Apps,dc=example,dc=com
changetype: modify
replace: description
description: New description

EOF

# MODIFY operation successful for DN cn=My
App,ou=Apps,dc=example,dc=com
```

Authenticate With a Third-Party Certificate

When a client application authenticates with a self-signed certificate or a certificate signed by a public CA, the easiest certificate mapper to use is the fingerprint mapper. When using any other certificate mapper, make sure that the client certificate is in the client's entry, and that the SASL EXTERNAL handler has a `certificate-validation-policy:ifpresent` (default), or `certificate-validation-policy:always`. Any client certificate can be used to secure TLS for the connection. However, to trust the certificate for authentication, the server must ensure a unique match between the client's certificate and the client's entry.

A client application creating its own certificate can set subject DN, issuer DN, and other fields as desired, so these cannot be used to established trust. When obtaining a signature from a public CA, the client might set many fields as desired. The issuer DN guarantees only that the CA signed the certificate.

The following example demonstrates safe use of blind trust and fingerprint mapping. This demonstration is also appropriate when clients use certificates signed by public CAs, in which case, you could use the JVM trust manager, for example.

Enable a blind trust manager:

```
$ dsconfig \
 create-trust-manager-provider \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --type blind \
 --provider-name "Blind Trust" \
 --set enabled:true \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt
```

Use SHA-256 as the fingerprint certificate mapper algorithm, which is the default.

Configure the handler for SASL EXTERNAL binds to use the fingerprint mapper, rather than the default subject DN mapper. As in this example, do not enable a more lenient mapper when using blind trust or public trust:

```
$ dsconfig \
 set-sasl-mechanism-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name External \
 --set certificate-mapper:"Fingerprint Mapper" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt
```

After making these configuration changes, enable the trust manager on the appropriate connection handler. The following command enables blind trust on the LDAP connection handler, where the client will use StartTLS. Notice that only blind trust is enabled. If you want to allow blind trust for some applications and private CA trust for others, use a separate connection handler listening on a separate port:

```
$ dsconfig \
 set-connection-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --handler-name LDAP \
 --set trust-manager-provider:"Blind Trust" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt
```

Make sure the client application certificate fingerprints use SHA-256. The following command updates the example client entry to change the fingerprint appropriately:

```
$ SHA256_FINGERPRINT=$(keytool \
 -list \
 -v \
 -alias myapp-cert \
 -keystore /path/to/opendj/my-keystore \
 -storepass $MY_KEYSTORE_PIN | awk '/SHA256:/{print $2}')

$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=kvaughan,ou=People,dc=example,dc=com \
 --bindPassword bribery << EOF
dn: cn=My App,ou=Apps,dc=example,dc=com
changetype: modify
replace: ds-certificate-fingerprint
ds-certificate-fingerprint: $SHA256_FINGERPRINT

EOF
```

When the client binds successfully, it can modify its description. The server JVM checks client certificate validity and proves the client has the private key during the process of setting up TLS. During the SASL EXTERNAL bind, the server verifies that the fingerprint in the client entry matches the certificate:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --saslOption mech="EXTERNAL" \
 --certNickName myapp-cert \
 --keyStorePath /path/to/opendj/my-keystore \
 --keyStorePassword $MY_KEYSTORE_PIN <<EOF
dn: cn=My App,ou=Apps,dc=example,dc=com
changetype: modify
replace: description
description: New description

EOF

# MODIFY operation successful for DN cn=My
App,ou=Apps,dc=example,dc=com
```

For additional verification during the bind, include the client certificate in the client's entry.

Authenticate With Kerberos

Windows, UNIX, and Linux systems support Kerberos v5 authentication, which can operate safely on an open, unprotected network. In Kerberos authentication, the client application obtains temporary credentials for a service from an authorization server, in the form of tickets and session keys. The service server must be able to handle its part of the Kerberos mutual authentication process.

DS servers can interoperate with Kerberos systems through their GSSAPI SASL authentication mechanism.

Meet the following constraints when working with Kerberos systems:

- The clocks on the host systems where the DS server runs must be kept in sync with other hosts in the system.

  For example, you can use Network Time Protocol (NTP) services to keep the clocks in sync.
- Each DS server needs its own keytab file, the file which holds its pairs of Kerberos principals and keys.
- DS server debug logging can display exceptions about unsupported encryption types when a mismatch occurs. The exceptions are visible only when you activate debug logging.

Configure DS as a Kerberos Service Server

Follow these steps when setting up DS servers as Kerberos service servers:

1. Make sure the clock on the DS server host system is in sync with the other hosts' clocks in the Kerberos system.
2. Make sure that DNS resolves fully qualified domain names correctly on all systems involved.
3. Make sure the necessary ports are open on the DS server host system.
4. Make sure the encryption strengths required by the Kerberos system are supported by the JVM that the DS server uses.
5. Make sure that Kerberos is operating correctly for other services, including Kerberos client services on the DS server host.

   This step depends on the implementation, but usually includes adding a Kerberos principal for the host.

6. Add a Kerberos principal for the DS server, such as `ldap/ds.example.com`.
7. Create a keytab file for the DS server.

   This step depends on the Kerberos implementation, but generally consists of extracting the key for the DS server Kerberos principal, such as `ldap/ds.example.com` on the host where the DS server runs.
8. Make the keytab file readable only by the DS server, and copy it to the `/path/to/opendj/config/` directory.
9. Configure the DS server to handle GSSAPI SASL authentication:

```
$ dsconfig \
 set-sasl-mechanism-handler-prop \
 --handler-name GSSAPI \
 --set enabled:true \
 --set keytab:/path/to/opendj/config/opendj.keytab \
 --set server-fqdn:ds.example.com \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

10. If your Kerberos principal user identifiers are not of the form *`name@realm`*, configure an appropriate `identity-mapper` for the GSSAPI SASL mechanism handler.

    By default, the DS server uses the regular expression identity mapper, which expects user identifiers to match the pattern `^([^@])@.$`. It maps the string before `@` to the value of the UID attribute. This works well for identifiers like `bjensen@EXAMPLE.COM`. For background information, see Identity Mappers.
11. Restart the DS server to ensure the configuration changes are taken into account:

```
$ stop-ds --restart

...GSSAPI SASL mechanism using a server fully qualified
domain name of:
 ds.example.com
...GSSAPI mechanism using a principal name of:
 principal="opendj/ds.example.com"
```

```
...The GSSAPI SASL mechanism handler initialization was
successful
```

12. Test that the mechanism works, by authenticating as a Kerberos user:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --baseDN dc=example,dc=com \
 --saslOption mech=GSSAPI \
 --saslOption authid=bjensen@EXAMPLE.COM \
 uid=bjensen \
 cn

dn: uid=bjensen,ou=People,dc=example,dc=com
cn: Barbara Jensen
cn: Babs Jensen
```

Passwords

DS servers simplify safe, centralized password management. DS servers use password policies to govern passwords.

Which Password Policy Applies

The operational attribute, `pwdPolicySubentry`, identifies an account’s password policy. The default global access control instructions prevent this operational attribute from being visible to normal users. The following example grants access to a group of administrators:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
```

```
dn: ou=People,dc=example,dc=com
changetype: modify
add: aci
aci: (targetattr = "pwdPolicySubentry||ds-pwp-password-policy-dn")
 (version 3.0;acl "Allow Administrators to manage user's password
policy";
 allow (all) (groupdn = "ldap:///cn=Directory
Administrators,ou=Groups,dc=example,dc=com");)
EOF

$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
 --bindPassword bribery \
 --baseDN dc=example,dc=com \
 "(uid=bjensen)" \
 pwdPolicySubentry

dn: uid=bjensen,ou=People,dc=example,dc=com
pwdPolicySubentry: cn=Default Password Policy,cn=Password
Policies,cn=config
```

Configure Password Policies

Adjust the Default Password Policy

You can reconfigure the default password policy, for example, to check that passwords do not contain complete attribute values, and to prevent password reuse. The default policy is a per-server password policy.

1. Apply the changes to the default password policy:

```
$ dsconfig \
 set-password-policy-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --policy-name "Default Password Policy" \
 --set password-history-count:7 \
 --set password-validator:Attribute\ Value \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

2. Check your work:

```
$ dsconfig \
 get-password-policy-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --policy-name "Default Password Policy" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt

Property                                   : Value(s)
-------------------------------------------:---------------
-----------
account-status-notification-handler        : -
allow-expired-password-changes             : false
allow-user-password-changes                : true
default-password-storage-scheme            : PBKDF2-HMAC-
SHA256
deprecated-password-storage-scheme         : -
expire-passwords-without-warning           : false
force-change-on-add                        : false
force-change-on-reset                      : false
grace-login-count                          : 0
idle-lockout-interval                      : 0 s
last-login-time-attribute                  : -
last-login-time-format                     : -
lockout-duration                           : 0 s
lockout-failure-count                      : 0
lockout-failure-expiration-interval        : 0 s
max-password-age                           : 0 s
max-password-reset-age                     : 0 s
min-password-age                           : 0 s
password-attribute                         : userPassword
password-change-requires-current-password : false
```

```
password-expiration-warning-interval      : 5 d
password-generator                        : Random
Password Generator
password-history-count                    : 7
password-history-duration                 : 0 s
password-validator                        : Attribute
Value
previous-last-login-time-format           : -
require-change-by-time                    : -
require-secure-authentication             : true
require-secure-password-changes           : true
```

3. Test changes to the default password policy.

   For example, the following tests demonstrate the attribute value password validator. The attribute value password validator rejects a new password when the password is contained in attribute values on the user's entry.

   By default, the attribute value password validator checks all attributes, checks whether portions of the password string match attribute values, where the portions are strings of length 5, and checks the reverse of the password as well:

```
$ dsconfig \
 get-password-validator-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --validator-name Attribute\ Value \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt

Property               : Value(s)
-----------------------:----------------------------------
----------------
check-substrings       : true
enabled                : true
match-attribute        : All attributes in the user entry
will be checked.
min-substring-length   : 5
test-reversed-password : true
```

   Consider the attributes present on Babs Jensen's entry:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --baseDN dc=example,dc=com \
 "(uid=bjensen)"

dn: uid=bjensen,ou=People,dc=example,dc=com
objectClass: person
objectClass: cos
objectClass: jsonObject
objectClass: inetOrgPerson
objectClass: organizationalPerson
objectClass: posixAccount
objectClass: top
classOfService: bronze
cn: Barbara Jensen
cn: Babs Jensen
departmentNumber: 3001
description: Original description
diskQuota: 10 GB
facsimileTelephoneNumber: +1 408 555 1992
gidNumber: 1000
givenName: Barbara
homeDirectory: /home/bjensen
json:
{"access_token":"123","expires_in":59,"token_type":"Bearer
","refresh_token":"456"}
l: San Francisco
mail: bjensen@example.com
mailQuota: 1 GB
manager: uid=trigden, ou=People, dc=example,dc=com
ou: Product Development
ou: People
preferredLanguage: en, ko;q=0.8
roomNumber: 0209
sn: Jensen
street: 201 Mission Street Suite 2900
telephoneNumber: +1 408 555 1862
uid: bjensen
uidNumber: 1076
```

Using the attribute value password validator, passwords like `bjensen12` and `babsjensenspwd` are not valid because substrings of the password match complete attribute values:

```
$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN "uid=bjensen,ou=people,dc=example,dc=com" \
 --bindPassword hifalutin \
 --newPassword bjensen12

The LDAP password modify operation failed: 19 (Constraint
Violation)
Additional Information:  The provided new password failed
the validation
checks defined in the server: The provided password was
found in another
attribute in the user entry

$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN "uid=bjensen,ou=people,dc=example,dc=com" \
 --bindPassword hifalutin \
 --newPassword babsjensenspwd

The LDAP password modify operation failed: 19 (Constraint
Violation)
Additional Information:  The provided new password failed
the validation
checks defined in the server: The provided password was
found in another
attribute in the user entry
```

The attribute value password validator does not check, however, whether the password contains substrings of attribute values:

```
$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN "uid=bjensen,ou=people,dc=example,dc=com" \
 --bindPassword hifalutin \
 --newPassword babsp4ssw0rd

The LDAP password modify operation was successful

$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN "uid=bjensen,ou=people,dc=example,dc=com" \
 --bindPassword babsp4ssw0rd \
 --newPassword example.com

The LDAP password modify operation was successful
```

To avoid the problem of the latter example, you could use a dictionary password validator where the dictionary includes `example.com`.

Configure a NIST-Inspired Subentry Policy

You can configure a password policy inspired by NIST 800-63 requirements:

- Use a strong password storage scheme.
- Enforce a minimum password length of 8 characters.
- Check for matches in a dictionary of compromised passwords.
- Do not use composition rules for password validation.

  In other words, do not require a mix of special characters, upper and lower case letters, numbers, or other composition rules.
- Do not enforce arbitrary password changes.

  In other words, do not set a maximum password age.

Follow these steps to set up a replicated, NIST-inspired LDAP subentry password policy:

1. Gzip a copy of a text file of common compromised passwords, one word per line.

   This example shows the gzipped text file as `/tmp/10k_most_common.gz`. After successfully updating a subentry password policy with the dictionary data, the input file is no longer required. Lists of common passwords can be found online.

2. Make sure you have enabled a strong storage scheme.

   Creating a password storage scheme requires access to edit the server configuration, which you might not have when creating a subentry password policy. This example therefore uses the `PBKDF2-HMAC-SHA512` storage scheme, which is enabled by default to use 10,000 iterations.

   This scheme is intentionally much slower and more CPU-intensive than the `PBKDF2-HMAC-SHA256` scheme with 10 iterations used by the default password policy when you install DS. Test that you have enough resources to sustain the expected peak rates of impacted operations before using a much stronger password storage scheme in your production deployment.

   Impacted operations include:

   - Adding or importing entries with passwords.
   - Authenticating using a password, such as simple bind.
   - Updating or resetting a password.

3. Make sure password policy administrators have the `subentry-write` privilege, and any required ACIs needed to write password policy subentries in the directory data.

   The following example grants access to password administrators. The administrator accounts are in the data where the password policy is to be stored:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
```

```
dn: cn=subentry-write privilege for
administrators,dc=example,dc=com
objectClass: collectiveAttributeSubentry
objectClass: extensibleObject
objectClass: subentry
objectClass: top
cn: subentry-write privilege for administrators
ds-privilege-name;collective: subentry-write
subtreeSpecification: {base "ou=people",
specificationFilter
  "(isMemberOf=cn=Directory
Administrators,ou=Groups,dc=example,dc=com)" }

dn: dc=example,dc=com
changetype: modify
add: aci
aci: (target="ldap:///dc=example,dc=com")
 (targetattr = "*||ds-pwp-password-policy-
dn||pwdPolicySubentry||subtreeSpecification")
 (version 3.0; acl "Admins can manage entries and password
policies"; allow(all)
 groupdn = "ldap:///cn=Directory
Administrators,ou=Groups,dc=example,dc=com";)
EOF
```

Notice here that the directory superuser, `uid=admin`, assigns privileges. Any administrator with the `privilege-change` privilege can assign privileges. However, if the administrator can update administrator privileges, they can assign themselves the `bypass-acl` privilege. Then they are no longer bound by access control instructions, including both user data ACIs and global ACIs. For this reason, do not assign the `privilege-change` privilege to normal administrator users.

4. Create the password policy as one of the password policy administrators:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
 --bindPassword bribery << EOF
dn: cn=NIST inspired policy,dc=example,dc=com
```

```
objectClass: top
objectClass: subentry
objectClass: ds-pwp-password-policy
objectClass: ds-pwp-validator
objectClass: ds-pwp-length-based-validator
objectClass: ds-pwp-dictionary-validator
cn: NIST inspired policy
ds-pwp-password-attribute: userPassword
ds-pwp-default-password-storage-scheme: PBKDF2-HMAC-SHA512
ds-pwp-length-based-min-password-length: 8
ds-pwp-dictionary-data:<file:///tmp/10k_most_common.gz
subtreeSpecification: {base "ou=people",
specificationFilter "(objectclass=person)" }
EOF
```

After successfully adding the policy with the dictionary data, you can delete the input file.

5. Check the password policy works appropriately.

   The following example shows a rejected password modification:

```
$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN "uid=bjensen,ou=people,dc=example,dc=com" \
 --bindPassword hifalutin \
 --newPassword secret12

The LDAP password modify operation failed: 19 (Constraint
Violation)
Additional Information:  The provided new password failed
the validation
checks defined in the server: The provided password was
found in another
attribute in the user entry
```

   The following example shows an accepted password modification:

```
$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN "uid=bjensen,ou=people,dc=example,dc=com" \
 --bindPassword hifalutin \
 --newPassword aET1OjQeVJECSMgxDPs3U6In

The LDAP password modify operation was successful
```

Create a Per-Server Password Policy

This example adds a per-server password policy for new users who have not yet used their credentials to bind:

1. Create the new password policy:

```
$ dsconfig \
 create-password-policy \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --policy-name "New Account Password Policy" \
 --set default-password-storage-scheme:PBKDF2-HMAC-SHA256
\
 --set force-change-on-add:true \
 --set password-attribute:userPassword \
 --type password-policy \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

   As per-server password policies are not replicated, repeat this step on all replica directory servers.

2. Check your work:

```
$ dsconfig \
 get-password-policy-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --policy-name "New Account Password Policy" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt

Property                                  : Value(s)
------------------------------------------:---------------
----
account-status-notification-handler       : -
allow-expired-password-changes            : false
allow-user-password-changes               : true
default-password-storage-scheme           : PBKDF2-HMAC-
SHA256
deprecated-password-storage-scheme        : -
expire-passwords-without-warning          : false
force-change-on-add                       : true
force-change-on-reset                     : false
grace-login-count                         : 0
idle-lockout-interval                     : 0 s
last-login-time-attribute                 : -
last-login-time-format                    : -
lockout-duration                          : 0 s
lockout-failure-count                     : 0
lockout-failure-expiration-interval       : 0 s
max-password-age                          : 0 s
max-password-reset-age                    : 0 s
min-password-age                          : 0 s
password-attribute                        : userPassword
password-change-requires-current-password : false
password-expiration-warning-interval      : 5 d
password-generator                        : -
password-history-count                    : 0
password-history-duration                 : 0 s
password-validator                        : -
previous-last-login-time-format           : -
require-change-by-time                    : -
require-secure-authentication             : false
require-secure-password-changes           : false
```

3. Change the user’s password policy after the password is successfully updated.

For instructions on assigning a per-server password policy, s See Assign a Password Policy to a User.

List Subentry Password Policies

Per-server password policies are part of the DS server configuration. Use the `dsconfig` command to list, read, and edit them.

Subentry policies are part of the DS directory data. Use the `ldapsearch` command to list and read them.

The following command lists the subentry password policies under `dc=example,dc=com`:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
 --bindPassword bribery \
 --baseDn dc=example,dc=com \
 "(&(objectClass=subEntry)(objectClass=ds-pwp-password-policy))"

dn: cn=NIST inspired policy,dc=example,dc=com
objectClass: top
objectClass: subentry
objectClass: ds-pwp-password-policy
objectClass: ds-pwp-validator
objectClass: ds-pwp-length-based-validator
cn: NIST inspired policy
ds-pwp-default-password-storage-scheme: PBKDF2-HMAC-SHA512
ds-pwp-length-based-min-password-length: 8
ds-pwp-password-attribute: userPassword
ds-pwp-dictionary-data: <data>
```

Assign Password Policies

Type	To Assign...
Per-server password policy	Set the ds-pwp-password-policy-dn operational attribute on the user’s account.

Type	To Assign...
Subentry password policy	Use one of the following methods: • Set the ds-pwp-password-policy-dn operational attribute on the user's account. • Add the policy to an LDAP subentry whose immediate superior is the root of the subtree containing the accounts. For example, add the subentry password policy under `ou=People,dc=example,dc=com`. It applies to all accounts under `ou=People,dc=example,dc=com`. • Use the capabilities of LDAP subentries. Refine the scope of application by setting the subtreeSpecification attribute on the policy entry.

IMPORTANT

Do not assign more than one password policy to the same account. Conflicting password policies will yield inconsistent results.

You can review the password policy assigned to an account by reading the `pwdPolicySubentry` attribute on the entry.

Assign a Password Policy to a User

1. Make sure the password administrator has access to manage password policies:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: cn=subentry-write privilege for
administrators,dc=example,dc=com
objectClass: collectiveAttributeSubentry
objectClass: extensibleObject
objectClass: subentry
objectClass: top
```

```
cn: subentry-write privilege for administrators
ds-privilege-name;collective: subentry-write
subtreeSpecification: {base "ou=people",
specificationFilter
  "(isMemberOf=cn=Directory
Administrators,ou=Groups,dc=example,dc=com)" }

dn: dc=example,dc=com
changetype: modify
add: aci
aci: (target="ldap:///dc=example,dc=com")
 (targetattr = "*||ds-pwp-password-policy-
dn||pwdPolicySubentry||subtreeSpecification")
 (version 3.0; acl "Admins can manage entries and password
policies"; allow(all)
 groupdn = "ldap:///cn=Directory
Administrators,ou=Groups,dc=example,dc=com";)
EOF
```

Notice here that the directory superuser, `uid=admin`, assigns privileges. Any administrator with the `privilege-change` privilege can assign privileges. However, if the administrator can update administrator privileges, they can assign themselves the `bypass-acl` privilege. Then they are no longer bound by access control instructions, including both user data ACIs and global ACIs. For this reason, do not assign the `privilege-change` privilege to normal administrator users.

2. Set the user’s `ds-pwp-password-policy-dn` attribute as the password administrator:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
 --bindPassword bribery << EOF
dn: uid=newuser,ou=People,dc=example,dc=com
uid: newuser
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
objectClass: top
```

```
cn: New User
sn: User
ou: People
mail: newuser@example.com
userPassword: chngthspwd
ds-pwp-password-policy-dn: cn=NIST inspired
policy,dc=example,dc=com
EOF
```

3. Check your work:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
 --bindPassword bribery \
 --baseDN dc=example,dc=com \
 "(uid=newuser)" \
 pwdPolicySubentry

dn: uid=newuser,ou=People,dc=example,dc=com
pwdPolicySubentry: cn=NIST inspired
policy,dc=example,dc=com
```

Assign a Password Policy to a Group

You can use a collective attribute to assign a password policy. Collective attributes provide a standard mechanism for defining attributes that appear on all the entries in a subtree. For details, see Collective Attributes:

1. Make sure the password administrator has the privilege to write subentries:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: uid=kvaughan,ou=People,dc=example,dc=com
changetype: modify
add: ds-privilege-name
ds-privilege-name: subentry-write
EOF
```

Notice here that the directory superuser, `uid=admin`, assigns privileges. Any administrator with the `privilege-change` privilege can assign privileges. However, if the administrator can update administrator privileges, they can assign themselves the `bypass-acl` privilege. Then they are no longer bound by access control instructions, including both user data ACIs and global ACIs. For this reason, do not assign the `privilege-change` privilege to normal administrator users.

2. Create a subentry defining the collective attribute that sets the `ds-pwp-password-policy-dn` attribute for group members' entries:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
 --bindPassword bribery << EOF
dn: cn=Password Policy for Dir Admins,dc=example,dc=com
objectClass: collectiveAttributeSubentry
objectClass: extensibleObject
objectClass: subentry
objectClass: top
cn: Password Policy for Dir Admins
ds-pwp-password-policy-dn;collective: cn=Root Password
Policy,cn=Password Policies,cn=config
subtreeSpecification: { base "ou=People",
specificationFilter
  "(isMemberOf=cn=Directory
Administrators,ou=Groups,dc=example,dc=com)"}
EOF
```

3. Check your work:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
 --bindPassword bribery \
 --baseDN dc=example,dc=com \
 "(uid=kvaughan)" \
 pwdPolicySubentry

dn: uid=kvaughan,ou=People,dc=example,dc=com
pwdPolicySubentry: cn=Root Password Policy,cn=Password
Policies,cn=config
```

Assign a Password Policy to a Branch

These steps apply only to subentry password policies:

1. Give an administrator the privilege to write subentries, such as those used for setting password policies:

   ```
   $ ldapmodify \
    --hostname localhost \
    --port 1636 \
    --useSsl \
    --usePkcs12TrustStore /path/to/opendj/config/keystore \
    --trustStorePassword:file
   /path/to/opendj/config/keystore.pin \
    --bindDN uid=admin \
    --bindPassword password << EOF
   dn: uid=kvaughan,ou=People,dc=example,dc=com
   changetype: modify
   add: ds-privilege-name
   ds-privilege-name: subentry-write
   EOF
   ```

   Notice here that the directory superuser, `uid=admin`, assigns privileges. Any administrator with the `privilege-change` privilege can assign privileges. However, if the administrator can update administrator privileges, they can assign themselves the `bypass-acl` privilege. Then they are no longer bound by access control instructions, including both user data ACIs and global ACIs. For this reason, do not assign the `privilege-change` privilege to normal administrator users.

2. Configure a subentry password policy with a `subtreeSpecification` attribute that defines which accounts are assigned the policy.

   The following example assigns `cn=NIST inspired policy` to accounts under `ou=People,dc=example,dc=com`:

   ```
   $ ldapmodify \
    --hostname localhost \
    --port 1636 \
    --useSsl \
    --usePkcs12TrustStore /path/to/opendj/config/keystore \
    --trustStorePassword:file
   /path/to/opendj/config/keystore.pin \
    --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
    --bindPassword bribery << EOF
   dn: cn=NIST inspired policy,dc=example,dc=com
   changetype: modify
   replace: subtreeSpecification
   subtreeSpecification: { base "ou=people" }
   EOF
   ```

   The subtree specification assigns the policy to the people branch with `{ base "ou=people" }`. You could relax the subtree specification value to `{}` to apply the policy to all entries anywhere underneath `dc=example,dc=com`. You could further restrict the subtree specification by adding a `specificationFilter`. For details, see About Subentry Scope.

3. Check your work to see that an account under `ou=People` has the policy:

   ```
   $ ldapsearch \
    --hostname localhost \
    --port 1636 \
    --useSsl \
    --usePkcs12TrustStore /path/to/opendj/config/keystore \
    --trustStorePassword:file
   /path/to/opendj/config/keystore.pin \
    --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
    --bindPassword bribery \
    --baseDN dc=example,dc=com \
    "(uid=alutz)" \
    pwdPolicySubentry
   ```

```
dn: uid=alutz,ou=People,dc=example,dc=com
pwdPolicySubentry: cn=NIST inspired
policy,dc=example,dc=com
```

About Subentry Scope

LDAP subentries reside with the user data and so the server replicates them. Subentries hold operational data. They are not visible in search results unless explicitly requested. This section describes how a subentry's `subtreeSpecification` attribute defines the scope of the subtree that the subentry applies to.

An LDAP subentry's subtree specification identifies a subset of entries in a branch of the DIT. The subentry scope is these entries. In other words, these are the entries that the subentry affects.

The attribute value for a `subtreeSpecification` optionally includes the following parameters:

`base`
: Indicates the entry, *relative to the subentry's parent*, at the base of the subtree.

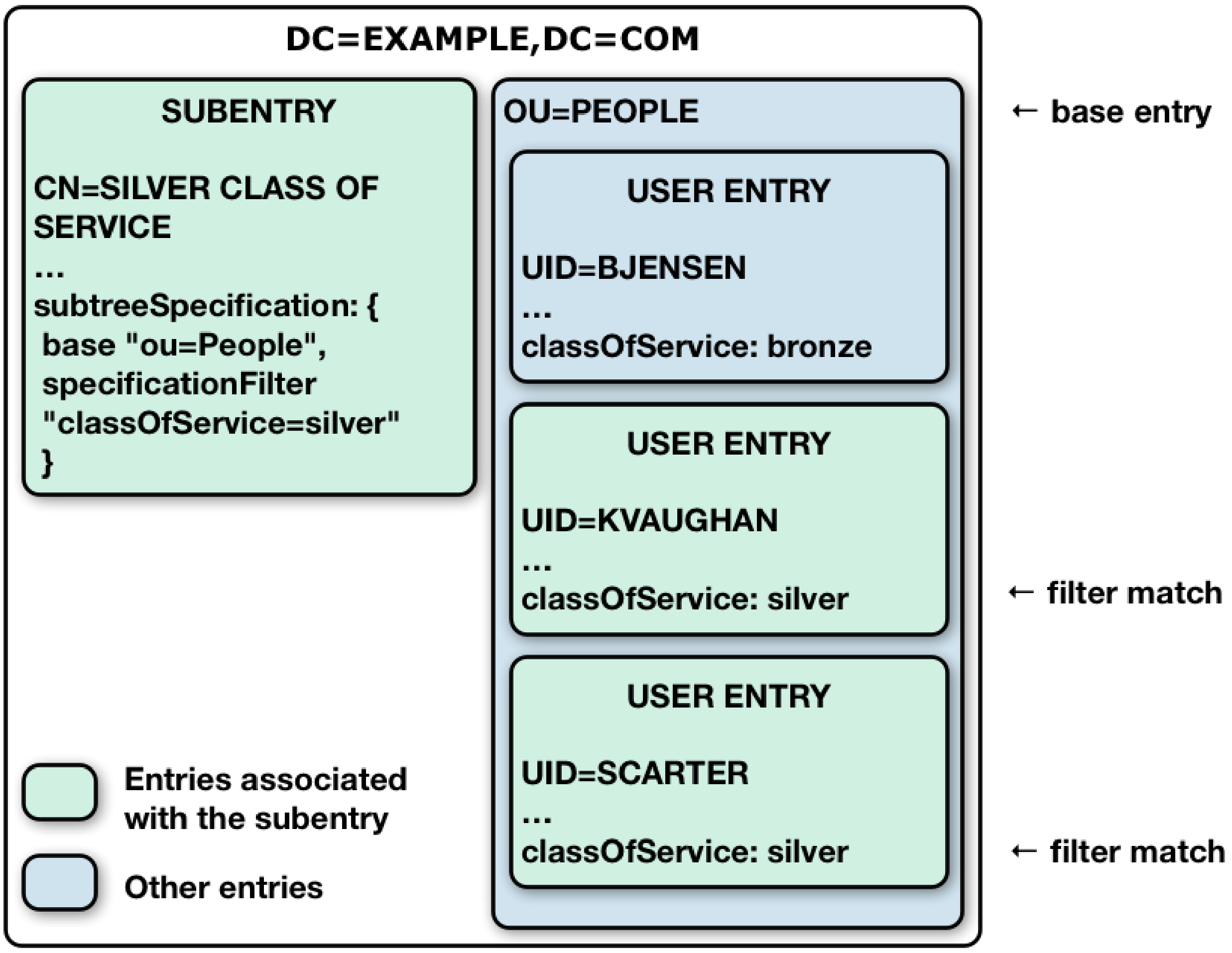
  By default, the base is the subentry's parent.

`specificationFilter`
: Indicates an LDAP filter. Entries matching the filter are in scope.

  DS servers extend the standard implementation to allow any search filter, not just an assertion about the `objectClass` attribute.

  By default, all entries under the base entry are in scope.

The following illustration shows this for an example collective attribute subentry:



Notice that the base of `ou=People` on the subentry `cn=Silver Class of Service,dc=example,dc=com` indicates that the base entry is `ou=People,dc=example,dc=com`.

The filter `"(classOfService=silver)"` means that Kirsten Vaughan and Sam Carter's entries are in scope. Babs Jensen's entry, with `classOfService: bronze` does not match and is therefore not in scope. The `ou=People` organizational unit entry does not have a `classOfService` attribute, and so is not in scope, either.

Strong and Safe Passwords

The difficulty with passwords is that they tend to be relatively easy to guess. Despite decades of advice on how to pick strong passwords, people still routinely pick very weak passwords using common words and phrases or simple variations of them. This makes them extremely easy to guess. Attackers with access to even modest hardware can make billions of guesses per second.

DS servers provide flexible password validation to fit your policies about password content, and to reject weak passwords when users try to save them. It also provides a variety of one-way and reversible password storage schemes. Password strength is a function of both password minimum length, which you can set as part of password policy, and password quality, which requires password validation.

Password Validation

When a password is added or updated, a password validator determines whether the server should accept it. Validation does not affect existing passwords.

A user's password policy specifies which password validators apply whenever that user provides a new password.

Subentry password policies can include attributes of password validator object classes. Each object class derives from the abstract `ds-pwp-validator` class:

- ds-pwp-attribute-value-validator Attributes
- ds-pwp-character-set-validator Attributes
- ds-pwp-dictionary-validator Attributes
- ds-pwp-length-based-validator Attributes
- ds-pwp-repeated-characters-validator Attributes
- ds-pwp-similarity-based-validator Attributes
- ds-pwp-unique-characters-validator Attributes

The example that follows shows a password policy that requires new passwords to have at least three of the following four character classes:

- English lowercase characters (a through z)
- English uppercase characters (A through Z)
- Base 10 digits (0 through 9)
- Punctuation characters (for example, !, $, #, %)

Notice how the `character-set` values are constructed. The initial `0:` means the set is optional, whereas `1:` means the set is required:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: cn=Policy with character set validation,dc=example,dc=com
objectClass: top
objectClass: subentry
objectClass: ds-pwp-password-policy
objectClass: ds-pwp-validator
```

```
objectClass: ds-pwp-character-set-validator
cn: Policy with character set validation
ds-pwp-password-attribute: userPassword
ds-pwp-default-password-storage-scheme: PBKDF2-HMAC-SHA256
ds-pwp-character-set-allow-unclassified-characters: true
ds-pwp-character-set-character-set-ranges: 0:a-z
ds-pwp-character-set-character-set-ranges: 0:A-Z
ds-pwp-character-set-character-set-ranges: 0:0-9
ds-pwp-character-set-character-set: 0:!$%^.#
ds-pwp-character-set-min-character-sets: 3
subtreeSpecification: { base "ou=people", specificationFilter "
(uid=bjensen)" }
EOF

$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password \
 --authzID "u:bjensen" \
 --newPassword '!ABcd$%^'
```

An attempt to set an invalid password fails as shown in the following example:

```
$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password \
 --authzID "u:bjensen" \
 --newPassword hifalutin

The LDAP password modify operation failed: 19 (Constraint
Violation)
Additional Information:  The provided new password failed the
validation
checks defined in the server: The provided password did not
contain characters
from at least 3 of the following character sets or ranges:
```

```
'!$%^.#', '0-9',
'A-Z', 'a-z'
```

Per-server password policies use validators that are separate configuration objects. The following example lists the password validators available by default for per-server password policies. By default, no password validators are configured in the default password policy:

```
$ dsconfig \
 list-password-validators \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt

Password Validator                   : Type                 :
enabled
------------------------------------:---------------------:-------
-
At least 8 characters                : length-based         : true
Attribute Value                      : attribute-value      : true
Character Set                        : character-set        : true
Common passwords                     : dictionary           : true
Dictionary                           : dictionary           : false
Length-Based Password Validator      : length-based         : true
Repeated Characters                  : repeated-characters  : true
Similarity-Based Password Validator  : similarity-based     : true
Unique Characters                    : unique-characters    : true
```

For details, see Password Validator.

For an example showing how to test password quality, see Check Password Quality.

Password Storage

Password storage schemes, described in Password Storage Scheme, encode new passwords and store the encoded version. When a client application authenticates with the password, the server encodes the plaintext password using the configured storage scheme, and checks whether the result matches the encoded value stored by the server. If the encoded version is appropriately secure, it is difficult to guess the plaintext password from its encoded value.

DS servers offer a variety of reversible and one-way password storage schemes. With a reversible encryption scheme, an attacker who gains access to the server can recover the plaintext passwords. With a one-way hash storage scheme, the attacker who gains access to the server must still crack the password by brute force, encoding passwords over and over to generate guesses until a match is found. If you have a choice, use a one-way password storage scheme.

Some one-way hash functions are not designed specifically for password storage, but also for use in message authentication and digital signatures. Such functions, like those defined in the Secure Hash Algorithm (SHA-1 and SHA-2) standards, are designed for high performance. Because they are fast, they allow the server to perform authentication at high throughput with low response times. However, high-performance algorithms also help attackers use brute force techniques. One estimate in 2017 is that a single GPU can calculate over one billion SHA-512 hashes per second.

WARNING

Some one-way hash functions are designed to be computationally *expensive*. Such functions, like PBKDF2 and Bcrypt, are designed to be relatively slow even on modern hardware. This makes them generally less susceptible to brute force attacks.

However, computationally expensive functions reduce authentication throughput and increase response times. With the default number of iterations, the GPU mentioned above might only calculate 100,000 PBKDF2 hashes per second (or 0.01% of the corresponding hashes calculated with SHA-512). If you use these functions, be aware of the potentially dramatic performance impact and plan your deployment accordingly.

Modern hardware and techniques to pre-compute attempts, such as rainbow tables, make it increasingly easy for attackers to crack passwords by brute force. Password storage schemes that use *salt* make brute force attacks more expensive. In this context, salt is a random value appended to the password before encoding. The salt is then stored with the encoded value and used when comparing an incoming password to the stored password.

Reversible password storage schemes, such as AES and Blowfish, use symmetric keys for encryption.

The following example lists available alternatives, further described in Password Storage Schemes:

```
$ dsconfig \
 list-password-storage-schemes \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt

Password Storage Scheme : Type                 : enabled
------------------------:----------------------:--------
3DES                    : triple-des           : false
AES                     : aes                  : false
Base64                  : base64               : false
Bcrypt                  : bcrypt               : true
Blowfish                : blowfish             : false
Clear                   : clear                : false
CRYPT                   : crypt                : false
PBKDF2                  : pbkdf2               : false
PBKDF2-HMAC-SHA256      : pbkdf2-hmac-sha256 : true
PBKDF2-HMAC-SHA512      : pbkdf2-hmac-sha512 : true
PKCS5S2                 : pkcs5s2              : false
Salted SHA-1            : salted-sha1          : false
Salted SHA-256          : salted-sha256        : false
Salted SHA-384          : salted-sha384        : false
Salted SHA-512          : salted-sha512        : false
SCRAM-SHA-256           : scram-sha256         : true
SCRAM-SHA-512           : scram-sha512         : true
SHA-1                   : sha1                 : false
```

As shown in Adjust the Default Password Policy, the default password storage scheme for users is PBKDF2-HMAC-SHA256. When you add users or import user entries with `userPassword` values in plaintext, the DS server hashes them with the default password storage scheme. The default directory superuser has a different password policy, shown in Assign a Password Policy to a Group. The Root Password Policy uses PBKDF2-HMAC-SHA256 by default.

TIP

The choice of default password storage scheme for normal users can significantly impact server performance. Each time a normal user authenticates using simple bind (username/password) credentials, the directory server encodes the user’s password according to the storage scheme in order to compare it with the encoded value in the user’s entry.

Schemes such as Salted SHA-512 call for relatively high-performance encoding. Schemes such as PBKDF2-HMAC-SHA256, which are designed to make the encoding process computationally expensive, reduce the bind throughput that can be achieved on equivalent hardware.

Take this performance impact into consideration when sizing your deployment. With a computationally expensive scheme such as PBKDF2-HMAC-SHA256, make sure the directory service has enough compute power to absorb the additional load.

Password Storage Schemes

Name	Type of Algorithm	Notes
3DES(1)	Reversible encryption(2)	Triple DES (Data Encryption Standard) in EDE (Encrypt Decrypt Encrypt) mode. Key size: 168 bits.
AES(1)	Reversible encryption(2)	Advanced Encryption Standard, successor to DES, published by the US National Institute of Standards and Technology (NIST). Key size: 128 bits.
Base64	Reversible encoding	Transfer encoding for representing binary password values in text. *Not intended as a secure storage scheme.*
Bcrypt	One-way hash	Computationally intensive hashing function, based on the Blowfish cipher. Default cost: 12 (2^12 iterations).
Blowfish(1)	Reversible encryption(2)	Public domain cipher designed by Bruce Schneier as a successor to DES. Key size: 128 bits.

Name	Type of Algorithm	Notes
Clear	Cleartext, no encoding	For backwards compatibility and use with certain legacy applications. *Not intended as a secure storage scheme.*
CRYPT	One-way hash	Based on the UNIX Crypt algorithm. For backwards compatibility and use with certain legacy applications. *Not intended as a secure storage scheme.* Default algorithm: `unix` .
MD5	One-way hash	Based on the MD5 algorithm defined in RFC 1321 . For backwards compatibility and use with certain legacy applications. *Not intended as a secure storage scheme.*
PBKDF2	One-way hash	Computationally intensive hashing function, based on PBKDF2 algorithm defined in RFC 8018, 5.2. PBKDF2 . Default iterations: 10000. The pseudorandom function for the algorithm corresponds to the HMAC based on SHA-1.
PBKDF2-HMAC-SHA256	One-way hash	Computationally intensive hashing function using PBKDF2. Default iterations: 10000. The pseudorandom function for the algorithm corresponds to the HMAC based on SHA-2, where the hash function is SHA-256.

Name	Type of Algorithm	Notes
PBKDF2-HMAC-SHA512	One-way hash	Computationally intensive hashing function using PBKDF2. Default iterations: 10000. The pseudorandom function for the algorithm corresponds to the HMAC based on SHA-2, where the hash function is SHA-512.
PKCS5S2	One-way hash	Computationally intensive hashing function, based on Atlassian’s adaptation of the PBKDF2. Number of iterations: 10000.
RC4(1)	Reversible encryption(2)	Based on the Rivest Cipher 4 algorithm. For backwards compatibility and use with certain legacy applications. *Not intended as a secure storage scheme.* Key size: 128 bits.
Salted MD5	One-way hash	Based on MD5, with 64 bits of random salt appended to the plaintext before hashing, and then appended to the hash.
Salted SHA-1	One-way hash	Based on SHA-1, with 64 bits of random salt appended to the plaintext before hashing, and then appended to the hash.
Salted SHA-256	One-way hash	Based on the SHA-256 hash function using 32-bit words and producing 256-bit digests. SHA-256 is defined in the SHA-2 (Secure Hash Algorithm 2) standard developed by the US National Security Agency (NSA) and published by NIST. The salt is applied as for Salted SHA-1.

Name	Type of Algorithm	Notes
Salted SHA-384	One-way hash	Based on the SHA-384 hash function that effectively truncates the digest of SHA-512 to 384 bits. SHA-384 is defined in the SHA-2 (Secure Hash Algorithm 2) standard developed by the NSA and published by NIST. The salt is applied as for Salted SHA-1.
Salted SHA-512	One-way hash	Based on the SHA-512 hash function using 64-bit words and producing 512-bit digests. SHA-512 is defined in the SHA-2 (Secure Hash Algorithm 2) standard developed by the NSA and published by NIST. The salt is applied as for Salted SHA-1.

Name	Type of Algorithm	Notes
SCRAM-SHA-256	One-way hash	For use with the standard SASL Salted Challenge Response Authentication Mechanism (SCRAM), named `SCRAM-SHA-256`. A SASL SCRAM mechanism provides a secure alternative to transmitting plaintext passwords during binds. It is an appropriate replacement for DIGEST-MD5 and CRAM-MD5. With a SCRAM SASL bind, the client must demonstrate proof that it has the original plaintext password. During the SASL bind, the client must perform computationally intensive processing to prove that it has the plaintext password. This computation is like what the server performs for PBKDF2, but the password is not communicated during the bind. Once the server has stored the password, the client pays the computational cost to perform the bind. The server only pays a high computational cost when the password is updated, for example, when an entry with a password is added or during a password modify operation. A SASL SCRAM mechanism therefore offers a way to offload the high computational cost of secure password storage to client applications during authentication. Passwords storage using a SCRAM storage scheme is compatible with simple binds and SASL PLAIN binds. When a password is stored using a SCRAM storage scheme, the server pays the computational cost to perform the bind during a simple bind or SASL PLAIN bind. The SCRAM password storage scheme must match the SASL SCRAM mechanism used for authentication. In other words, SASL SCRAM-SHA-256 requires a SCRAM-SHA-256 password storage scheme. SASL SCRAM-SHA-512 requires a SCRAM-SHA-512 password storage scheme. Default iterations: 10000.

Name	Type of Algorithm	Notes
		The pseudorandom function for the algorithm corresponds to the HMAC based on SHA-2, where the hash function is SHA-256.
SCRAM-SHA-512	One-way hash	Like SCRAM-SHA-256, but the hash function is SHA-512. The corresponding SASL mechanism is named `SCRAM-SHA-512`.
SHA-1	One-way hash	SHA-1 (Secure Hash Algorithm 1) standard developed by the NSA and published by NIST. *Not intended as a secure storage scheme.*

(1) Reversible encryption schemes are deprecated.

(2) When you configure a reversible password storage scheme, enable the `adminRoot` backend, and configure a replication domain for `cn=admin data`. These additional steps let the replicas store and replicate the secret keys for password encryption.

Password storage schemes listed in the following table have additional configuration settings.

Additional Password Storage Scheme Settings

Scheme	Setting	Description
Bcrypt	`bcrypt-cost`	The cost parameter specifies a key expansion iteration count as a power of two. A default value of 12 (2^{12} iterations) is considered in 2016 as a reasonable balance between responsiveness and security for regular users.
	`rehash-policy`	Whether the server should rehash passwords after the cost has been changed.

Scheme	Setting	Description
Crypt	`crypt-password-storage-encryption-algorithm`	Specifies the crypt algorithm to use to encrypt new passwords. The following values are supported: ***unix*** The password is encrypted with the weak Unix crypt algorithm. This is the default setting. ***md5*** The password is encrypted with the BSD MD5 algorithm and has a `$1$` prefix. ***sha256*** The password is encrypted with the SHA256 algorithm and has a `$5$` prefix. ***sha512*** The password is encrypted with the SHA512 algorithm and has a `$6$` prefix.
PBKDF2 PBKDF2-HMAC-SHA256 PBKDF2-HMAC-SHA512	`pbkdf2-iterations`	The number of algorithm iterations. The default is 10000.
	`rehash-policy`	Whether the server should rehash passwords after the cost has been changed.
SCRAM SCRAM-SHA-256 SCRAM-SHA-512	`scram-iterations`	The number of algorithm iterations. The default is 10000.

You change the default password policy storage scheme for users by changing the applicable password policy:

```
$ dsconfig \
 set-password-policy-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --policy-name "Default Password Policy" \
 --set default-password-storage-scheme:PBKDF2-HMAC-SHA512 \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt
```

Notice that the change in default password storage scheme does not cause the DS server to update any stored password values. By default, the server only stores a password with the new storage scheme the next time the password is changed.

For subentry password policies, set the `ds-pwp-default-password-storage-scheme` attribute to the common name of an enabled password storage scheme. To list the names of enabled password storage schemes, use the **`dsconfig list-password-storage-schemes`** command. The name appears in the first column of the output. The third column shows whether the scheme is enabled.

DS servers prefix passwords with the scheme used to encode them, which means it is straightforward to see which password storage scheme is used. After the default password storage scheme is changed to PBKDF2-HMAC-SHA512, old user passwords remain encoded with PBKDF2-HMAC-SHA256:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=bjensen,ou=people,dc=example,dc=com \
 --bindPassword hifalutin \
 --baseDN dc=example,dc=com \
 "(uid=bjensen)" \
 userPassword

dn: uid=bjensen,ou=People,dc=example,dc=com
userPassword: {PBKDF2-HMAC-SHA512}10000:<hash>
```

When the password is changed, the new default password storage scheme takes effect:

```
$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password \
 --authzID "u:bjensen" \
 --newPassword changeit

The LDAP password modify operation was successful

$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=bjensen,ou=people,dc=example,dc=com \
 --bindPassword changeit \
 --baseDN dc=example,dc=com \
 "(uid=bjensen)" \
 userPassword

dn: uid=bjensen,ou=People,dc=example,dc=com
userPassword: {PBKDF2-HMAC-SHA512}10000:<hash>
```

When you change the password storage scheme for users, realize that the user passwords must change in order for the DS server to encode them with the chosen storage scheme. If you are changing the storage scheme because the old scheme was too weak, then you no doubt want users to change their passwords anyway.

If, however, the storage scheme change is not related to vulnerability, use the `deprecated-password-storage-scheme` property in per-server password policies, or the `ds-pwp-deprecated-password-storage-scheme` attribute in subentry password policies. This setting causes the DS server to store the password in the new format after successful authentication. This makes it possible to do password migration for active users as users gradually change their passwords:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
 --bindPassword bribery \
 --baseDN dc=example,dc=com \
 "(uid=kvaughan)" \
 userPassword

dn: uid=kvaughan,ou=People,dc=example,dc=com
userPassword: {PBKDF2-HMAC-SHA256}10000:<hash>

$ dsconfig \
 set-password-policy-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --policy-name "Default Password Policy" \
 --set deprecated-password-storage-scheme:PBKDF2-HMAC-SHA256 \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt

$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=kvaughan,ou=people,dc=example,dc=com \
 --bindPassword bribery \
 --baseDN dc=example,dc=com \
 "(uid=kvaughan)" \
 userPassword

dn: uid=kvaughan,ou=People,dc=example,dc=com
userPassword: {PBKDF2-HMAC-SHA512}10000:<hash>
```

Notice that with `deprecated-password-storage-scheme` set appropriately, Kirsten Vaughan's password was hashed again after she authenticated successfully.

Password Generation

DS servers use password generators when responding with a generated password for the LDAP Password Modify extended operation. A directory administrator resetting a user's password has the server generate the new password, and the server sends the new password in the response:

```
$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password \
 --authzID "u:bjensen"

The LDAP password modify operation was successful
Generated Password:  <random>
```

The default password policy uses the Random Password Generator, described in Random Password Generator:

```
$ dsconfig \
 get-password-policy-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --policy-name "Default Password Policy" \
 --property password-generator \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt

Property           : Value(s)
-------------------:--------------------------
password-generator : Random Password Generator

$ dsconfig \
 get-password-generator-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --generator-name "Random Password Generator" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt

Property                : Value(s)
-----------------------:------------------------------------------
-------------
enabled                 : true
password-character-set :
```

```
alphanum:abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRS
                       : TUVWXYZ0123456789
password-format        : alphanum:10
```

Notice that the default configuration for the Random Password Generator sets the `password-character-set` property, and references the settings in the `password-format` property. Generated passwords have eight characters: three from the `alpha` set, followed by two from the `numeric` set, followed by three from the `alpha` set. The `password-character-set` name must be ASCII.

Subentry password policies configure `ds-pwp-random-generator` object class attributes. The following example creates a password with password generation, and demonstrates its use:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: cn=Policy with random password generation,dc=example,dc=com
objectClass: top
objectClass: subentry
objectClass: ds-pwp-password-policy
objectClass: ds-pwp-validator
objectClass: ds-pwp-length-based-validator
objectClass: ds-pwp-random-generator
cn: Policy with random password generation
ds-pwp-password-attribute: userPassword
ds-pwp-default-password-storage-scheme: PBKDF2-HMAC-SHA256
ds-pwp-random-password-character-set:
alpha:ABCDEFGHIJKLMNOPQRSTUVWabcdefghijklmnopqrstuvwxyz
ds-pwp-random-password-character-set: punct:,.!&+=-_
ds-pwp-random-password-character-set: numeric:0123456789
ds-pwp-random-password-format:
alpha:3,punct:1,numeric:2,punct:2,numeric:3,alpha:3,punct:2
ds-pwp-length-based-min-password-length: 8
subtreeSpecification: { base "ou=people" }
EOF

$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password \
 --authzID "u:bjensen"

The LDAP password modify operation was successful
Generated Password:  <random>
```

For details, see ds-pwp-random-generator Attributes.

When configuring both password validators and password generators, make sure the generated passwords are acceptable to the validator. In this case, the minimum length is less than the generated password length, for example.

Sample Password Policies

Lock Accounts After Repeated Bind Failures

To help you prevent brute-force attacks, where an attacker tries many passwords in the hope of eventually guessing correctly, DS password policies support configurable account lockout. This feature is an important part of a secure password policy.

NOTE

When you configure account lockout as part of password policy, DS servers lock an account after the specified number of consecutive authentication failures. *Account lockout is not transactional across all replicas in a deployment.* Global account lockout occurs as soon as the authentication failure times have been replicated.

The following commands demonstrate a subentry password policy that locks accounts for five minutes after three consecutive bind failures. With this policy, the directory server records failure times, and slowly discards them. As a result, a brute-force attack is hopefully too slow to be effective, but no administrative action is needed when a user temporarily forgets or mistypes their password.

Once an account is locked, binds continue to fail for the lockout period, even if the credentials are correct. An account administrator can use the **`manage-account`** command to view the account status, and to change it if necessary:

▼ Show example commands

```
# Set the password policy:
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: cn=Lock After Repeated Bind Failures,dc=example,dc=com
objectClass: top
objectClass: subentry
objectClass: ds-pwp-password-policy
cn: Lock After Repeated Bind Failures
ds-pwp-password-attribute: userPassword
ds-pwp-default-password-storage-scheme: PBKDF2-HMAC-SHA256
ds-pwp-lockout-duration: 5 m
ds-pwp-lockout-failure-count: 3
ds-pwp-lockout-failure-expiration-interval: 2 m
subtreeSpecification: { base "ou=people", specificationFilter "
(objectClass=posixAccount)" }
EOF

# Attempt to bind three times using the wrong password:
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDn uid=bjensen,ou=people,dc=example,dc=com \
 --bindPassword wrongPassword \
 --baseDn dc=example,dc=com \
 "(uid=bjensen)"

The LDAP bind request failed: 49 (Invalid Credentials)

$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDn uid=bjensen,ou=people,dc=example,dc=com \
 --bindPassword wrongPassword \
 --baseDn dc=example,dc=com \
 "(uid=bjensen)"
```

```
The LDAP bind request failed: 49 (Invalid Credentials)

$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDn uid=bjensen,ou=people,dc=example,dc=com \
 --bindPassword wrongPassword \
 --baseDn dc=example,dc=com \
 "(uid=bjensen)"

The LDAP bind request failed: 49 (Invalid Credentials)

# Observe the results:
$ manage-account \
 get-all \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --targetDN uid=bjensen,ou=people,dc=example,dc=com \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin

Password Policy DN:  cn=Lock After Repeated Bind
Failures,dc=example,dc=com
Seconds Until Authentication Failure Unlock:  <seconds>
```

Enforce Regular Password Changes

The following commands configure a subentry password policy that sets age limits on passwords, requiring that users change their passwords at least every 13 weeks, but not more often than every 4 weeks. The policy also sets the number of passwords to keep in the password history of the entry, preventing users from reusing the same password on consecutive changes:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: cn=Enforce Regular Password Changes,dc=example,dc=com
objectClass: top
objectClass: subentry
objectClass: ds-pwp-password-policy
cn: Enforce Regular Password Changes
ds-pwp-password-attribute: userPassword
ds-pwp-default-password-storage-scheme: PBKDF2-HMAC-SHA256
ds-pwp-max-password-age: 13 w
ds-pwp-min-password-age: 4 w
ds-pwp-password-history-count: 7
subtreeSpecification: { base "ou=people" }
EOF
```

Track Last Login Time

The following commands configure a subentry password policy that keeps track of the last successful login:

1. Set up an attribute to which the DS directory server can write a timestamp value on successful login.

   For additional information, see Active Accounts:

   ```
   $ ldapmodify \
    --hostname localhost \
    --port 1636 \
    --useSsl \
    --usePkcs12TrustStore /path/to/opendj/config/keystore \
    --trustStorePassword:file /path/to/opendj/config/keystore.pin
   \
    --bindDN uid=admin \
    --bindPassword password << EOF
   dn: cn=schema
   changetype: modify
   add: attributeTypes
   attributeTypes: ( lastLoginTime-oid
     NAME 'lastLoginTime'
     DESC 'Last time the user logged in'
     EQUALITY generalizedTimeMatch
     ORDERING generalizedTimeOrderingMatch
     SYNTAX 1.3.6.1.4.1.1466.115.121.1.24
     SINGLE-VALUE
     NO-USER-MODIFICATION
   ```

```
   USAGE directoryOperation
   X-ORIGIN 'DS example documentation' )
 EOF
```

2. Create the password policy to write the timestamp to the attribute on successful login:

   ```
   $ dsconfig \
    create-password-policy \
    --hostname localhost \
    --port 4444 \
    --bindDN uid=admin \
    --bindPassword password \
    --policy-name "Track Last Login Time" \
    --type password-policy \
    --set default-password-storage-scheme:PBKDF2-HMAC-SHA256 \
    --set password-attribute:userPassword \
    --set last-login-time-attribute:lastLoginTime \
    --set last-login-time-format:"yyyyMMddHH'Z'" \
    --usePkcs12TrustStore /path/to/opendj/config/keystore \
    --trustStorePassword:file /path/to/opendj/config/keystore.pin
   \
    --no-prompt
   ```

Deprecate a Password Storage Scheme

The following commands configure a subentry password policy for deprecating a password storage scheme. This policy uses elements from Enforce Regular Password Changes. The DS server applies the new password storage scheme to re-encode passwords:

- When they change.
- When the user successfully binds with the correct password, and the password is currently hashed with a deprecated scheme.

```
$ dsconfig \
 set-password-storage-scheme-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --scheme-name "Salted SHA-512" \
 --set enabled:true \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
```

```
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt

$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: cn=Deprecate a Password Storage Scheme,dc=example,dc=com
objectClass: top
objectClass: subentry
objectClass: ds-pwp-password-policy
cn: Deprecate a Password Storage Scheme
ds-pwp-password-attribute: userPassword
ds-pwp-default-password-storage-scheme: PBKDF2-HMAC-SHA256
ds-pwp-deprecated-password-storage-scheme: Salted SHA-512
ds-pwp-max-password-age: 13 w
ds-pwp-min-password-age: 4 w
ds-pwp-password-history-count: 7
subtreeSpecification: { base "ou=people" }
EOF
```

Lock Idle Accounts

The following commands configure a subentry password policy that locks accounts idle for more than 13 weeks. This policy extends the example from Track Last Login Time. The DS server must track last successful login time to calculate how long the account has been idle. You must first add the `lastLoginTime` attribute type in order for the DS server to accept this new password policy:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: cn=Lock Idle Accounts,dc=example,dc=com
objectClass: top
objectClass: subentry
```

```
objectClass: ds-pwp-password-policy
cn: Lock Idle Accounts
ds-pwp-password-attribute: userPassword
ds-pwp-default-password-storage-scheme: PBKDF2-HMAC-SHA256
ds-pwp-idle-lockout-interval: 13 w
ds-pwp-last-login-time-attribute: lastLoginTime
ds-pwp-last-login-time-format: yyyyMMddHH'Z'
subtreeSpecification: { base "ou=people" }
EOF
```

Allow Log In to Change an Expired Password

The following commands configure a subentry password policy that lets users log in twice with an expired password to set a new password:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: cn=Allow Grace Login,dc=example,dc=com
objectClass: top
objectClass: subentry
objectClass: ds-pwp-password-policy
cn: Allow Grace Login
ds-pwp-password-attribute: userPassword
ds-pwp-default-password-storage-scheme: PBKDF2-HMAC-SHA256
ds-pwp-grace-login-count: 2
subtreeSpecification: { base "ou=people" }
EOF
```

Require Password Change on Add or Reset

The following commands configure a subentry password policy that requires new users to change their password after logging in for the first time. This policy also requires users to change their password after it is reset:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
  --useSsl \
  --usePkcs12TrustStore /path/to/opendj/config/keystore \
  --trustStorePassword:file /path/to/opendj/config/keystore.pin \
  --bindDN uid=admin \
  --bindPassword password << EOF
dn: cn=Require Password Change on Add or Reset,dc=example,dc=com
objectClass: top
objectClass: subentry
objectClass: ds-pwp-password-policy
cn: Require Password Change on Add or Reset
ds-pwp-password-attribute: userPassword
ds-pwp-default-password-storage-scheme: PBKDF2-HMAC-SHA256
ds-pwp-force-change-on-add: true
ds-pwp-force-change-on-reset: true
subtreeSpecification: { base "ou=people" }
EOF
```

About Password Policies

DS password policies govern passwords, account lockout, and account status notification.

DS servers support per-server password policies stored in the configuration, and subentry password policies stored in the (replicated) directory data:

Type	Notes
Per-Server Password Policies	• Use for default policies, and policies for top-level administrative accounts. • You must manually apply policy updates to each replica server configuration. • Updates require write access to the server configuration.
DS Subentry Password Policies	• Use for all user accounts stored in application data. • Replication applies each policy update to all replicas. • Updates require the `subentry-write` privilege, and ACIs to write the policy.

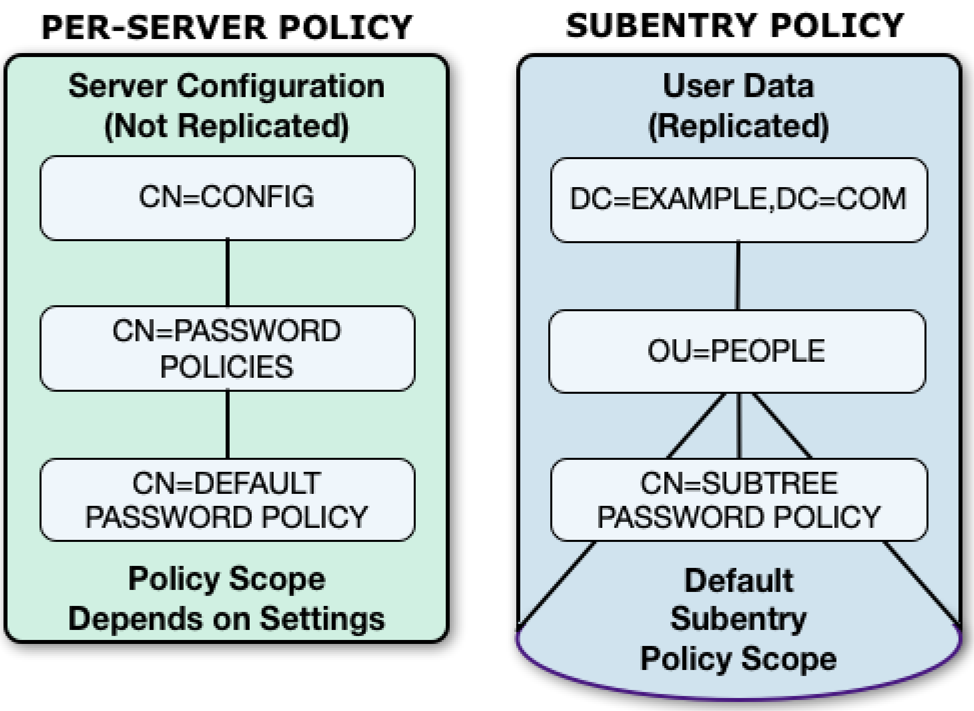

PER-SERVER POLICY
Server Configuration (Not Replicated)
CN=CONFIG
CN=PASSWORD POLICIES
CN=DEFAULT PASSWORD POLICY
Policy Scope Depends on Settings
SUBENTRY POLICY
User Data (Replicated)
DC=EXAMPLE,DC=COM
OU=PEOPLE
CN=SUBTREE PASSWORD POLICY
Default Subentry Policy Scope

Figure 1. Per-Server and Subentry Password Policies

Per-Server Password Policies

You manage per-server password policies with the **`dsconfig`** command. When changing a per-server policy, you must update each replica in your deployment.

By default, there are two per-server password policies:

- The `Default Password Policy` for users.
- The `Root Password Policy` for the directory superuser, `uid=admin`.

The following example displays the default per-server password policy for users:

```
$ dsconfig \
 get-password-policy-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --policy-name "Default Password Policy" \
 --advanced \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt

Property                                   : Value(s)
-------------------------------------------:------------------------
-------------
```

```
account-status-notification-handler        : -
allow-expired-password-changes             : false
allow-multiple-password-values             : false
allow-pre-encoded-passwords                : false
allow-user-password-changes                : true
default-password-storage-scheme            : PBKDF2-HMAC-SHA256
deprecated-password-storage-scheme         : -
expire-passwords-without-warning           : false
force-change-on-add                        : false
force-change-on-reset                      : false
grace-login-count                          : 0
idle-lockout-interval                      : 0 s
java-class                                 :
org.opends.server.core.PasswordPoli
                                           : cyFactory
last-login-time-attribute                  : -
last-login-time-format                     : -
lockout-duration                           : 0 s
lockout-failure-count                      : 0
lockout-failure-expiration-interval        : 0 s
max-password-age                           : 0 s
max-password-reset-age                     : 0 s
min-password-age                           : 0 s
password-attribute                         : userPassword
password-change-requires-current-password : false
password-expiration-warning-interval       : 5 d
password-generator                         : Random Password
Generator
password-history-count                     : 0
password-history-duration                  : 0 s
password-validator                         : At least 8 characters,
Common
                                           : passwords
previous-last-login-time-format            : -
require-change-by-time                     : -
require-secure-authentication              : true
require-secure-password-changes            : true
skip-validation-for-administrators         : false
state-update-failure-policy                : reactive
```

For detailed descriptions of each property, see Password Policy.

These settings are configured by default:

- When granted access, users can change their passwords.

- DS servers use the standard `userPassword` attribute to store passwords.

  DS servers also support the alternative standard `authPassword` attribute.
- When you import LDIF with `userPassword` values, DS servers apply a one-way hash to the passwords before storing them.

  When a user provides a password value during a bind, for example, the server hashes the incoming password, and compares it with the stored value. This mechanism helps prevent even the directory superuser from recovering the plain text password:

  ```
  $ ldapsearch \
   --hostname localhost \
   --port 1636 \
   --useSsl \
   --usePkcs12TrustStore /path/to/opendj/config/keystore \
   --trustStorePassword:file /path/to/opendj/config/keystore.pin
  \
   --bindDN uid=admin \
   --bindPassword password \
   --baseDN dc=example,dc=com \
   "(uid=bjensen)" \
   userpassword

  dn: uid=bjensen,ou=People,dc=example,dc=com
  userpassword: {PBKDF2-HMAC-SHA256}10000:<hash>
  ```
- The server can set a random password when a password administrator resets a user's password.

Many capabilities are not set by default:

- No lockout.
- No password expiration.
- No password validator to check that passwords contain the appropriate mix of characters.

If the directory service enforces password policy, configure at least the default password policy accordingly.

DS Subentry Password Policies

You manage password policies as LDAP subentries in the application data. Replication applies updates to subentry password policies to all other replicas. Password policy administrators do not need access to the server configuration.

The DS subentry password policy entries have the object classes:

- `ds-pwp-password-policy` for most password policy features.
- A set of password validator object classes for specific validators that derive from the abstract `ds-pwp-validator` class for password validation configuration.
- `ds-pwp-random-generator` for password generation on reset.

The following tables describe password policy attributes per object class:

ds-pwp-password-policy Attributes

Attribute	Description
`ds-pwp-password-attribute` (required)	The attribute type used to hold user passwords.
`ds-pwp-default-password-storage-scheme` (required)	Names of enabled password storage schemes used to encode plaintext passwords. Default: PBKDF2-HMAC-SHA256.
`cn`	Name of the password policy
`ds-pwp-allow-user-password-changes`	Whether users can change their passwords, assuming access control allows it. Default: true.
`ds-pwp-account-status-notification-handler`	Names of enabled account status notification handlers to use with this policy. Use the **`dsconfig list-account-status-notification-handlers`** command. The first column of the output shows the names. The third column shows whether the handler is enabled.
`ds-pwp-allow-expired-password-changes`	Whether the user can change an expired password with the password modify extended operation. Default: false.

Attribute	Description
`ds-pwp-allow-multiple-password-values`	Whether user entries can have multiple distinct passwords. Any password is sufficient to authenticate. Default: false.
`ds-pwp-allow-pre-encoded-passwords`	Whether users can change their passwords by providing a pre-encoded value. Default: false.
`ds-pwp-deprecated-password-storage-scheme`	Names of deprecated password storage schemes for this policy. On successful authentication, encode the password with the default.
`ds-pwp-expire-passwords-without-warning`	Whether to allow a user's password to expire even if that user has never seen an expiration warning notification. Default: false.
`ds-pwp-force-change-on-add`	Whether users are forced to change their passwords upon first authentication after their accounts are added. Default: false.
`ds-pwp-force-change-on-reset`	Whether users are forced to change their passwords after password reset by an administrator. For this purpose, anyone with permission to change a given user's password other than that user is an administrator. Default: false.
`ds-pwp-grace-login-count`	Number of grace logins that a user is allowed after the account has expired so the user can update their password. Default: 0 (disabled).

Attribute	Description
`ds-pwp-idle-lockout-interval`	Maximum number of seconds that an account may remain idle (the associated user does not authenticate to the server) before that user is locked out. Requires maintaining a last login time attribute. Default: 0 seconds (inactive).
`ds-pwp-last-login-time-attribute`	Name or OID of the attribute type that is used to hold the last login time for users. Default: The `last-login-time-attribute` setting from the default password policy. By default, `last-login-time-attribute` is not set.
`ds-pwp-last-login-time-format`	Format string that is used to generate the last login time value for users. The format string must match the syntax of the `ds-pwp-last-login-time-attribute` attribute, and must be a valid format string for the `java.text.SimpleDateFormat` class. Default: `yyyyMMddHHmmss'Z'`.
`ds-pwp-lockout-duration`	Duration that an account is locked after too many authentication failures. Default: 0 seconds (account remains locked until the administrator resets the password).
`ds-pwp-lockout-failure-count`	Maximum number of authentication failures that a user is allowed before the account is locked out. Default: 0 (disabled).
`ds-pwp-lockout-failure-expiration-interval`	Duration before an authentication failure is no longer counted against a user for the purposes of account lockout. Default: 0 seconds (never expire).

Attribute	Description
`ds-pwp-max-password-age`	Duration that a user can continue using the same password before it must be changed (the password expiration interval). Default: 0 seconds (passwords never expire).
`ds-pwp-max-password-reset-age`	Maximum number of seconds that users have to change passwords after they have been reset by an administrator before they become locked. Default: 0 seconds.
`ds-pwp-min-password-age`	Minimum duration after a password change before the user is allowed to change the password again. Default: 0 seconds.
`ds-pwp-password-change-requires-current-password`	Whether user password changes must include the user's current password before the change is allowed. This can be done with either the password modify extended operation, or a modify operation using delete and add. Default: false.
`ds-pwp-password-expiration-warning-interval`	Duration before a user's password actually expires that the server begins to include warning notifications in bind responses for that user. Default: 5 days.

Attribute	Description
`ds-pwp-password-history-count`	Maximum number of former passwords to maintain in the password history. A value of zero indicates that either no password history is to be maintained if the password history duration has a value of zero seconds, or that there is no maximum number of passwords to maintain in the history if the password history duration has a value greater than zero seconds. Default: 0.
`ds-pwp-password-history-duration`	Maximum number of seconds that passwords remain in the password history. Default: 0 seconds (inactive).
`ds-pwp-previous-last-login-time-format`	Format string(s) that might have been used with the last login time at any point in the past for users associated with the password policy. Default: `yyyyMMddHHmmss'Z'`.
`ds-pwp-require-change-by-time`	Time by which all users with the associated password policy must change their passwords. Specified in generalized time form.
`ds-pwp-require-secure-authentication`	Whether users with the associated password policy are required to authenticate in a secure manner. Default: false.
`ds-pwp-require-secure-password-changes`	Whether users with the associated password policy are required to change their password in a secure manner that does not expose the credentials. Default: false.

Attribute	Description
`ds-pwp-skip-validation-for-administrators`	Whether passwords set by administrators are allowed to bypass the password validation process. Default: false.
`ds-pwp-state-update-failure-policy`	How the server deals with the inability to update password policy state information during an authentication attempt. One of the following: • `ignore` : If a bind attempt would otherwise be successful, then do not reject it if a problem occurs while attempting to update the password policy state information for the user. • `proactive` : Proactively reject any bind attempt if it is known ahead of time that it would not be possible to update the user’s password policy state information. • `reactive` (default): Even if a bind attempt would otherwise be successful, reject it if a problem occurs while attempting to update the password policy state information for the user.

ds-pwp-attribute-value-validator Attributes

Attribute	Description
`ds-pwp-attribute-value-test-reversed-password`	Whether this password validator should test the reversed value of the provided password as well as the order in which it was given. Default: false.

Attribute	Description
`ds-pwp-attribute-value-match-attribute`	Name(s) of the attribute(s) whose values should be checked to determine whether they match the provided password. If no values are provided, then the server checks if the proposed password matches the value of any user attribute in the user's entry. The server does not check values of operational attributes.

Attribute	Description
`ds-pwp-attribute-value-check-substrings`	Whether this password validator is to match portions of the password string against attribute values. When false, the server checks whether the entire password matches any user attribute values. When true, the server checks whether the password contains any user attribute values. Consider the case of Babs Jensen (`uid: bjensen`) changing her password. The following table describes the effects of the settings:

Setting	New Password	Password Modification Result
`ds-pwp-attribute-value-check-substrings: false`	`bjense`	Success
`ds-pwp-attribute-value-check-substrings: false`	`bjensen`	Failure: 19 (Constraint Violation)
`ds-pwp-attribute-value-check-substrings: false`	`bjensens`	Success

Attribute	Description
	<table><tr><th>Setting</th><th>New Password</th><th>Password Modification Result</th></tr><tr><td>`ds-pwp-attribute-value-check-substrings: true`</td><td>`bjense`</td><td>Success</td></tr><tr><td>`ds-pwp-attribute-value-check-substrings: true`</td><td>`bjensen`</td><td>Failure: 19 (Constraint Violation)</td></tr><tr><td>`ds-pwp-attribute-value-check-substrings: true`</td><td>`bjensens`</td><td>Failure: 19 (Constraint Violation)</td></tr></table> In summary: <ul><li>`bjense` is allowed in both cases because the password does not contain any of the attribute values in Babs's entry.</li><li>`bjensen` is rejected in both cases because the password exactly matches and contains Babs's user ID.</li><li>`bjensens` is allowed when the setting is false because the password does not exactly match, and rejected when the setting is true because the password contains Babs's user ID.</li></ul> Default: false.

Attribute	Description
`ds-pwp-attribute-value-min-substring-length`	The minimal length of the substring within the password when substring checking is enabled. Default: 0.

ds-pwp-character-set-validator Attributes

Attribute	Description
`ds-pwp-character-set-allow-unclassified-characters`	Whether this password validator allows passwords to contain characters outside of any of the user-defined character sets and ranges. Default: false.
`ds-pwp-character-set-min-character-sets`	Minimum number of character sets and ranges that a password must contain. Use in conjunction with optional character sets and ranges (those requiring zero characters). The value must include any mandatory character sets and ranges (those requiring greater than zero characters). This is useful in situations where a password must contain characters from mandatory character sets and ranges, and characters from at least N optional character sets and ranges. For example, it is quite common to require that a password contains at least one non-alphanumeric character as well as characters from two alphanumeric character sets (lower-case, upper-case, digits). In this case, this property should be set to 3.

Attribute	Description
`ds-pwp-character-set-character-set`	A character set containing characters that a password may contain, and a value indicating the minimum number of characters required from that set. Each value must be an integer (indicating the minimum required characters from the set which may be zero, indicating that the character set is optional) followed by a colon and the characters to include in that set. For example, `3:abcdefghijklmnopqrstuvwxyz` indicates that a user password must contain at least three characters from the set of lowercase ASCII letters. Multiple character sets can be defined in separate values, although no character can appear in more than one character set.
`ds-pwp-character-set-character-set-ranges`	A character range containing characters that a password may contain, and a value indicating the minimum number of characters required from that range. Each value must be an integer (indicating the minimum required characters from the range which may be zero, indicating that the character range is optional) followed by a colon and one or more range specifications. A range specification is 3 characters: the first character allowed, a minus, and the last character allowed. For example, `3:A-Za-z0-9`. The ranges in each value should not overlap, and the characters in each range specification should be ordered.

ds-pwp-dictionary-validator Attributes

Attribute	Description

Attribute	Description
`ds-pwp-dictionary-data` (required)	A gzipped password dictionary, one word per line. This is a single-valued attribute.
`ds-pwp-dictionary-case-sensitive-validation`	Whether this password validator should treat password characters in a case-sensitive manner. Default: false.
`ds-pwp-dictionary-check-substrings`	Whether this password validator is to match portions of the password string against dictionary words. Default: false (match only the entire password against dictionary words).
`ds-pwp-dictionary-min-substring-length`	The minimal length of the substring within the password in case substring checking is enabled. Default: 0.
`ds-pwp-dictionary-test-reversed-password`	Whether this password validator should test the reversed value of the provided password as well as the order in which it was given. Default: false.

ds-pwp-length-based-validator Attributes

Attribute	Description
`ds-pwp-length-based-max-password-length`	Minimum plaintext password length. Default: 0 (undefined).
`ds-pwp-length-based-min-password-length`	Minimum plaintext password length. Default: 6.

ds-pwp-repeated-characters-validator Attributes

Attribute	Description

Attribute	Description
`ds-pwp-repeated-characters-max-consecutive-length`	The maximum number of times that any character can appear consecutively in a password value. Default: 0 (no maximum limit is enforced).
`ds-pwp-repeated-characters-case-sensitive-validation`	Whether this password validator should treat password characters in a case-sensitive manner. Default: false.

ds-pwp-similarity-based-validator Attributes

Attribute	Description
`ds-pwp-similarity-based-min-password-difference`	The minimum difference the new and old password. The implementation uses the Levenshtein Distance algorithm to determine the minimum number of changes (where a change may be inserting, deleting, or replacing a character) to transform one string into the other. It can prevent users from making only minor changes to their current password when setting a new password. Note that for this password validator to be effective, it must have access to the user’s current password. Therefore, if this password validator is to be enabled, also set `ds-pwp-password-change-requires-current-password: true`. Default: 0 (no difference between passwords is acceptable).

ds-pwp-unique-characters-validator Attributes

Attribute	Description

Attribute	Description
`ds-pwp-unique-characters-case-sensitive-validation`	Whether this password validator should treat password characters in a case-sensitive manner. Default: false.
`ds-pwp-unique-characters-min-unique-characters`	The minimum number of unique characters that a password will be allowed to contain. Default: 0 (no minimum value is enforced).

ds-pwp-random-generator Attributes

Attribute	Description
`ds-pwp-random-password-character-set` (required)	Named character sets. The format of the character set is the name of the set followed by a colon and the characters that are in that set. For example, the value `alpha:abcdefghijklmnopqrstuvwxyz` defines a character set named `alpha` containing all of the lower-case ASCII alphabetic characters.
`ds-pwp-random-password-format` (required)	The format to use for the generated password. The value is a comma-delimited list of elements in which each of those elements is comprised of the name of a character set defined in the password-character-set property, a colon, and the number of characters to include from that set. For example, a value of `alpha:3,numeric:2,alpha:3` generates an 8-character password in which the first three characters are from the `alpha` set, the next two are from the `numeric` set, and the final three are from the `alpha` set.

Interoperable Subentry Password Policies

DS servers support the Internet-Draft, Password Policy for LDAP Directories (version 09). The password policies are expressed as LDAP subentries with `objectClass: pwdPolicy`. An Internet-Draft password policy effectively overrides settings in the default per-server password policy for users, inheriting settings that it does not support or does not include from the per-server password policy.

The following table describes Internet-Draft policy attributes:

pwdPolicy Attributes

Attribute	Description
`pwdAttribute` (required)	The attribute type used to hold user passwords.
`pwdAllowUserChange`	Whether users can change their passwords. Default: true.
`pwdExpireWarning`	Maximum number of seconds before a user’s password actually expires that the server begins to include warning notifications in bind responses for that user. Default: 432000 seconds.
`pwdFailureCountInterval`	Length of time before an authentication failure is no longer counted against a user for the purposes of account lockout. Default: 0 seconds (never expire).
`pwdGraceAuthNLimit`	Number of grace logins that a user is allowed after the account has expired so the user can update their password. Default: 0 (disabled).
`pwdInHistory`	Maximum number of former passwords to maintain in the password history. Default: 0 (disabled).

Attribute	Description
`pwdLockoutDuration`	Number of seconds that an account is locked after too many authentication failures. Default: 0 seconds (account remains locked indefinitely).
`pwdMaxAge`	Maximum number of seconds that a user can continue using the same password before it must be changed (the password expiration interval). Default: 0 seconds (disabled).
`pwdMaxFailure`	Maximum number of authentication failures that a user is allowed before the account is locked out. Default: 0.
`pwdMinAge`	Minimum number of seconds after a password change before the user is allowed to change the password again. Default: 0 seconds (disabled).
`pwdMustChange`	Whether users are forced to change their passwords after password reset by an administrator. Default: false.
`pwdSafeModify`	Whether user password changes must use the password modify extended operation, and must include the user's current password before the change is allowed. Default: false.

The following table lists Internet-Draft policy attributes that override the per-server policy properties:

Internet-Draft Policy Attribute	Overrides This Server Policy Property
`pwdAllowUserChange`	`allow-user-password-changes`

Internet-Draft Policy Attribute	Overrides This Server Policy Property
`pwdMustChange`	`force-change-on-reset`
`pwdGraceAuthNLimit`	`grace-login-count`
`pwdLockoutDuration`	`lockout-duration`
`pwdMaxFailure`	`lockout-failure-count`
`pwdFailureCountInterval`	`lockout-failure-expiration-interval`
`pwdMaxAge`	`max-password-age`
`pwdMinAge`	`min-password-age`
`pwdAttribute`	`password-attribute`
`pwdSafeModify`	`password-change-requires-current-password`
`pwdExpireWarning`	`password-expiration-warning-interval`
`pwdInHistory`	`password-history-count`

DS servers *ignore* the following Internet-Draft password policy attributes:

- `pwdCheckQuality` , because DS servers have password validators.
- `pwdMinLength` , because you can use a length-based password validator instead.
- `pwdLockout` , because DS servers use other lockout-related password policy attributes.

Internet-Draft based password policies inherit these settings from the default per-server policy for users:

- `account-status-notification-handlers`
- `allow-expired-password-changes`
- `allow-multiple-password-values`
- `allow-pre-encoded-passwords`
- `default-password-storage-schemes`
- `deprecated-password-storage-schemes`
- `expire-passwords-without-warning`
- `force-change-on-add`

- `idle-lockout-interval`
- `last-login-time-attribute`
- `last-login-time-format`
- `max-password-reset-age`
- `password-generator`
- `password-history-duration`
- `password-validators`
- `previous-last-login-time-formats`
- `require-change-by-time`
- `require-secure-authentication`
- `require-secure-password-changes`
- `skip-validation-for-administrators`
- `state-update-failure-policy`

Administrative Roles

The server setup process creates one directory superuser account. The directory superuser has unrestricted access to manage the directory service and data.

For any directory service with more than one administrator, one account is not enough. Instead, grant appropriate access to each administrator, based on their duties.

Administrative Access

Only the directory superuser should have all the default access and privileges of that account. Directory service administrators should have limited access, as outlined in the following table:

Tasks	**Required Access(1) and Privileges(2)**
Install and upgrade servers	Access to file system. Access to run server commands.
Delegate administration	Access to write administration-related attributes on others' entries. DS server privileges: `config-read`, `config-write`, `modify-acl`, `privilege-change`.

Tasks	Required Access$^{(1)}$ and Privileges$^{(2)}$
Manage server processes (start, restart, stop)	Access to run the **`start-ds`** and **`stop-ds`** commands. DS server privileges: `server-restart`, `server-shutdown`.
Manage changes to server configuration, including global and default settings	Access to read and write to `cn=config`, `cn=schema`, and potentially other administrative DNs, such as `cn=tasks`. DS server privileges: `config-read`, `config-write`, `modify-acl` (for global ACIs), `update-schema`.
Manage containers for user data, including backends and indexes	File system access for backup data and exported LDIF. Access to create entries under `cn=tasks`. DS server privileges: `backend-backup`, `backend-restore`, `config-read`, `config-write`, `ldif-export`, `ldif-import`, `modify-acl` (for ACIs in user data), `subentry-write`.
Manage changes to server schemas	Access to write to `cn=schema`. File system access to add schema files. DS server privilege: `update-schema`.
Manage directory server data replication	File system access to read `logs/replication`. Access to read and write to `cn=admin data`, if any password policies configure a reversible password storage scheme. Access to run the **`dsrepl`** command. DS server privileges: `changelog-read`, `config-read`, `data-sync`.
Monitor the directory service	Access to read `cn=monitor`. DS server privileges: `monitor-read`, `jmx-notify`, `jmx-read`, `jmx-write` (the last three being useful when using JMX for monitoring).

Tasks	Required Access[1] and Privileges[2]
Back up and restore directory data	File system access for backup data and exported LDIF. Access to create entries under `cn=tasks`. DS server privileges: `backend-backup`, `backend-restore`, `ldif-export`, `ldif-import`.
Troubleshoot problems with the directory service	File system access to read log messages. Write access to create entries under `cn=tasks`. Access to read `cn=monitor`. DS server privileges: `bypass-lockdown`, `cancel-request`, `config-read`, `config-write` (to enable debug logging, for example), `disconnect-client`, `monitor-read`, `server-lockdown`.

(1) Access control is covered in Access Control.

(2) Privileges are covered in Administrative Privileges.

Directory data administrators should have limited access, as outlined in the following table:

Tasks	Required Access[1] and Privileges[2]
Manage changes to users, groups, and other accounts for their organizations	Access to read and write to others' entries.
Delegate administration within their organizations	Access to write administration-related attributes on others' entries. DS server privileges: `modify-acl`, `privilege-change`.
Update administrative user data, such as subentry password policies and access controls	Access to write administration-related attributes on others' entries. DS server privileges: `modify-acl`, `subentry-write`.

Tasks	Required Access[1] and Privileges[2]
Help users who are locked out, or have forgotten or lost their password	Access to use the **`manage-account`** command. Access to request a password modify extended operation. Access to update passwords on user entries. DS server privilege: `password-reset`.
Assist users and application developers who access the directory service	Access to read (and potentially write to) others' entries. If performing operations on behalf of other users, access to request proxied authorization. DS server privilege: `proxied-auth`.

[1] Access control is covered in Access Control.

[2] Privileges are covered in Administrative Privileges.

Administrative Accounts

TIP

Limit use of the `uid=admin` superuser account.

To bootstrap the system, the default directory superuser account is not subject to access control. It has privileges to perform almost every administrative operation, including increasing its own privileges.

Treat this account as you would the UNIX `root` account, or the Windows `Administrator` account. Use it only when you must.

Use a Non-Default Superuser Account

The following steps replace the directory superuser account:

1. Create an administrator account that duplicates the default directory superuser account.

   ▼ *Show details*

The default administrator account is `uid=admin`. It is stored in its own backend, `rootUser`. The `rootUser` LDIF backend holds the file `db/rootUser.ldif`.

The following example shows the LDIF for an alternative directory superuser:

```
dn: uid=altadmin
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
cn: Alternative Superuser
givenName: Alternative
sn: Superuser
ds-rlim-size-limit: 0
ds-rlim-time-limit: 0
ds-rlim-idle-time-limit: 0
ds-rlim-lookthrough-limit: 0
ds-rlim-cursor-entry-limit: 100000
ds-pwp-password-policy-dn: cn=Root Password
Policy,cn=Password Policies,cn=config
ds-privilege-name: bypass-lockdown
ds-privilege-name: bypass-acl
ds-privilege-name: modify-acl
ds-privilege-name: config-read
ds-privilege-name: config-write
ds-privilege-name: ldif-import
ds-privilege-name: ldif-export
ds-privilege-name: backend-backup
ds-privilege-name: backend-restore
ds-privilege-name: server-lockdown
ds-privilege-name: server-shutdown
ds-privilege-name: server-restart
ds-privilege-name: disconnect-client
ds-privilege-name: cancel-request
ds-privilege-name: password-reset
ds-privilege-name: update-schema
ds-privilege-name: privilege-change
ds-privilege-name: unindexed-search
ds-privilege-name: subentry-write
ds-privilege-name: changelog-read
ds-privilege-name: monitor-read
uid: altadmin
userPassword: password
```

Do not use `altadmin`, since it shows up here in the documentation.

2. Create a private backend to store the new directory superuser account.

   ▼ *Show details*

   The following example creates an LDIF backend to store the entry. Before creating the backend, create a separate directory to hold the backend files:

```
$ mkdir /path/to/opendj/db/altRootUser

$ dsconfig \
 create-backend \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --backend-name altRootUser \
 --type ldif \
 --set enabled:true \
 --set base-dn:uid=altadmin \
 --set ldif-file:db/altRootUser/altRootUser.ldif \
 --set is-private-backend:true \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

3. Import the new directory superuser entry into the new backend:

   ▼ *Show details*

```
$ import-ldif \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --backendID altRootUser \
 --ldifFile /tmp/alt-root.ldif \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin
```

4. Remove the backend database and the unused LDIF files for the default superuser account:

   ▼ *Show details*

```
$ dsconfig \
 delete-backend \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --backend-name rootUser \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt

$ rm -rf /path/to/opendj/db/rootUser
```

In this example, `uid=altadmin` now has the same rights as the original superuser.

5. Repeat these steps for other DS servers.

   The account is not replicated by default.

Make Users Administrators

You can assign access and privileges to any directory account:

1. Adjust the resource limits to as needed.

   For details, see Resource Limits.
2. Assign any necessary administrative privileges.

   For details, see Administrative Privileges.
3. Add any necessary ACIs.

   For details, see Access Control.

Administrative Privileges

Privileges provide access control for server administration independently from ACIs. The default directory superuser, `uid=admin`, is granted the privileges marked with an asterisk (*):

Privilege	Description
`backend-backup` *	Request a task to back up data, or to purge backup files.
`backend-restore` *	Request a task to restore data from backup.
`bypass-acl` *	Perform operations without regard to ACIs.
`bypass-lockdown` *	Perform operations without regard to lockdown mode.
`cancel-request` *	Cancel any client request.
`changelog-read` *	Read the changelog (under `cn=changelog`).
`config-read` *	Read the server configuration.
`config-write` *	Change the server configuration.
`data-sync`	Perform data synchronization.
`disconnect-client` *	Close any client connection.
`jmx-notify`	Subscribe to JMX notifications.
`jmx-read`	Read JMX attribute values.
`jmx-write`	Write JMX attribute values.
`ldif-export` *	Export data to LDIF.
`ldif-import` *	Import data from LDIF.
`modify-acl` *	Change ACIs.
`monitor-read` *	Read metrics under `cn=monitor` , `/metrics/api` , `/metrics/prometheus` , and over JMX.
`password-reset` *	Reset other users' passwords.
`privilege-change` *	Change the privileges assigned to users, including their own privileges.
`proxied-auth`	Use the LDAP proxied authorization control.
`server-lockdown` *	Put the server into and take the server out of lockdown mode.

Privilege	Description
`server-restart` *	Request a task to restart the server.
`server-shutdown` *	Request a task to stop the server.
`subentry-write` *	Perform LDAP subentry write operations.
`unindexed-search` *	Search using a filter with no corresponding index.
`update-schema` *	Change LDAP schema definitions.

Assign Individual Account Privileges

Specify privileges as values of the `ds-privilege-name` operational attribute:

1. Determine the privileges to add.

   Kirsten Vaughan has access to modify user entries. Kirsten lacks privileges to read the server configuration, and reset user passwords:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN "uid=kvaughan,ou=people,dc=example,dc=com" \
 --bindPassword bribery \
 --baseDN cn=config \
 "(objectclass=*)"

# The LDAP search request failed: 50 (Insufficient Access
Rights)
# Additional Information:  You do not have sufficient
privileges to perform search operations in the Directory
Server configuration

$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN "uid=kvaughan,ou=people,dc=example,dc=com" \
 --bindPassword bribery \
 --authzID "dn:uid=scarter,ou=People,dc=example,dc=com" \
 --newPassword chngthspwd

The LDAP password modify operation failed: 50
(Insufficient Access Rights)
Additional Information:  You do not have sufficient
privileges to perform
password reset operations
```

2. Apply the change as a user with the `privilege-change` privilege, and give Kirsten access to read `cn=config`:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: uid=kvaughan,ou=People,dc=example,dc=com
changetype: modify
add: ds-privilege-name
ds-privilege-name: config-read
ds-privilege-name: password-reset
EOF

$ dsconfig \
 set-access-control-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --add "global-aci:(target=\"ldap:///cn=config\")
(targetattr=\"*||+\")\
(version 3.0; acl \"Config read for Kirsten Vaughan\";
allow (read,search,compare)\
userdn=\"ldap:///uid=kvaughan,ou=People,dc=example,dc=com\
";)" \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --no-prompt
```

Kirsten can perform the operations now:

▼ *Show details*

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN "uid=kvaughan,ou=people,dc=example,dc=com" \
 --bindPassword bribery \
 --baseDN cn=config \
 "(objectclass=*)"

dn: cn=config

$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN "uid=kvaughan,ou=people,dc=example,dc=com" \
 --bindPassword bribery \
 --authzID "dn:uid=scarter,ou=People,dc=example,dc=com" \
 --newPassword chngthspwd

The LDAP password modify operation was successful
```

Assign Group Privileges

Use a collective attribute subentry to assign privileges to a group:

1. Create an LDAP subentry that specifies the collective attributes:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: cn=Administrator Privileges,dc=example,dc=com
objectClass: collectiveAttributeSubentry
objectClass: extensibleObject
objectClass: subentry
objectClass: top
cn: Administrator Privileges
ds-privilege-name;collective: config-read
ds-privilege-name;collective: config-write
ds-privilege-name;collective: ldif-export
ds-privilege-name;collective: modify-acl
ds-privilege-name;collective: password-reset
ds-privilege-name;collective: proxied-auth
subtreeSpecification: {base "ou=people",
specificationFilter
  "(isMemberOf=cn=Directory
Administrators,ou=Groups,dc=example,dc=com)" }
EOF
```

For details, see Collective Attributes, and About Subentry Scope.

2. The change takes effect immediately:

▼ *Show details*

```
$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN "uid=hmiller,ou=people,dc=example,dc=com" \
 --bindPassword hillock \
 --authzID "dn:uid=scarter,ou=People,dc=example,dc=com"
```

```
The LDAP password modify operation was successful
Generated Password:  <password>
```

Limit Inherited Privileges

Privileges assigned by collective attributes are inherited by every target account. To limit effective privileges, override the privilege in the account by preceding the privilege attribute value with a `-`.

This examples shows how to prevent Kirsten Vaughan from using the privilege to reset passwords:

1. Check the privilege settings for the account:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password \
 --baseDN dc=example,dc=com \
 "(uid=kvaughan)" \
 ds-privilege-name

dn: uid=kvaughan,ou=People,dc=example,dc=com
ds-privilege-name: config-read
ds-privilege-name: password-reset
```

2. Use the override to deny the privilege:

```
$ ldapmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN uid=admin \
 --bindPassword password << EOF
dn: uid=kvaughan,ou=people,dc=example,dc=com
```

```
changetype: modify
add: ds-privilege-name
ds-privilege-name: -password-reset
EOF
```

3. The change takes effect immediately:

▼ *Show details*

```
$ ldappasswordmodify \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file
/path/to/opendj/config/keystore.pin \
 --bindDN "uid=kvaughan,ou=people,dc=example,dc=com" \
 --bindPassword bribery \
 --authzID "dn:uid=scarter,ou=People,dc=example,dc=com"

The LDAP password modify operation failed: 50
(Insufficient Access Rights)
Additional Information:  You do not have sufficient
privileges to perform
password reset operations
```

Identity Management

DS software lets you manage user and administrator accounts individually or in groups. If the deployment calls for provisioning and workflow capabilities, or custom tools, then consider using identity management software, such as ForgeRock Identity Management. For details, see the IDM documentation.

Simple or local deployments might require a GUI for generic or system-specific administrative operations. DS software's support for standards like LDAP v3 makes it interoperable with many third-party tools.

Access Control

The DS evaluation setup profile leaves access more open, especially to sample Example.com data. This makes it easy to demonstrate and learn features before you fully understand access control. When deploying DS servers in production, grant only the necessary access.

Access Control Mechanisms

DS servers support two access control mechanisms, ACIs for directory servers, and global access control policies for proxy servers.

Characteristic	**Access Control Instructions (ACIs)**	**Global Access Control Policies**
Default for	Directory Servers.	Directory Proxy Servers.
Where	Operational `aci` attributes (replicated). `global-aci` properties (not replicated).	`global-access-control-policy` entries (not replicated).
Default access	No access unless explicitly granted.[(1)]	No access unless explicitly granted.
Level of control	Very fine-grained control that can depend on the directory data.	Overall control that does not require access to directory data.
Interaction[(2)]	When configured, global policies have no effect.	When configured, ACIs have no effect.
Reference	Directory Server ACIs. DSEE Compatible Access Control Handler.	Global Access Control Policy.

[(1)] The `bypass-acl` privilege grants users access regardless of ACIs.

[(2)] In the rare event that you choose to change the type of server and the type of its access control handler, you must stop the server and make the change with the **`dsconfig --offline`** command.

Some operations require administrative privileges *and* access control. By combining access control and privileges, you effectively restrict the scope of the privileges. Privileges are described in Administrative Roles.

Directory Server ACIs

- ACIs set scoped permissions which depend on what operation is requested, who requested the operation, and how the client connected to the server.

- To let other users change ACIs, grant them the `modify-acl` privilege and permission to edit `aci` attributes.

  For examples, see Learn Access Control.

ACI Syntax

```
targets (version 3.0; acl "name"; permissions subjects;)
```

targets
: The ACI applies to the target entries, attributes, controls, and extended operations.

  To define multiple *targets*, put each target in parentheses, `()`. All targets must match for the ACI to apply (`AND`).

name
: Human-readable description of what the ACI does.

permissions
: Actions to allow, and which to deny.

  Paired with subjects.

subjects
: Clients the permissions apply to, and the conditions under which they apply.

  Paired with permissions.

  Separate multiple permissions-subjects pairs with semicolons, `;`. At least one must match for the ACI to apply (`OR`).

ACI Targets

Most target expressions let you use either `=` (target must match), or `!=` (target must not match):

`(target [!]= "ldap:///DN")`
: The ACI scope is the entry with distinguished name DN, and subordinates.

  Use an asterisk, `\*`, to replace attribute types, attribute values, and entire DN components. The following example targets `uid=bjensen,ou=People,dc=example,dc=com` and `cn=My App,ou=Apps,dc=example,dc=com`:

  ```
  (target = "ldap:///*=*,*,dc=example,dc=com")
  ```

The DN must be in the subtree of the entry where the ACI is defined.

If you omit `target`, the ACI applies to its entry.

If you omit `targetscope` as well, the ACI applies to its entry and all subordinates.

(targetattr [!]= "attr-list")

The ACI targets the specified attributes.

In the attr-list, separate attribute names with `||`.

This ACI affects its entry, or the entries specified by other targets in the ACI.

For best performance, explicitly list attributes. Use an asterisk, `\*`, to specify all user attributes. Use a plus sign, `\+`, to specify all operational attributes.

A negated attr-list of operational attributes matches only other operational attributes, never any user attributes, and vice-versa.

If you omit `targetattr`, by default this ACI does not affect attributes.

(targetfilter [!]= "ldap-filter")

This ACI is scoped to match the ldap-filter dynamically, as in an LDAP search. The ldap-filter can be any valid LDAP filter.

(targattrfilters = "expression")

Use this target specification when managing changes made to particular attributes.

The expression takes one of the following forms. Separate expressions with commas (,):

```
op=attr1:filter1[&& attr2:filter2 ...][,op=attr3:filter3[&&
attr4:filter4 ...] ...]
```

The op can be either `add` for operations creating attributes, or `del` for operations removing them.

Replace attr with an attribute type. Replace filter with an LDAP filter that corresponds to the attr attribute type.

(targetscope = "base|onelevel|subtree|subordinate")

- `base` refers to the entry with the ACI.
- `onelevel` refers to immediate children.
- `subtree` refers to the base entry and all children.
- `subordinate` refers to all children only. If you omit `targetscope`, the default is `subtree`.

(targetcontrol [!]= "alias-or-OID")

The ACI targets the LDAP control with the specified alias or object identifier alias-or-OID. Separate multiple aliases or OIDs with `||`.

DS servers support the following control aliases for ACIs:

- `AccountUsable`, `AccountUsability` (`1.3.6.1.4.1.42.2.27.9.5.8`)
- `ActiveDirectoryChangeNotification`, `AdChangeNotification` (`1.2.840.113556.1.4.528`)
- `Affinity` (`1.3.6.1.4.1.36733.2.1.5.2`)
- `Assertion`, `LdapAssertion` (`1.3.6.1.1.12`)
- `AuthorizationIdentity`, `AuthzId` (`2.16.840.1.113730.3.4.16`)
- `Csn`, `ChangeNumber`, `ChangeSequenceNumber` (`1.3.6.1.4.1.42.2.27.9.5.9`)
- `Ecl`, `EclCookie`, `ExternalChangelogCookie` (`1.3.6.1.4.1.26027.1.5.4`)
- `EffectiveRights`, `GetEffectiveRights` (`1.3.6.1.4.1.42.2.27.9.5.2`)
- `ManageDsaIt` (`2.16.840.1.113730.3.4.2`)
- `MatchedValues` (`1.2.826.0.1.3344810.2.3`)
- `NoOp` (`1.3.6.1.4.1.4203.1.10.2`)
- `PasswordPolicy`, `PwdPolicy`, `PwpPolicy` (`1.3.6.1.4.1.42.2.27.8.5.1`)
- `PasswordQualityAdvice` (`1.3.6.1.4.1.36733.2.1.5.5`)
- `PermissiveModify` (`1.2.840.113556.1.4.1413`)
- `PersistentSearch`, `PSearch` (`2.16.840.1.113730.3.4.3`)
- `PostRead` (`1.3.6.1.1.13.2`)
- `PreRead` (`1.3.6.1.1.13.1`)
- `ProxiedAuthV1` (`2.16.840.1.113730.3.4.12`)
- `ProxiedAuthV2`, `ProxiedAuth` (`2.16.840.1.113730.3.4.18`)
- `RealAttrsOnly`, `RealAttributesOnly` (`2.16.840.1.113730.3.4.17`)
- `RelaxRules` (`1.3.6.1.4.1.4203.666.5.12`)
- `ReplicationRepair` (`1.3.6.1.4.1.26027.1.5.2`)
- `ServerSideSort`, `Sort` (`1.2.840.113556.1.4.473`)
- `SimplePagedResults`, `PagedResults` (`1.2.840.113556.1.4.319`)
- `SubEntries` (`1.3.6.1.4.1.4203.1.10.1`)
- `SubtreeDelete`, `TreeDelete` (`1.2.840.113556.1.4.805`)
- `TransactionId`, `TxnId` (`1.3.6.1.4.1.36733.2.1.5.1`)
- `VirtualAttrsOnly`, `VirtualAttributesOnly` (`2.16.840.1.113730.3.4.19`)

- `Vlv`, `VirtualListView` (`2.16.840.1.113730.3.4.9`)

To use an LDAP control, the bind DN user must have `allow(read)` permissions. This target cannot be restricted to a specific subtree.

(extop [!]= "alias-or-OID")

This ACI targets the LDAP extended operation with the specified alias or object identifier alias-or-OID. Separate multiple aliases or OIDs with `||`.

DS servers support the following extended operation aliases for ACIs:

- `Cancel` (`1.3.6.1.1.8`)
- `GetConnectionId`, `ConnectionId` (`1.3.6.1.4.1.26027.1.6.2`)
- `GetSymmetricKey`, `SymmetricKey` (`1.3.6.1.4.1.26027.1.6.3`)
- `PasswordModify` (`1.3.6.1.4.1.4203.1.11.1`)
- `PasswordPolicyState` (`1.3.6.1.4.1.26027.1.6.1`)
- `StartTls` (`1.3.6.1.4.1.1466.20037`)
- `WhoAmI` (`1.3.6.1.4.1.4203.1.11.3`)

To use an LDAP extended operation, the bind DN user must have `allow(read)` permissions. This target cannot be restricted to a specific subtree.

ACI Permissions

ACI permission definitions take one of the following forms:

```
allow(action[, action ...])
deny(action[, action ...])
```

TIP

Avoid using `deny`.

Instead, explicitly `allow` access only as needed. What looks harmless and simple in tests and examples can grow complicated quickly with nested ACIs.

The action is one of the following:

add

Entry creation, as for an LDAP add operation.

all

All permissions, except `export`, `import`, `proxy`.

compare

Attribute value comparison, as for an LDAP compare operation.

delete
: Entry deletion, as for an LDAP delete operation.

export
: Entry export during a modify DN operation.

  Despite the name, this action is unrelated to LDIF export operations.

import
: Entry import during a modify DN operation.

  Despite the name, this action is unrelated to LDIF import operations.

proxy
: Access the ACI target using the rights of another user.

read
: Read entries and attributes, or use an LDAP control or extended operation.

search
: Search the ACI targets.

  Combine with `read` to read the search results.

selfwrite
: Add or delete own DN from a group.

write
: Modify attributes on ACI target entries.

ACI Subjects

Subjects restrict whether the ACI applies depending on who connected, and when, where, and how they connected.

Most target expressions allow you to use either `=` (condition must match), or `!=` (condition must not match):

authmethod [!]= "none|simple|ssl|sasl mech"

- `none` : ignore the authentication method.
- `simple` : simple authentication.
- `ssl` : certificate-based authentication over LDAPS.
- `sasl` *mech* : SASL authentication, where *mech* is the SASL mechanism, such as `EXTERNAL` , or `GSSAPI` .

dayofweek [!]= "day[, day…]"
: Valid days:

- `sun` (Sunday)
- `mon` (Monday)
- `tue` (Tuesday)
- `wed` (Wednesday)
- `thu` (Thursday)
- `fri` (Friday)
- `sat` (Saturday)

`dns [!]= "hostname"`
Use an asterisk, `*`, to replace name components, as in `dns = "*.example.com"`.

`groupdn [!]= "ldap:///DN [|| ldap:///DN …]"`
The subjects are the members of the group with the specified DN.

`ip [!]= "addresses"`
Valid IP addresses:

- Individual IPv4 or IPv6 addresses.

  Put IPv6 addresses in brackets, as in `ldap://[`*`address`*`]/`*`subnet-prefix`*, where `/`*`subnet-prefix`* is optional.
- Addresses with asterisk (`*`) for a subnet or host number.
- CIDR notation.
- Forms such as `192.168.0.*+255.255.255.0` to specify subnet masks.

`ssf = "strength"`
The security strength factor (`ssf`) reflects the cipher key strength for a secure connection.

The `ssf` takes an integer in the range 0-1024:

- `ssf = 0`: send plain text with no connection security.
- `ssf = 1`: configure TLS without a cipher. The server verifies integrity using packet checksums, but all content is sent in cleartext.
- `ssf >= "256"`: require a cipher strength of at least 256 bits.

The `ssf` setting can help to neutralize STRIPTLS attacks. A TLS stripping attack is a man-in-the-middle attack. It takes advantage of the fact that the initial TLS handshake starts on an unencrypted connection. An attacker who has control of the network makes it appear during the handshake that TLS is not available. Client applications may then fall back to using the connection without TLS encryption. In this case, ACIs with `ssf` settings greater than `1` require encryption to grant access. Use an appropriately high `ssf` setting in your ACIs, such as `ssf >= "256"` to ensure secure encryption.

`timeofday = "hhmm"`

Express times, *hhmm*, as on a 24-hour clock.

For example, 1:15 PM is written `1315`.

`userattr [!]= "attr#value"`

The `userattr` subject specifies an attribute that must match on the bind entry and the ACI target entry:

- Use `userattr [!]= "`*`attr#value`*`"` when the bind entry and target entry have the same attribute. The attr is a user attribute. The value is the attribute value.

  The server does an internal search to get the attributes of the bind entry. Therefore, this ACI subject does not work with operational attributes.

- Use `userattr [!]=` *`ldap-url`*`#LDAPURL"` when the target entry is identified by the LDAP URL, and the bind entry is in the subtree scope of the DN in the LDAP URL.

- Use `userattr [!]= "[parent[`*`child-level`*`].]`*`attr`*`#GROUPDN"` when the bind DN is a member of the group identified by the attr of the target entry.

- Use `userattr [!]= "[parent[`*`child-level`*`].]`*`attr`*`#USERDN"` when the bind DN is referenced by the attr of the target entry.

The optional inheritance specification, `parent[`*`child-level`*`].`, defines how many levels below the target entry inherit the ACI. The child-level is a number from 0 to 9, with 0 indicating the target entry only. Separate multiple child-level digits with commas (`,`).

`userdn [!]= "ldap-url+_[|| _ldap-url+ …]"`

This subject matches either a valid LDAP URL, or a special LDAP URL-like keyword from the following list:

`ldap:///all`

Match authenticated users.

`ldap:///anyone`

Match anonymous and authenticated users.

`ldap:///parent`

Match when the bind DN is a parent of the ACI target.

`ldap:///self`

Match when the bind DN entry corresponds to ACI target.

ACI Evaluation

The rules the server follows are simple:

1. To determine whether an operation is allowed or denied, DS servers look in the directory for the target of the operation. The server collects any ACI values from that entry, and then walks up the directory tree to the base DN, collecting all ACI values en route. It then collects global ACI values.
2. The server separates the ACI values into two lists. One list contains all the ACI values that match the target and deny the required access. The other list contains all the ACI values that match the target and allow the required access.
3. If the deny list contains any ACI values after this procedure, access is immediately denied.
4. If the deny list is empty, the server processes the allow list. If the allow list contains any ACI values, access is allowed.
5. If both lists are empty, access is denied.

NOTE

Some operations require multiple permissions and involve multiple targets. Evaluation therefore takes place multiple times.

For example, a search operation requires the `search` permission for each attribute in the search filter. If applicable ACIs allow all search permissions, the server uses `read` permissions to decide which attributes and values to return.

ACI by Operation

Add

The ACI must allow the `add` permission to entries in the target. This implicitly lets users set attributes and values.

Use `targattrfilters` to explicitly deny access to any values if required.

For example, this ACI lets `uid=bjensen,ou=People,dc=example,dc=com` add an entry:

```
aci: (version 3.0;acl "Add entry"; allow (add)
 (userdn = "ldap:///uid=bjensen,ou=People,dc=example,dc=com");)
```

Bind

Because a bind establishes the user’s identity and derived authorizations, ACI is irrelevant for this operation and is not checked.

To prevent authentication, disable the account instead.

Compare

The ACI must allow the `compare` permission to the attribute in the target entry.

For example, this ACI lets `uid=bjensen,ou=People,dc=example,dc=com` compare values against the `sn` attribute:

```
aci: (targetattr = "sn")(version 3.0;acl "Compare surname";
allow (compare)
 (userdn = "ldap:///uid=bjensen,ou=People,dc=example,dc=com");)
```

Delete

The ACI must allow the `delete` permission to the target entry. This implicitly lets users delete attributes and values in the target.

Use `targattrfilters` to explicitly deny access to the values if required.

For example, this ACI lets `uid=bjensen,ou=People,dc=example,dc=com` delete an entry:

```
aci: (version 3.0;acl "Delete entry"; allow (delete)
 (userdn = "ldap:///uid=bjensen,ou=People,dc=example,dc=com");)
```

Modify

The ACI must allow the `write` permission to attributes in the target entries. This implicitly lets users modify all values of the target attribute.

Use `targattrfilters` to explicitly deny access to specific values if required.

For example, this ACI lets `uid=bjensen,ou=People,dc=example,dc=com` modify the `description` attribute in an entry:

```
aci: (targetattr = "description")(version 3.0; acl "Modify
description";
 allow (write) (userdn =
"ldap:///uid=bjensen,ou=People,dc=example,dc=com");)
```

ModifyDN

If the entry is being moved to a `newSuperior`, the `export` permission must be allowed on the target, and the `import` permission must be allowed on the `newSuperior` entry.

The ACI must allow `write` permission to the attributes in the old RDN and the new RDN. This implicitly lets users write all values of the old RDN and new RDN.

Use `targattrfilters` to explicitly deny access to values used if required.

For example, this ACI lets `uid=bjensen,ou=People,dc=example,dc=com` rename entries named with the `uid` attribute to new locations:

```
aci: (targetattr = "uid")(version 3.0;acl "Rename uid=
entries";
 allow (write, import, export)
 (userdn = "ldap:///uid=bjensen,ou=People,dc=example,dc=com");)
```

Search

ACI is required to process the search filter, and to determine which attributes and values the server returns. The `search` permission allows particular attributes in the search filter. The `read` permission allows particular attributes to be returned.

If `read` permission is allowed to any attribute, the server automatically allows reads of the `objectClass` attribute.

For example, this ACI lets `uid=bjensen,ou=People,dc=example,dc=com` search for `uid` attributes, and read that attribute in matching entries:

```
aci: (targetattr = "uid")(version 3.0;acl "Search and read
uid";
 allow (search, read) (userdn =
"ldap:///uid=bjensen,ou=People,dc=example,dc=com");)
```

Use Control or Extended Operation

The ACI must allow the `read` permission to the `targetcontrol` or `extop` OIDs.

For example, this ACI lets `uid=bjensen,ou=People,dc=example,dc=com` use the Persistent Search request control with OID `2.16.840.1.113730.3.4.3`:

```
aci: (targetcontrol = "PSearch")
 (version 3.0;acl "Request Persistent Search"; allow (read)
 (userdn = "ldap:///uid=bjensen,ou=People,dc=example,dc=com");)
```

Default Global ACIs

Modifying and removing global ACIs can have deleterious effects. Modifications to global ACIs fall into the following categories:

- Modification or removal is permitted.

  You must test client applications when deleting the specified ACI.

- Modification or removal may affect applications.

  You must test client applications when modifying or deleting the specified ACI.

- Modification or removal may affect applications, but is not recommended.

You must test client applications when modifying or deleting the specified ACI.

- Do not modify or delete.

Name	Description	ACI Definition
Anonymous extended operation access	Anonymous and authenticated users can request the LDAP extended operations that are specified by OID or alias. Modification or removal may affect applications.	`(extop="Cancel||GetSymmetricKey||PasswordModify||StartTls||WhoAmI") (version 3.0; acl "Anonymous extended operation access"; allow(read) userdn="ldap:///anyone";)`
Anonymous extended operation access	Anonymous and authenticated users can request the LDAP extended operations that are specified by OID or alias. Modification or removal may affect applications.	`(targetcontrol="Assertion||AuthorizationIdentity||MatchedValues||NoOp||PasswordPolicy||PasswordQualityAdvice||PermissiveModify||PostRead||PreRead||RealAttrsOnly||SimplePagedResults||TransactionId||VirtualAttrsOnly||Vlv") (version 3.0; acl "Anonymous extended operation access"; allow(read) userdn="ldap:///anyone";)`
Authenticated users extended operation access	Authenticated users can request the LDAP extended operations that are specified by OID or alias. Modification or removal may affect applications.	`(targetcontrol="ManageDsaIt||RelaxRules||ServerSideSort||SubEntries||SubtreeDelete") (version 3.0; acl "Authenticated users extended operation access"; allow(read) userdn="ldap:///all";)`

Name	Description	ACI Definition
Authenticated users extended operation access	Authenticated users can request the LDAP extended operations that are specified by OID or alias. Modification or removal may affect applications.	`(extop="PasswordPolicyState") (version 3.0; acl "Authenticated users extended operation access"; allow(read) userdn="ldap:///all";)`
User-Visible Monitor Attributes	Authenticated users can read monitoring information if they have the monitor read privilege. Modification or removal may affect applications.	`(target="ldap:///cn=monitor") (targetattr="*||+") (version 3.0; acl "User-Visible Monitor Attributes"; allow (read,search,compare) userdn="ldap:///all";)`
User-Visible Root DSE Operational Attributes	Anonymous and authenticated users can read attributes that describe what the server supports. Modification or removal may affect applications.	`(target="ldap:///") (targetscope="base") (targetattr="objectClass||namingContexts||subSchemaSubEntry||supportedAuthPasswordSchemes||supportedControl||supportedExtension||supportedFeatures||supportedLDAPVersion||supportedSASLMechanisms||supportedTLSCiphers||supportedTLSProtocols||vendorName||vendorVersion||fullVendorVersion||alive||healthy") (version 3.0; acl "User-Visible Root DSE Operational Attributes"; allow (read,search,compare) userdn="ldap:///anyone";)`

Name	Description	ACI Definition
User-Visible Schema Operational Attributes	Authenticated users can read LDAP schema definitions. Modification or removal may affect applications.	`(target="ldap:///cn=schema") (targetscope="base") (targetattr="objectClass||attributeTypes||dITContentRules||dITStructureRules ||ldapSyntaxes||matchingRules||matchingRuleUse||nameForms||objectClasses||etag||modifiersName||modifyTimestamp") (version 3.0; acl "User-Visible Schema Operational Attributes"; allow (read,search,compare) userdn="ldap:///all";)`

Effective Rights

As the number of ACIs increases, it can be difficult to understand what rights a user actually has. The Get Effective Rights control (OID `1.3.6.1.4.1.42.2.27.9.5.2`) lets you see the rights as evaluated by the server.

By default, only users who can bypass ACIs can use the Get Effective Rights control, and the related operational attributes, `aclRights` and `aclRightsInfo`. The following command grant access to My App:

```
$ dsconfig \
 set-access-control-handler-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --add global-aci:\(targetcontrol=\"EffectiveRights\"\)\ \
(version\ 3.0\;acl\ \"Allow\ My\ App\ to\ get\ effective\
rights\"\;\ allow\(read\)\ userdn=\"ldap:///cn=My\
App,ou=Apps,dc=example,dc=com\"\;\) \
 --add global-aci:\(targetattr=\"aclRights\|\|aclRightsInfo\"\)\
(version\ 3.0\;\ acl\ \"Allow\ My\ App\ to\ read\ effective\
rights\ attributes\"\;\ allow\ \(read,search,compare\)\
userdn=\"ldap:///cn=My\ App,ou=Apps,dc=example,dc=com\"\;\) \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt
```

In this example, Babs Jensen owns the LDAP group that includes people who are willing to carpool:

```
$ ldapsearch \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN "uid=bjensen,ou=people,dc=example,dc=com" \
 --bindPassword hifalutin \
 --baseDN "ou=Self Service,ou=Groups,dc=example,dc=com" \
 "(cn=Carpoolers)"

dn: cn=Carpoolers,ou=Self Service,ou=Groups,dc=example,dc=com
objectClass: top
objectClass: groupOfNames
description: People who are willing to carpool
cn: Carpoolers
owner: uid=bjensen,ou=People,dc=example,dc=com
member: uid=bjensen,ou=People,dc=example,dc=com
```

When My App does the same search with the get effective rights control, and requests the `aclRights` attribute, it sees the rights it has on the entry:

```
$ ldapsearch \
 --control effectiverights \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN "cn=My App,ou=Apps,dc=example,dc=com" \
 --bindPassword password \
 --baseDN "ou=Self Service,ou=Groups,dc=example,dc=com" \
 "(cn=Carpoolers)" \
 aclRights
```

```
dn: cn=Carpoolers,ou=Self Service,ou=Groups,dc=example,dc=com
aclRights;entryLevel: add:1,delete:1,read:1,write:1,proxy:1
```

When My App requests the `aclRightsInfo` attribute, the server shows the ACIs that apply:

```
$ ldapsearch \
 --control effectiverights \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN "cn=My App,ou=Apps,dc=example,dc=com" \
 --bindPassword password \
 --baseDN "ou=Self Service,ou=Groups,dc=example,dc=com" \
 "(cn=Carpoolers)" \
 aclRights \
 aclRightsInfo

dn: cn=Carpoolers,ou=Self Service,ou=Groups,dc=example,dc=com
aclRights;entryLevel: add:1,delete:1,read:1,write:1,proxy:1
aclRightsInfo;logs;entryLevel;add: acl_summary(main): access
allowed(add) on entry/attr(cn=Carpoolers,ou=Self
Service,ou=Groups,dc=example,dc=com, NULL) to (cn=My
App,ou=Apps,dc=example,dc=com) (not proxied) ( reason: evaluated
allow , deciding_aci: Proxied authorization for apps)
aclRightsInfo;logs;entryLevel;delete: acl_summary(main): access
allowed(delete) on entry/attr(cn=Carpoolers,ou=Self
Service,ou=Groups,dc=example,dc=com, NULL) to (cn=My
App,ou=Apps,dc=example,dc=com) (not proxied) ( reason: evaluated
allow , deciding_aci: Proxied authorization for apps)
aclRightsInfo;logs;entryLevel;proxy: acl_summary(main): access
allowed(proxy) on entry/attr(cn=Carpoolers,ou=Self
Service,ou=Groups,dc=example,dc=com, NULL) to (cn=My
App,ou=Apps,dc=example,dc=com) (not proxied) ( reason: evaluated
allow , deciding_aci: Proxied authorization for apps)
aclRightsInfo;logs;entryLevel;read: acl_summary(main): access
allowed(read) on entry/attr(cn=Carpoolers,ou=Self
Service,ou=Groups,dc=example,dc=com, objectClass) to (cn=My
App,ou=Apps,dc=example,dc=com) (not proxied) ( reason: evaluated
allow , deciding_aci: Anonymous read and search access)
aclRightsInfo;logs;entryLevel;write: acl_summary(main): access
allowed(write) on entry/attr(cn=Carpoolers,ou=Self
Service,ou=Groups,dc=example,dc=com, NULL) to (cn=My
App,ou=Apps,dc=example,dc=com) (not proxied) ( reason: evaluated
allow , deciding_aci: Proxied authorization for apps)
```

To request effective rights for another user, use the `--getEffectiveRightsAuthzid` option. This option takes the authorization identity of the user as an argument. The following example shows My App checking Babs's rights to the same entry:

```
$ ldapsearch \
 --getEffectiveRightsAuthzid
"dn:uid=bjensen,ou=People,dc=example,dc=com" \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN "cn=My App,ou=Apps,dc=example,dc=com" \
 --bindPassword password \
 --baseDN "ou=Self Service,ou=groups,dc=example,dc=com" \
 "(cn=Carpoolers)" \
 aclRights

dn: cn=Carpoolers,ou=Self Service,ou=Groups,dc=example,dc=com
aclRights;entryLevel: add:0,delete:1,read:1,write:0,proxy:0
```

The following example checks anonymous user rights to the same entry. Notice that the authorization identity for an anonymous user is expressed as the empty DN:

```
$ ldapsearch \
 --getEffectiveRightsAuthzid "dn:" \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN "cn=My App,ou=Apps,dc=example,dc=com" \
 --bindPassword password \
 --baseDN "ou=Self Service,ou=groups,dc=example,dc=com" \
 "(cn=Carpoolers)" \
 aclRights

dn: cn=Carpoolers,ou=Self Service,ou=Groups,dc=example,dc=com
aclRights;entryLevel: add:0,delete:0,read:1,write:0,proxy:0
```

To check access to a potentially nonexistent attribute, use the `--getEffectiveRightsAttribute` option. This option takes a comma-separated attribute list as an argument. The following example checks Andy Hall's access to the member attribute for the Carpooler's group entry:

```
$ ldapsearch \
 --getEffectiveRightsAuthzid
"dn:uid=ahall,ou=People,dc=example,dc=com" \
 --getEffectiveRightsAttribute member \
 --hostname localhost \
 --port 1636 \
 --useSsl \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --bindDN "cn=My App,ou=Apps,dc=example,dc=com" \
 --bindPassword password \
 --baseDN "ou=Self Service,ou=groups,dc=example,dc=com" \
 "cn=Carpoolers" \
 aclRights

dn: cn=Carpoolers,ou=Self Service,ou=Groups,dc=example,dc=com
aclRights;entryLevel: add:0,delete:0,read:1,write:0,proxy:0
aclRights;attributeLevel;member:
search:1,read:1,compare:1,write:0,selfwrite-add:1,selfwrite-
delete:1,proxy:0
```

Proxy Global Policies

By default, a policy matches all entries, all types of connection, and all users. You set the properties of the policy to restrict its scope of application.

Global access control policies can allow the following:

- Requests for specified LDAP controls and extended operations.
- Access to specific attributes, with support for wildcards, @objectclass notation, and exceptions to simplify settings.
- Read access (for read, search, and compare operations).
- Write access (for add, delete, modify, and modify DN operations).
- Requiring authentication before other requests.
- Requests targeting a particular scope, with wildcards to simplify settings.
- Requests originating or not from specific client addresses or domains.
- Requests using a specified protocol.

- Requests using a specified port.
- Requests using a minimum security strength factor.
- Requests from a user whose DN does or does not match a DN pattern.

Default Global Policies

Access control rules are defined using individual access control policy entries. A user’s access is defined as the union of all access control rules that apply to that user. In other words, an individual access control rule can only grant additional access and can not remove rights granted by another rule. This approach results in an access control policy which is easier to understand and audit, since all rules can be understood in isolation.

Policy	Settings
Anonymous extended operation and control access	***authentication-required*** • `false` ***allowed-extended-operation*** • `Cancel` • `GetSymmetricKey` • `PasswordModify` • `StartTLS` • `WhoAmI` ***allowed-control*** • `Assertion` • `MatchedValues` • `NoOp` • `PasswordQualityAdvice` • `PermissiveModify` • `PostRead` • `PreRead` • `RealAttrsOnly` • `SimplePagedResults` • `VirtualAttrsOnly` • `AuthorizationIdentity` • `PasswordPolicy` • `TransactionId` • `Vlv`

Policy	Settings
Authenticated extended operation and control access	***authentication-required*** • `true` ***allowed-extended-operation*** • `PasswordPolicyState` ***allowed-control*** • `ManageDsaIt` • `SubEntries` • `RelaxRules` • `SubtreeDelete` • `ServerSideSort`
Schema access	***authentication-required*** • `true` ***request-target-dn-equal-to*** • `cn=schema` ***permission*** • `read` ***allowed-attribute*** • `objectClass` • `@subschema` • `etag` • `ldapSyntaxes` • `modifiersName` • `modifyTimestamp`

Policy	Settings
Root DSE access	***authentication-required*** • `false` ***request-target-dn-equal-to*** • `""` ***permission*** • `read` ***allowed-attribute*** • `objectClass` • `namingContexts` • `subSchemaSubEntry` • `supportedAuthPasswordSchemes` • `supportedControl` • `supportedExtension` • `supportedFeatures` • `supportedLDAPVersion` • `supportedSASLMechanisms` • `supportedTLSCiphers` • `supportedTLSProtocols` • `vendorName` • `vendorVersion` • `fullVendorVersion` • `alive` • `healthy`

Policy	Settings
Monitor access	***authentication-required*** • `true` ***request-target-dn-equal-to*** • `cn=monitor` ***permission*** • `read` ***allowed-attribute*** • `*` • `+`

Reject Unauthenticated Requests

The following example uses a single broad policy for authenticated access, and another narrow policy for anonymous extended operation access:

▼ Show commands

```
$ dsconfig \
 create-global-access-control-policy \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --policy-name "Authenticated access all entries" \
 --set authentication-required:true \
 --set request-target-dn-not-equal-to:"**,cn=changelog" \
 --set permission:read \
 --set allowed-attribute:"*" \
 --set allowed-attribute:createTimestamp \
 --set allowed-attribute:creatorsName \
 --set allowed-attribute:entryDN \
 --set allowed-attribute:entryUUID \
 --set allowed-attribute:etag \
 --set allowed-attribute:governingStructureRule \
 --set allowed-attribute:hasSubordinates \
 --set allowed-attribute:isMemberOf \
 --set allowed-attribute:modifiersName \
 --set allowed-attribute:modifyTimestamp \
 --set allowed-attribute:numSubordinates \
 --set allowed-attribute:structuralObjectClass \
 --set allowed-attribute:subschemaSubentry \
 --set allowed-attribute-exception:authPassword \
 --set allowed-attribute-exception:userPassword \
 --set allowed-attribute-exception:debugSearchIndex \
 --set allowed-attribute-exception:@changeLogEntry \
 --set allowed-control:Assertion \
 --set allowed-control:AuthorizationIdentity \
 --set allowed-control:Csn \
 --set allowed-control:ManageDsaIt \
 --set allowed-control:MatchedValues \
 --set allowed-control:Noop \
 --set allowed-control:PasswordPolicy \
 --set allowed-control:PermissiveModify \
 --set allowed-control:PostRead \
 --set allowed-control:PreRead \
 --set allowed-control:ProxiedAuthV2 \
 --set allowed-control:RealAttributesOnly \
 --set allowed-control:ServerSideSort \
 --set allowed-control:SimplePagedResults \
 --set allowed-control:TransactionId \
 --set allowed-control:VirtualAttributesOnly \
 --set allowed-control:Vlv \
 --set allowed-extended-operation:GetSymmetricKey \
 --set allowed-extended-operation:PasswordModify \
 --set allowed-extended-operation:PasswordPolicyState \
 --set allowed-extended-operation:StartTls \
 --set allowed-extended-operation:WhoAmI \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt

$ dsconfig \
 create-global-access-control-policy \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --policy-name "Anonymous extended operation access" \
 --set authentication-required:false \
 --set allowed-extended-operation:GetSymmetricKey \
 --set allowed-extended-operation:StartTls \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt
```

Require Secure Connections

The following example creates a policy with a minimum security strength factor of 128. This effectively permits only secure connections for requests targeting data in `dc=example,dc=com`. The security strength factor defines the key strength for GSSAPI, SSL, and TLS:

```
$ dsconfig \
 create-global-access-control-policy \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --policy-name "Require secure connections for example.com data" \
 --set request-target-dn-equal-to:"**,dc=example,dc=com" \
 --set request-target-dn-equal-to:dc=example,dc=com \
 --set connection-minimum-ssf:128 \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt
```

Anonymous Requests From Specific Network

The following example allows anonymous requests from clients in the `example.com` domain:

```
$ dsconfig \
 set-global-access-control-policy-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --policy-name "Anonymous extended operation access" \
 --set connection-client-address-equal-to:.example.com \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt

$ dsconfig \
 set-global-access-control-policy-prop \
 --hostname localhost \
 --port 4444 \
 --bindDN uid=admin \
 --bindPassword password \
 --policy-name "Root DSE access" \
 --set connection-client-address-equal-to:.example.com \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin \
 --no-prompt
```

You can also use the `connection-client-address-not-equal-to` property to reject requests from a particular host, domain, address, or address mask.

For additional details, see Global Access Control Policy.

Data Encryption

DS directory servers can encrypt directory data in backend files on disk. This keeps the data confidential until it is accessed by a directory client.

IMPORTANT

Encrypting stored directory data does not prevent it from being sent over the network in the clear.

Use secure connections to protect data sent over the network.

Advantages	Trade-offs
Confidentiality and integrity Encrypted directory data is confidential, remaining private until decrypted with a proper key. Encryption ensures data integrity at the moment it is accessed. The DS directory service cannot decrypt corrupted data.	**Equality indexes limited to equality matching** When an equality index is configured without confidentiality, the server maintains values in sorted order. It can then use the cleartext equality index can for searches that require ordering or match initial substrings. An example of a search that requires ordering is a search with a filter `"(cn<=App)"`. The filter matches entries with `commonName` up to those starting with `App` (case-insensitive) in alphabetical order. An example of a search that matches an initial substring is a search with a filter `"(cn=A*)"`. The filter matches entries having a `commonName` that starts with `a` (case-insensitive). In an equality index with confidentiality enabled, the server can no longer sort the values. As a result, you must either add indexes or accept that ordering and initial substring searches are unindexed.
Protection on shared infrastructure When you share the infrastructure, you relinquish full and sole control of directory data. For example, if the DS directory server runs in the cloud, or in a data center with shared disks, the file system and disk management are not under your control. With encrypted data, other users cannot read the files unless they also have the decryption keys.	**Performance impact** Encryption and decryption requires more processing than handling plaintext values. Encrypted values also take up more space than plaintext values.

To check what a system user with read access to backend database files can see, use the **`backendstat dump-raw-db`** command. The **`backendstat`** subcommands **`list-raw-`**

`dbs` and **`dump-raw-db`** let you list and view the low-level databases within a backend. Unlike the output of other subcommands, the output of the **`dump-raw-db`** subcommand is neither decrypted nor formatted for readability. Instead, you can see values as they are stored in the backend file.

In a backend database, the `id2entry` index holds LDIF for directory entries. For a database that is not encrypted, the corresponding low-level database shows the cleartext strings:

▼ Show details

```
$ stop-ds

$ backendstat list-raw-dbs --backendId dsEvaluation
...
/dc=com,dc=example/id2entry

$ backendstat \
 dump-raw-db \
 --backendId dsEvaluation \
 --dbName /dc=com,dc=example/id2entry
...
Key (len 8):
 00 00 00 00 00 00 00 1E                          ........
 Value (len 437):
 02 00 81 B1 03 01 06 27 75 69 64 3D 62 6A 65 6E
.......'uid=bjen
 73 65 6E 2C 6F 75 3D 50 65 6F 70 6C 65 2C 64 63
sen,ou=People,dc
 3D 65 78 61 6D 70 6C 65 2C 64 63 3D 63 6F 6D 01
=example,dc=com.
 06 11 01 08 01 13 62 6A 65 6E 73 65 6E 40 65 78
......bjensen@ex
 61 6D 70 6C 65 2E 63 6F 6D 01 09 01 04 30 32 30
ample.com....020
 39 01 16 01 0C 65 6E 2C 20 6B 6F 3B 71 3D 30 2E  9....en,
ko;q=0.
 38 01 10 01 29 75 69 64 3D 74 72 69 67 64 65 6E
8...)uid=trigden
 2C 20 6F 75 3D 50 65 6F 70 6C 65 2C 20 64 63 3D  , ou=People,
dc=
 65 78 61 6D 70 6C 65 2C 64 63 3D 63 6F 6D 01 04
example,dc=com..
 02 13 50 72 6F 64 75 63 74 20 44 65 76 65 6C 6F  ..Product
Develo
 70 6D 65 6E 74 06 50 65 6F 70 6C 65 01 0B 01 07
```

```
pment.People....
 42 61 72 62 61 72 61 01 0C 01 0F 2B 31 20 34 30  Barbara....+1
40
 38 20 35 35 35 20 31 38 36 32 01 0D 01 06 4A 65  8 555
1862....Je
 6E 73 65 6E 01 07 02 0E 42 61 72 62 61 72 61 20
nsen....Barbara
 4A 65 6E 73 65 6E 0B 42 61 62 73 20 4A 65 6E 73  Jensen.Babs
Jens
 65 6E 01 0E 01 0D 2F 68 6F 6D 65 2F 62 6A 65 6E
en..../home/bjen
 73 65 6E 01 0F 01 0F 2B 31 20 34 30 38 20 35 35  sen....+1 408
55
 35 20 31 39 39 32 01 11 01 04 31 30 30 30 01 12  5
1992....1000..
 01 2E 7B 53 53 48 41 7D 33 45 66 54 62 33 70 37  ..
{SSHA}3EfTb3p7
 71 75 6F 75 73 4B 35 34 2B 41 4F 34 71 44 57 6C
quousK54+AO4qDWl
 56 33 4F 39 54 58 48 57 49 4A 49 32 4E 41 3D 3D
V3O9TXHWIJI2NA==
 01 13 01 04 31 30 37 36 01 05 01 14 4F 72 69 67
....1076....Orig
 69 6E 61 6C 20 64 65 73 63 72 69 70 74 69 6F 6E  inal
description
 01 14 01 07 62 6A 65 6E 73 65 6E 01 15 01 0D 53
....bjensen....S
 61 6E 20 46 72 61 6E 63 69 73 63 6F 01 01 02 01  an
Francisco....
 24 38 38 37 37 33 32 65 38 2D 33 64 62 32 2D 33  $887732e8-
3db2-3
 31 62 62 2D 62 33 32 39 2D 32 30 63 64 36 66 63  1bb-b329-
20cd6fc
 65 63 63 30 35                                   ecc05
```

Enable confidentiality to encrypt database backends. When confidentiality is enabled, the server encrypts entries before storing them in the backend. The following command enables backend confidentiality with the default encryption settings:

```
$ dsconfig \
 set-backend-prop \
 --hostname localhost \
 --port 4444 \
 --bindDn uid=admin \
 --bindPassword password \
 --backend-name dsEvaluation \
 --set confidentiality-enabled:true \
 --no-prompt \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin
```

After confidentiality is enabled, the server encrypts entries *only when it next writes them*. The server does not automatically rewrite all entries in encrypted form. Instead, it encrypts each entry the next time it is updated, or when you re-import data from LDIF.

To see the impact of backend encryption, import the data and view the results, as in the following example:

▼ Show details

```
$ stop-ds

$ export-ldif \
 --offline \
 --backendId dsEvaluation \
 --includeBranch dc=example,dc=com \
 --ldifFile backup.ldif

$ import-ldif \
 --offline \
 --backendId dsEvaluation \
 --includeBranch dc=example,dc=com \
 --ldifFile backup.ldif

$ backendstat \
 dump-raw-db \
 --backendId dsEvaluation \
 --dbName /dc=com,dc=example/id2entry
...
Key (len 8):
 00 00 00 00 00 00 00 C2                            ........
 Value (len 437):
 02 02 81 82 01 C4 95 87 5B A5 2E 47 97 80 23 F4  ........
[..G..#.
 CE 5D 93 25 97 D4 13 F9 0A A3 A8 31 9A D9 7A 70
.].%.......1..zp
 FE 3E AC 9D 64 41 EB 7B D5 7F 7E B8 B7 74 52 B8  .>..dA.{.^?
~..tR.
 C7 7F C8 79 19 46 7D C5 5D 5B 83 9C 5B 9F 85 28  .^?.y.F}.][..
[..(
 83 A2 5F A0 C1 B1 09 FC 2F E3 D8 82 4A AA 8B D9
```

```
.._...../...J...
 78 43 34 50 AE A1 52 88 5B 70 97 D2 E1 EA 87 CA  xC4P..R.
[p......
 3B 4D 07 DC F9 F8 30 BB D2 76 51 C8 75 FF FA 80
;M....0..vQ.u...
 77 E1 6A 8B 5B 8F DA A4 F4 0B B5 20 56 B3 19 19  w.j.[......
V...
 22 D8 9D 38 04 E3 4D 94 A7 99 4B 81 16 AD 88 46
"..8..M...K....F
 FC 3F 7E 78 66 B8 D1 E9 86 A0 F3 AC B6 68 0D A9  .?
~xf........h..
 9A A7 3C 30 40 37 97 4E 90 DD 63 16 8E 11 0F 5E  ..
<0@7.N..c....^
 9D 5B 86 90 AF 4E E2 1F 9E 70 73 14 0A 11 5C DB  .
[...N...ps...\.
 B7 BC B8 A9 31 3F 74 8D 0A 9F F4 6C E1 B0 36 78  ....1?
t....l..6x
 F0 5A 5E CD 7C B3 A2 36 66 8E 88 86 A0 8B 9A 77
.Z^.|..6f......w
 D5 CD 7E 9C 4E 62 20 0E D0 DB AD E7 7E 99 46 4F  ..~.Nb
.....~.FO
 67 C7 A6 7E 2C 24 82 50 51 9F A7 B2 02 44 5B 30
g..~,$.PQ....D[0
 74 41 99 D9 83 69 EF AE 2E C0 FF C4 E6 4F F2 2F
tA...i.......O./
 95 FB 93 65 30 2A 2D 8D 20 88 83 B5 DE 35 B6 20  ...e0*-.
....5.
 47 17 30 25 60 FD E3 43 B9 D6 A4 F7 47 B6 6C 9F
G.0%`..C....G.l.
 47 FD 63 8E 7F A5 00 CE 6C 3E BC 95 23 69 ED D0
G.c.^?...l>..#i..
 69 4F BE 61 BD 30 C2 40 66 F6 F9 C3 3E C1 D7 8C
iO.a.0.@f...>...
 B0 C8 4A 2E 27 BE 13 6C 40 88 B0 13 A3 12 F4 50
..J.'..l@......P
 CA 92 D8 EB 4A E5 3F E2 64 A3 76 C7 5C 2B D8 89
....J.?.d.v.\+..
 A3 6E C1 F7 0A C2 37 7A BD AF 14 4B 52 04 6B F2
.n....7z...KR.k.
 8F 4F C3 F8 00 90 BA 0F EC 6D B1 2D A8 18 0C A6
.O.......m.-....
 29 96 82 3B 5C BC D0 F4 2B BE 9C C5 8B 18 7A DE
)..;\...+.....z.
 C7 B5 10 2D 45 50 4F 77 ED F7 23 34 95 AF C3 2E  ...-
```

```
EPOw..#4....
 B0 9B FA E9 DF                                        .....
```

The settings for data confidentiality depend on the encryption capabilities of the JVM. See the explanations in javax.crypto.Cipher. You can accept the default settings, or choose to specify the following:

- The cipher algorithm defining how the plaintext is encrypted and decrypted.
- The cipher mode of operation defining how a block cipher algorithm transforms data larger than a single block.
- The cipher padding defining how to pad the plaintext to reach appropriate size for the algorithm.
- The cipher key length, where longer key lengths strengthen encryption at the cost of lower performance.

The default settings for confidentiality are `cipher-transformation: AES/GCM/NoPadding`, and `cipher-key-length: 128`. This means the algorithm is the Advanced Encryption Standard (AES), and the cipher mode is Galois/Counter Mode (GCM). The syntax for the `cipher-transformation` is *`algorithm/mode/padding`*. You must specify the *algorithm*, *mode*, and *padding*. When the algorithm does not require a mode, use `NONE`. When the algorithm does not require padding, use `NoPadding`.

DS servers encrypt data using symmetric keys. Servers store symmetric keys, encrypted with the shared master key, with the data they encrypt. As long as servers have the same shared master key, any server can recover symmetric keys needed to decrypt data.

Symmetric keys for (deprecated) reversible password storage schemes are the exception to this rule. When you configure a reversible password storage scheme, enable the `adminRoot` backend, and configure a replication domain for `cn=admin data`.

You can enable confidentiality by backend index, as long as confidentiality is enabled for the backend itself. Confidentiality hashes keys for equality type indexes using SHA-1. It encrypts the list of entries matching a substring key for substring indexes. The following example shows how to enable confidentiality for the `mail` index:

```
$ dsconfig \
 set-backend-index-prop \
 --hostname localhost \
 --port 4444 \
 --bindDn uid=admin \
 --bindPassword password \
 --backend-name dsEvaluation \
 --index-name mail \
 --set confidentiality-enabled:true \
 --no-prompt \
 --usePkcs12TrustStore /path/to/opendj/config/keystore \
 --trustStorePassword:file /path/to/opendj/config/keystore.pin
```

After changing the index configuration, rebuild the index to enforce confidentiality immediately.

Confidentiality cannot be enabled for VLV indexes. Avoid using sensitive attributes in VLV indexes.

When reading files for an encrypted backend database, be aware that although user data is kept confidential, the following are *not encrypted* on disk:

- Existing backend database files.

  The server encrypts backend database content *only when it is next written*. If you must make sure that all relevant data are encrypted, export the data to LDIF and then import the data again.
- The `dn2id` index and its keys.
- Substring index keys.

  Substring index values are encrypted.

Encrypting and decrypting data require cryptographic processing that reduces throughput, and extra space for larger encrypted values. Tests with default settings show that the cost of enabling confidentiality can be quite modest. Your results can vary based on the host system hardware, the JVM, and the settings for `cipher-transformation` and `cipher-key-length`. Make sure you test your deployment to qualify the impact of confidentiality before changing settings in production.

Client Best Practices

Encourage best practices for directory clients that you control and influence.

Handle Input Securely

When taking input directly from a user or another program, handle the input securely by using appropriate methods to sanitize the data. Failure to sanitize the input data can leave your client vulnerable to injection attacks.

For example, the DS Java APIs have `Filter.format()` and `DN.format()` methods. Like the Java `String.format()` methods, these escape input objects when formatting output.

When writing command-line or HTTP clients, make sure you sanitize the input.

Use Secure Connections

Use secure connections except when reading public information anonymously. Always use secure connections when sending credentials for authentication, and when reading or writing any data that is not public.

For LDAP clients, either connect to the directory server’s LDAPS port, or begin each session with the StartTLS extended operation on the insecure LDAP port.

For HTTP clients, use HTTPS.

Authenticate Appropriately

Unless your client only reads public information, authenticate to the directory server.

Use an account that is specific to your client when authenticating. This helps avoid risks involved in sharing credentials between accounts. Furthermore, it makes debugging easier because your client’s actions are associated with its account.

Avoid username/password credentials for clients, by certificate-based authentication. For details, see Certificate-Based Authentication.

Consider OAuth 2.0

DS servers support OAuth 2.0 for HTTP authorization. This lets the HTTP client access directory data without having a directory account. The directory acts as an OAuth 2.0 resource server, as described in Configure HTTP Authorization.

An OAuth 2.0 client gets authorization from the resource owner, such as the user, device, or thing whose account it needs to access, and presents the OAuth 2.0 bearer access token to get access to the account. Access tokens give the bearer access, regardless of the bearer’s identity.

Send access tokens only over secure HTTPS connections to prevent eavesdroppers from stealing the token.

Consider Proxied Authorization

DS servers support LDAP proxied authorization control. With proxied authorization, an LDAP client binds to the directory using its own account, and sends requests with the user authorization ID in the control. For details, see Proxied Authorization.

When the user is already safely authenticated by other means, proxied authorization makes it easy to reuse a connection that is dedicated and bound to the client.

Apply Resource Limits

LDAP clients can set time limits and size limits on search requests. Setting limits is appropriate when searches are partially or fully determined by user input.

You can use the DS Java APIs methods `SearchRequest.setSizeLimit()` and `SearchRequest.setTimeLimit()` for this purpose.

The Directory Services **`ldapsearch`** command has `--sizeLimit` and `--timeLimit` options.

Tests

In this context, *validation* means checking that you are building the right thing, whereas *verification* means checking that you are building it in the right way.

Requirement Validation

Before you begin to write specific tests, step back to review the big picture. Do the security requirements make sense for the directory service, and for the users of the service?

For the directory service, aim to prevent problems caused by security threats. The directory service must expose sensitive information only in secure ways. It must require strong authentication credentials, and limit access.

For the users, the service must permit access from any device or application using standard protocols and tools. The directory should protect your sensitive information while making it easy to connect.

In refining the security requirements for the directory service, make sure there is an appropriate balance between security and user needs. An ultimately secure directory service is one that denies all user access. An ultimately flexible directory service is one that relinquishes all control. The appropriate balance is somewhere in the middle.

Be aware that directory service security is only part of an overall strategy, one that aims to help users and application developers make appropriately secure choices. In validating the requirements, understand how the directory service fits into the overall identity and access management strategy.

Functional Tests

Long before you roll out the directory service, when you start to prepare server configurations in a lab environment, begin by testing the directory's functional capabilities. As you understand your users' requirements, reproduce what their client applications will do in your tests. In most cases, it is not feasible to exhaustively reproduce everything that every directory client will do. Instead, choose a representative sample of actions. Test your expectations, both for normal and for insecure client application and user behavior.

The goals for your functional testing are to verify compliance, to uncover problems, and to begin automating tests early in the project. Test automation should drive you to use version control software, and continuous integration software as well. Be ready to roll back any change you make if a test fails, and make sure that every change is reviewed and tested before it is pushed further along in the process.

Aim to keep the automated tests both representative and short. As you build out the deployment and complexity grows, automated tests let you build the service with confidence, by repeatedly iterating with small changes to fix problems, and to better match users' expectations.

Integration Tests

Before you apply a change to a production environment, verify the impact under conditions as close as possible to those of the production environment.

In the test environment, the directory service should mirror production for configuration, client application configuration and load, and directory data, including access policies. This is the environment where end-to-end testing first takes place.

Although you might not test at a scale that is identical to production, the test environment must remain representative. For example, when using replicas in production, also use replicas in the pre-production test environment. When using secure connections everywhere in production, also use them in the test environment. If you expect many client applications accessing the production directory, and particular client usage patterns, also simulate those in the test environment.

Automate your testing in the pre-production environment as well. Each change for the production environment should first pass the pre-production tests. You will need to iterate through the tests for each change.

In deployments where updates to new servers would not harm production data, consider using canary servers. You deploy a small group of canary servers in production that have the change. You then test and monitor these servers to compare with unchanged servers. If the change looks good after enough testing and monitoring, you roll out more servers with the change. If something goes wrong, you have only exposed a small proportion of production clients to the change. Only use this when you are sure that replication from a canary server would not corrupt any production data.

Continuous Verification

Directory services integrate with many monitoring solutions. Access monitoring information over HTTP, LDAP, SNMP, and JMX, and by sending Common Audit events to local log files and remote systems for further processing.

Adapt some of your tests to verify operation of the service in production. For example, a few end-to-end tests in production can alert you to problems early before they impact many users.

Was this helpful?