Red Hat JBoss Enterprise Application Platform 7.4

How to Configure Server Security

Instructions for securing Red Hat JBoss Enterprise Application Platform and its management interfaces.
Instructions for securing Red Hat JBoss Enterprise Application Platform and its management interfaces.
Abstract

The purpose of this document is to provide a practical guide to securing Red Hat JBoss Enterprise Application Platform (JBoss EAP). More specifically, this guide details how to secure all of the management interfaces on JBoss EAP. Before reading this guide, users should read through the Security Architecture document for Red Hat JBoss Enterprise Application Platform and have a solid understanding of how JBoss EAP handles security. This document also makes use of the JBoss EAP CLI interface for performing configuration changes. When completing this document, readers should have a solid, working understanding of how to secure JBoss EAP.
Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION .............................................. 9

MAKING OPEN SOURCE MORE INCLUSIVE ................................................................. 10

CHAPTER 1. SECURING THE SERVER AND ITS INTERFACES ........................................ 11

1.1. BUILDING BLOCKS .............................................................................................. 11
  1.1.1. Interfaces and socket bindings ..................................................................... 11
  1.1.2. Elytron Subsystem ...................................................................................... 11
     1.1.2.1. Enable Elytron Security Across the Server ........................................... 11
     1.1.2.2. Create an Elytron Security Domain ....................................................... 12
        Add a Security Domain Using the Management CLI ..................................... 12
        Add a Security Domain Using the Management Console ............................ 12
     1.1.2.3. Create an Elytron Security Realm ........................................................ 12
        Add a Security Realm Using the Management CLI ...................................... 12
        Add a Security Realm Using the Management Console ............................. 12
     1.1.2.4. Create an Elytron Role Decoder ............................................................ 13
        Add a Role Decoder Using the Management CLI .......................................... 13
        Add a Role Decoder Using the Management Console ................................. 13
     1.1.2.5. Adding a source-address-role-decoder to the elytron subsystem ............ 13
  1.1.3. Configuring an aggregate-role-decoder to the elytron subsystem .................. 14
     1.1.3.1. Create an Elytron Role Mapper ............................................................. 15
        Adding a Role Mapper Takes the General Form .......................................... 15
        Adding a Role Mapper Using the Management Console .............................. 15
     1.1.3.2. Create an Elytron Permission Set .......................................................... 15
        Add a Permission Set Using the Management CLI ........................................ 15
     1.1.3.3. Create an Elytron Permission Mapper .................................................... 15
        Add a Permission Mapper Using the Management CLI ............................... 15
        Add a Permission Mapper Using the Management Console ....................... 16
     1.1.3.4. Creating an Authentication Configuration ............................................. 16
        Add an Authentication Configuration Using the Management CLI ................ 16
        Add an Authentication Configuration Using the Management Console ........ 16
     1.1.3.5. Creating an Authentication Context ..................................................... 16
        Add an Authentication Context Using the Management CLI ........................ 16
        Add an Authentication Context Using the Management Console ............... 17
     1.1.3.6. Create an Elytron Authentication Factory .............................................. 17
        Add an Authentication Factory Using the Management CLI .......................... 17
        Add an Authentication Factory Using the Management Console .................. 17
     1.1.3.7. Create an Elytron Keystore .................................................................... 17
        Add a Keystore Using the Management CLI .................................................. 17
        Add a Keystore Using the Management Console ......................................... 18
     1.1.3.8. Create an Elytron Key Manager ............................................................. 18
        Add a Key Manager Using the Management CLI ........................................... 18
        Add a Key Manager Using the Management Console ................................. 18
     1.1.3.9. Create an Elytron Truststore ................................................................. 19
     1.1.3.10. Create an Elytron Trust Manager ......................................................... 19
     1.1.3.11. Using the Out of the Box Elytron Components ...................................... 19
        1.1.3.11.1. Securing Management Interfaces .................................................. 19
        1.1.3.11.2. Securing Applications ................................................................ 19
        1.1.3.11.3. Using SSL/TLS ........................................................................... 19
        1.1.3.11.4. Using Elytron with Other Subsystems ......................................... 20
     1.1.3.12. Enable and Disable the Elytron Subsystem ......................................... 21
1.1.4. Legacy Security Subsystem
1.1.4.1. Disabling the security subsystem
1.1.4.2. Enabling the security subsystem
1.1.5. Legacy security realms
1.1.6. Using authentication and socket bindings for securing the management interfaces

1.2. HOW TO SECURE THE MANAGEMENT INTERFACES

Elytron Integration with the Management CLI
1.2.1. Configure networking and ports
1.2.2. Disabling the management console
1.2.3. Disabling remote access to JMX
1.2.4. Silent authentication

1.2.5. One-way SSL/TLS for the management interfaces using the Elytron subsystem
1.2.5.1. Enabling one-way SSL/TLS using a security command
1.2.5.2. Enabling one-way SSL/TLS using the Elytron subsystem commands
1.2.5.3. Enabling one-way SSL/TLS using the management console

1.2.6. Two-way SSL/TLS for the management interfaces using the Elytron Subsystem
1.2.6.1. Generating client certificates
1.2.6.2. Enabling two-way SSL/TLS using a security command
1.2.6.3. Enabling two-way SSL/TLS using the Elytron subsystem commands

1.2.7. SASL authentication for the management interfaces using the CLI security command
Enable SASL authentication
Disable SASL authentication
Reorder SASL mechanisms

1.2.8. HTTP authentication for the management interfaces using the CLI security command
Enable HTTP authentication
Disable HTTP Authentication

1.2.9. Configuring the management interfaces for one-way SSL/TLS with legacy core management authentication
1.2.9.1. Creating a keystore to secure the management interfaces
1.2.9.2. Ensuring the management interfaces bind to HTTPS
1.2.9.3. Custom socket-binding-group
1.2.9.4. Creating a new security realm
1.2.9.5. Configuring the management interfaces to use a security realm
1.2.9.6. Configuring the management interfaces to use a keystore
1.2.9.7. Updating the jboss-cli.xml file

1.2.10. Setting up two-way SSL/TLS for the management interfaces with legacy core management authentication

1.2.11. HTTPS Listener Reference
1.2.11.1. About Cipher Suites

1.2.13. FIPS 140-2 Compliant Cryptography
1.2.13.1. Enable FIPS 140-2 Cryptography for SSL/TLS on Red Hat Enterprise Linux 7 and Later
Configuring the NSS database
Configure the Management CLI for FIPS 140-2 Compliant Cryptography for SSL/TLS
Configure the Elytron and Undertow Subsystems
Configure Undertow with the Legacy Core Management Authentication
1.2.13.2. Enable FIPS 140-2 Cryptography for SSL/TLS Using Bouncy Castle
Prerequisites
Configure the Management CLI for FIPS 140-2 Compliant Cryptography for SSL/TLS Using Elytron
Configure the Elytron and Undertow Subsystems

1.2.14. FIPS 140-2 Compliant Cryptography on IBM JDK
1.2.14.1. Key Storage
1.2.14.2. Management CLI Configuration
1.11.5. Reinitialize a Trust Manager
- Reinitializing a Trust Manager from the management CLI
- Reinitializing a Trust Manager from the management console

1.11.6. Keystore Alias

1.11.7. Using a client-ssl-context

1.11.8. Using a server-ssl-context
- Add a Server SSL Context Using the Management CLI
- Add a Server SSL Context Using the Management Console

1.11.9. Using a server-ssl-sni-context
- Configure server-ssl-sni-context Using the Management Console

1.11.10. Custom SSL Components
- 1.11.10.1. Add a Custom Component to Elytron
- 1.11.10.2. Including Arguments in a Custom Elytron Component
- 1.11.10.3. Using Custom Trust Managers with Elytron
  - Requirements for Implementing a Custom Trust Manager
  - Example Implementations
  - Adding the Custom Trust Manager

1.11.11. Default SSLContext

1.11.12. Using a Certificate Revocation List

1.11.13. Using a Certificate Authority to Manage Signed Certificates
- Configure a Let’s Encrypt Account
- Create an Account with the Certificate Authority
- Update an Account with the Certificate Authority
- Change the Account Key Associated with the Certificate Authority
- Deactivate the Account with the Certificate Authority
- Get the Metadata Associated with the Certificate Authority
- Configure a Let’s Encrypt Account Using Management Console

1.11.14. Keystore Manipulation Operations
- Keystore Manipulation Operations Using the Management CLI
- Keystore Manipulation Operations Using the Management Console
- Keystore Certificate Authority Operations Using the Management CLI
- Keystore Certificate Authority Operations Using the Management Console

1.11.15. Configuring Evidence Decoder for X.509 Certificate with Subject Alternative Name Extension

1.11.16. Configuring an Aggregate Evidence Decoder

1.11.17. Configuring X.500 Subject Evidence Decoder

1.11.18. Using Custom Evidence Decoder Implementation

CHAPTER 2. SECURING A MANAGED DOMAIN

2.1. CONFIGURING PASSWORD AUTHENTICATION FOR THE DOMIAN CONTROLLER WITH ELYTRON

2.2. CONFIGURING PASSWORD AUTHENTICATION FOR THE DOMIAN CONTROLLER WITH LEGACY CORE-MANAGEMENT AUTHENTICATION

2.3. CONFIGURING SSL OR TLS FOR THE DOMIAN CONTROLLER WITH ELYTRON

2.4. CONFIGURING SSL IN DOMAIN MODE BY USING ELYTRON

2.5. CONFIGURING SSL OR TLS FOR THE LEGACY CORE MANAGEMENT AUTHENTICATION MECHANISM

CHAPTER 3. SECURING USERS OF THE SERVER AND ITS MANAGEMENT INTERFACES

3.1. USER AUTHENTICATION WITH ELYTRON
- 3.1.1. Default Configuration
  - 3.1.1.1. Default Elytron HTTP Authentication Configuration
  - 3.1.1.2. Default Elytron Management CLI Authentication
- 3.1.2. Secure the Management Interfaces with a New Identity Store
3.1.3. Adding Silent Authentication
3.1.4. Mapping Identity for Authenticated Management Users
3.1.5. Using Elytron Client with the Management CLI

3.2. IDENTITY PROPAGATION AND FORWARDING WITH ELYTRON
3.2.1. Propagating Security Identities for Remote Calls
    Configure the Server for Security Propagation
    Review the Example Application Code That Propagates a Security Identity
3.2.2. Utilizing Authorization Forwarding Mode
    Configure the Authentication Client on the Forwarding Server
    Configure the Authorization Forwarding on the Receiving Server
3.2.3. Creating a case-principal-transformer to change the case characters of your principal username
3.2.4. Retrieving Security Identity Credentials
3.2.5. Mechanisms That Support Security Identity Propagation

3.3. IDENTITY SWITCHING WITH ELYTRON
3.3.1. Switching Identities in Server-to-server Jakarta Enterprise Beans Calls

3.4. USER AUTHENTICATION WITH LEGACY CORE MANAGEMENT AUTHENTICATION
3.4.1. Default User Configuration
3.4.2. Adding Authentication via LDAP
3.4.3. Using JAAS for Securing the Management Interfaces

3.5. ROLE-BASED ACCESS CONTROL
3.5.1. Enabling Role-Based Access Control
    CLI to Enable RBAC
    Management CLI Command to Disable RBAC
    XML Configuration to Enable or Disable RBAC
3.5.2. Changing the Permission Combination Policy
    Setting the Permission Combination Policy
3.5.3. Managing Roles
    3.5.3.1. Configure User Role Assignment Using the Management CLI
        Viewing Role Assignment Configuration
        Add a New Role
        Add a User as Included in a Role
        Add a User as Excluded in a Role
        Remove User Role Include Configuration
        Remove User Role Exclude Configuration
3.5.4. Configure User Role Assignment with the Elytron Subsystem
3.5.5. Roles and User Groups
3.5.6. Configure Group Role Assignment Using the Management CLI
        Viewing Group Role Assignment Configuration
        Add a New Role
        Add a Group as Included in a Role
        Add a Group as Excluded in a Role
        Remove Group Role Include Configuration
        Remove a User Group Exclude Entry
3.5.7. Using RBAC with LDAP
3.5.8. Scoped Roles
    3.5.8.1. Configuring Scoped Roles from the Management CLI
        Add a New Scoped Role
        Viewing and Editing a Scoped Role Mapping
        Delete a Scoped Role
        Adding and Removing Users
    3.5.8.2. Configuring Scoped Roles from the Management Console
        Add a New Scoped Role
        Edit a Scoped Role
CHAPTER 4. SECURE STORAGE FOR CREDENTIALS

4.1. CREDENTIUAL STORES IN ELYTRON

4.1.1. Credential stores provided by Elytron

4.1.1.1. KeyStoreCredentialStore/credential-store

4.1.1.2. PropertiesCredentialStore/secret-key-credential-store

4.1.2. Credential types in Elytron

4.1.3. Credential types supported by Elytron credential stores

4.1.4. Credential store operations using the JBoss EAP management CLI

4.1.4.1. Creating a KeyStoreCredentialStore/credential-store for a standalone server

4.1.4.2. Creating a KeyStoreCredentialStore/credential-store for a managed domain

4.1.4.3. Creating a PropertiesCredentialStore/secret-key-credential-store for a standalone server

4.1.4.4. Adding a PasswordCredential to a KeyStoreCredentialStore/credential-store

4.1.4.5. Generating a SecretKeyCredential in a KeyStoreCredentialStore/credential-store

4.1.4.6. Generating a SecretKeyCredential in a PropertiesCredentialStore/secret-key-credential-store

4.1.4.7. Listing credentials in the KeyStoreCredentialStore/credential-store

4.1.4.8. Listing credentials in the PropertiesCredentialStore/secret-key-credential-store

4.1.4.9. Removing a credential from KeyStoreCredentialStore/credential-store

4.1.4.10. Removing a credential from the PropertiesCredentialStore/secret-key-credential-store

4.1.5. Credential store operations using the WildFly Elytron tool

4.1.5.1. Creating a KeyStoreCredentialStore/credential-store using the WildFly Elytron tool

4.1.5.2. Creating a KeyStoreCredentialStore/credential-store using the Bouncy Castle provider

4.1.5.3. Creating a PropertiesCredentialStore/secret-key-credential-store using WildFly Elytron tool

4.1.5.4. WildFly Elytron tool KeyStoreCredentialStore/credential-store operations

4.1.5.5. WildFly Elytron tool PropertiesCredentialStore/secret-key-credential-store operations

4.1.5.6. Adding a credential store created with the WildFly Elytron tool to a JBoss EAP Server

4.1.5.7. WildFly Elytron tool key pair management operations

4.1.5.8. Example use of stored key pair in the Elytron configuration files

4.1.5.9. Generating masked encrypted strings using the WildFly Elytron tool

4.1.6. Encrypted expressions in Elytron

4.1.6.1. Creating an encrypted expression in Elytron

4.1.6.2. Using a PasswordCredential in your JBoss EAP configuration

4.1.6.3. Using an encrypted expression to secure a KeyStoreCredentialStore/credential-store

4.1.6.4. Automatic update of credentials in credential store

4.1.7. Defining FIPS 140-2 compliant credential stores

4.1.7.1. Defining a FIPS 140-2 compliant credential store using an NSS database

4.1.7.2. Defining a FIPS 140-2 compliant credential store using Bouncy Castle providers

4.1.7.3. Using a custom implementation of the credential store

4.1.7.4. Obtain the password for the credential store from an external source

4.1.7.5. Providing an initial key to JBoss EAP to unlock secured resources

4.1.7.6. Creating a PropertiesCredentialStore/secret-key-credential-store for a standalone server
4.1.14.2. Creating an encrypted expression in Elytron
4.1.14.3. Using an encrypted expression to secure a KeyStoreCredentialStore/credential-store
4.1.15. Converting password vaults to credential stores
   4.1.15.1. Converting a single password vault to a Credential Store using the WildFly Elytron tool
   4.1.15.2. Bulk converting password vault to a credential store using the WildFly Elytron tool
4.1.16. Example of using a credential store with Elytron client
4.2. PASSWORD VAULT
   4.2.1. Set Up a Password Vault
   4.2.2. Initialize the Password Vault
   4.2.3. Use a Password Vault
   4.2.4. Store a Sensitive String in the Password Vault
   4.2.5. Use an Encrypted Sensitive String in Configuration
   4.2.6. Use an Encrypted Sensitive String in an Application
   4.2.7. Check if a Sensitive String is in the Password Vault
   4.2.8. Remove a Sensitive String from the Password Vault
      Remove a Sensitive String Interactively
   4.2.9. Configure Red Hat JBoss Enterprise Application Platform to Use a Custom Implementation of the Password Vault
   4.2.10. Obtain Keystore Password From External Source

CHAPTER 5. JAVA SECURITY MANAGER ................................................................. 221
5.1. ABOUT THE JAVA SECURITY MANAGER ............................................. 221
5.2. ABOUT JAVA SECURITY POLICY
   5.2.1. About defining policies in the security manager subsystem
   5.2.2. About defining policies in the deployment
   5.2.3. About defining policies in modules
5.3. RUN JBOSS EAP WITH THE JAVA SECURITY MANAGER
   5.3.1. Using the -secmgr flag with startup configuration script.
   5.3.2. Using the startup configuration file
5.4. CONSIDERATIONS BEFORE MOVING FROM PREVIOUS VERSIONS .......... 224

APPENDIX A. REFERENCE MATERIAL ............................................................ 225
A.1. ELYTRON SUBSYSTEM COMPONENTS REFERENCE ............................... 225
A.2. CONFIGURE YOUR ENVIRONMENT TO USE THE BOUNCYCASTLE PROVIDER 268
A.3. SASL AUTHENTICATION MECHANISMS REFERENCE
   A.3.1. Support Level for SASL Authentication Mechanisms
   A.3.2. SASL Authentication Mechanism Properties
A.4. SECURITY AUTHORIZATION ARGUMENTS
   Mechanism Specific Attributes
A.5. ELYTRON CLIENT SIDE ONE WAY EXAMPLE ...................................... 275
A.6. ELYTRON CLIENT SIDE TWO WAY EXAMPLE ..................................... 276
PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. To provide feedback, you can highlight the text in a document and add comments. Follow the steps in the procedure to learn about submitting feedback on Red Hat documentation.

Prerequisites

- Log in to the Red Hat Customer Portal.
- In the Red Hat Customer Portal, view the document in Multi-page HTML format.

Procedure

1. Click Feedback to see existing reader comments.

   NOTE
   
   The feedback feature is enabled only in the Multi-page HTML format.

2. Highlight the section of the document where you want to provide feedback.

3. In the prompt menu that displays near the text you selected, click Add Feedback.
   
   A text box opens in the feedback section on the right side of the page.

4. Enter your feedback in the text box and click Submit.
   
   You have created a documentation issue.

5. To view the issue, click the issue tracker link in the feedback view.
MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright’s message.
CHAPTER 1. SECURING THE SERVER AND ITS INTERFACES

1.1. BUILDING BLOCKS

1.1.1. Interfaces and socket bindings

JBoss EAP utilizes its host’s interfaces, for example **inet-address** and **nic**, as well as ports for communication for both its web applications as well as its management interfaces. These interfaces and ports are defined and configured through the **interfaces** and **socket-binding-groups** settings in the JBoss EAP.

For more information on how to define and configure **interfaces** and **socket-binding-groups**, see the **Socket Bindings** section of the JBoss EAP **Configuration Guide**.

Example: Interfaces

```xml
<interfaces>
  <interface name="management">
    <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>
  </interface>
  <interface name="public">
    <inet-address value="${jboss.bind.address:127.0.0.1}"/>
  </interface>
</interfaces>
```

Example: Socket Binding Group

```xml
<socket-binding-group name="standard-sockets" default-interface="public" port-offset="${jboss.socket.binding.port-offset:0}">
  <socket-binding name="management-http" interface="management" port="${jboss.management.http.port:9990}"/>
  <socket-binding name="management-https" interface="management" port="${jboss.management.https.port:9993}"/>
  <socket-binding name="ajp" port="${jboss.ajp.port:8009}"/>
  <socket-binding name="http" port="${jboss.http.port:8080}"/>
  <socket-binding name="https" port="${jboss.https.port:8443}"/>
  <socket-binding name="txn-recovery-environment" port="4712"/>
  <socket-binding name="txn-status-manager" port="4713"/>
  <outbound-socket-binding name="mail-smtp">
    <remote-destination host="localhost" port="25"/>
  </outbound-socket-binding>
</socket-binding-group>
```

1.1.2. Elytron Subsystem

1.1.2.1. Enable Elytron Security Across the Server

There is a simple way to enable Elytron across the server. JBoss EAP 7.1 introduced an example configuration script that enables Elytron as the security provider. This script resides in the **EAP_HOME/docs/examples** directory in the server installation.

Execute the following command to enable Elytron security across the server.

```bash
<interfaces>
  <interface name="management">
    <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>
  </interface>
  <interface name="public">
    <inet-address value="${jboss.bind.address:127.0.0.1}"/>
  </interface>
</interfaces>
```
$ EAP_HOME/bin/jboss-cli.sh --file=EAP_HOME/docs/examples/enable-elytron.cli

1.1.2.2. Create an Elytron Security Domain

Security domains in the `elytron` subsystem, when used in conjunction with security realms, are used for both core management authentication as well as for authentication with applications.

**IMPORTANT**

Deployments are limited to using one Elytron security domain per deployment. Scenarios that may have required multiple legacy security domains can now be accomplished using a single Elytron security domain.

**Add a Security Domain Using the Management CLI**

```
/subsystem=elytron/security-domain=domainName:add(realms=[{realm=realmName,role-decoder=roleDecoderName}],default-realm=realmName,permission-mapper=permissionMapperName,role-mapper=roleMapperName,...)
```

**Add a Security Domain Using the Management Console**

1. Access the management console. For more information, see the Management Console section in the JBoss EAP Configuration Guide.

2. Navigate to Configuration → Subsystems → Security (Elytron) → Other Settings and click View.

3. Select SSL → Security Domain and use the Add button to configure a new security domain.

1.1.2.3. Create an Elytron Security Realm

Security realms in the `elytron` subsystem, when used in conjunction with security domains, are used for both core management authentication as well as for authentication with applications. Security realms are also specifically typed based on their identity store, for example `jdbc-realm`, `filesystem-realm`, `properties-realm`, etc.

**Add a Security Realm Using the Management CLI**

```
/subsystem=elytron/type-of-realm=realmName:add(....)
```

Examples of adding specific realms, such as `jdbc-realm`, `filesystem-realm`, and `properties-realm` can be found in previous sections.

**Add a Security Realm Using the Management Console**

1. Access the management console. For more information, see the Management Console section in the JBoss EAP Configuration Guide.


3. Select the appropriate security realm type from the Security Realm tab and click Add to configure a new security realm.
1.1.2.4. Create an Elytron Role Decoder

A role decoder converts attributes from the identity provided by the security realm into roles. Role decoders are also specifically typed based on their functionality, for example empty-role-decoder, simple-role-decoder, and custom-role-decoder.

Add a Role Decoder Using the Management CLI

```
/subsystem=elytron/ROLE-DECODER-TYPE=roleDeoderName:add(....)
```

Add a Role Decoder Using the Management Console

1. Access the management console. For more information, see the Management Console section in the JBoss EAP Configuration Guide.


3. Click on Role Decoder, select the appropriate role decoder type and click Add to configure a new role decoder.

1.1.2.5. Adding a source-address-role-decoder to the elytron subsystem

You can use either the management CLI or the Management Console to add the source-address-role-decoder role decoder to the elytron subsystem. By configuring this role decoder in the mappers element, you make use of the IP address of a client when making authorization decisions.

The source-address-role-decoder extracts the IP address of a client and checks if it matches the IP address specified in the pattern attribute or the source-address attribute. If the IP address of the client matches the IP address specified in either attribute then elytron uses the roles attribute to assign roles to the user.

NOTE

The procedure uses the management CLI to add source-address-role-decoder to the mappers element in the elytron subsystem. If you want to use the Management Console to complete this task, refer to the link provided in the Additional resources section.

Prerequisites

- Note the IP address of the server’s client.

Procedure

1. In the elytron subsystem, use the management CLI to add source-address-role-decoder. For a source-address-role-decoder, you must specify an IP address and at least one role for a user.

Example of adding the source-address-role-decoder to the mappers element:

```
/subsystem=elytron/source-address-role-decoder=decoder1:add(source-address="10.10.10.10", roles=["Administrator"])
```

The example shows a configured source-address-role-decoder, named as decoder1. When a client attempts to connect to a server, the elytron subsystem uses the source-address-role-
decoder to check that the client’s IP address matches the IP address that was specified in either the pattern attribute or the source-address attribute. In the previous example, the source-address-role-decoder checks if the client’s IP address is 10.10.10.10. If the client’s IP address is 10.10.10.10 then elytron uses the roles attribute to assign the Administrator role to the user.

NOTE

You can configure a source-address-role-decoder to assign specific roles to a user who needs to establish connections from different networks.

2. In the security-domain, reference the configured source-address-role-decoder in the role-decoder attribute. This ensures that an Elytron security domain uses source-address-role-decoder when making authorization decisions.

Example of referencing a configured source-address-role-decoder, decoder1, in the role-decoder attribute:

```
/subsystem=elytron/security-domain=domainName:add(role-decoder=decoder1,default-realm=realmName,realms=[{realm=realmName}])
```

Additional resources

- For information about adding a role decoder with the management console, see Elytron Subsystem.
- For information about the elytron subsystem, see Elytron Subsystem in the Security Architecture guide.

1.1.3. Configuring an aggregate-role-decoder to the elytron subsystem

The aggregate-role-decoder consists of two or more role decoders. You can use an aggregate-role-decoder to aggregate the roles returned from each role decoder.

Prerequisites

- Configure at least two role decoders in the elytron subsystem.

Procedure

- Add at least two role decoders to the aggregate-role-decoder role decoder.

Example of adding decoder1 and decoder2 to the aggregate-role-decoder role decoder:

```
/subsystem=elytron/aggregate-role-decoder=aggregateDecoder:add(role-decoders=[decoder1, decoder2])
```

Additional resources

- For information about available role decoders in the elytron subsystem, see Resources in the Elytron Subsystem in the Security Architecture guide.
1.1.3.1. Create an Elytron Role Mapper

A role mapper maps roles after they have been decoded to other roles. Examples include normalizing role names or adding and removing specific roles from principals after they have been decoded. Role mappers are also specifically typed based on their functionality, for example add-prefix-role-mapper, add-suffix-role-mapper, and constant-role-mapper.

Adding a Role Mapper Takes the General Form

/subsystem=elytron/ROLE-MAPPER-TYPE=roleMapperName:add(…)

1.1.3.2. Create an Elytron Permission Set

Permission sets can be used to assign permissions to an identity.

Add a Permission Set Using the Management CLI

/subsystem=elytron/permission-set=PermissionSetName:add(permissions=[{class-name="...", module="...", target-name="...", action="..."}…])

The permissions parameter consists of a set of permissions, where each permission has the following attributes:

- class-name is the fully qualified class name of the permission. This is the only permission attribute that is required.
- module is an optional module used to load the permission.
- target-name is an optional target name passed to the permission as it is constructed.
- action is an optional action passed to the permission as it is constructed.

1.1.3.3. Create an Elytron Permission Mapper

In addition to roles being assigned to an identity, permissions may also be assigned. A permission mapper assigns permissions to an identity. Permission mappers are also specifically typed based on their functionality, for example logical-permission-mapper, simple-permission-mapper, and custom-permission-mapper.

Add a Permission Mapper Using the Management CLI

/subsystem=elytron/simple-permission-mapper=PermissionMapperName:add(…)

For information about creating a role decoder, see Elytron Subsystem.
Add a Permission Mapper Using the Management Console

1. Access the management console. For more information, see the Management Console section in the JBoss EAP Configuration Guide.


3. Click on Principal Decoder, select the appropriate principal decoder type and click Add to configure a new principal decoder.

1.1.3.4. Creating an Authentication Configuration

An authentication configuration contains the credentials to use when making a connection. For more information on authentication configurations, see Configure Client Authentication with Elytron Client in How to Configure Identity Management for JBoss EAP.

NOTE

Instead of a credential store, you can configure an Elytron security domain to use the credentials of the accessing user. For instance, a security domain can be used in conjunction with Kerberos for authenticating incoming users. Follow the instructions in Configure the Elytron Subsystem in How to Set Up SSO with Kerberos for JBoss EAP, and set obtain-kerberos-ticket=true in the Kerberos security factory.

Add an Authentication Configuration Using the Management CLI

/subsystem=elytron/authentication-configuration=AUTHENTICATION_CONFIGURATION_NAME:add(authentication-name=AUTHENTICATION_NAME, credential-reference={clear-text=PASSWORD})

Add an Authentication Configuration Using the Management Console

1. Access the management console. For more information, see the Management Console section in the JBoss EAP Configuration Guide.

2. Navigate to Configuration → Subsystems → Security (Elytron) → Other Settings and click View.

3. Click on Authentication → Authentication Configuration and click Add to configure a new authentication configuration.

For the full list of authentication-configuration attributes, see Elytron Subsystem Components Reference.

1.1.3.5. Creating an Authentication Context

An authentication context contains a set of rules and either authentication configurations or SSL contexts to use for establishing a connection. For more information on authentication contexts, see Configure Client Authentication with Elytron Client in How to Configure Identity Management for JBoss EAP.

Add an Authentication Context Using the Management CLI

An authentication context can be created using the following management CLI command.
Typically, an authentication context will contain a set of rules and either an authentication configuration or a SSL context. The following CLI command provides demonstrates defining an authentication context that only functions when the hostname is localhost.

/add subsystem=elytron/authentication-context=AUTHENTICATION_CONTEXT_NAME:add(match-rules=[{authentication-configuration=AUTHENTICATION_CONFIGURATION_NAME, match-host=localhost}])

Add an Authentication Context Using the Management Console

1. Access the management console. For more information, see the Management Console section in the JBoss EAP Configuration Guide.

2. Navigate to Configuration → Subsystems → Security (Elytron) → Other Settings and click View.

3. Click on Authentication → Authentication Context and click Add to configure a new authentication context.

For the full list of authentication-context attributes, see Elytron Subsystem Components Reference.

1.1.3.6. Create an Elytron Authentication Factory

An authentication factory is an authentication policy used for specific authentication mechanisms. Authentication factories are specifically based on the authentication mechanism, for example http-authentication-factory, sasl-authentication-factory and kerberos-security-factory.

Add an Authentication Factory Using the Management CLI

/add subsystem=elytron/AUTH-FACTORY-TYPE=authFactoryName:add(....)

Add an Authentication Factory Using the Management Console

1. Access the management console. For more information, see the Management Console section in the JBoss EAP Configuration Guide.


3. Click on HTTP Factories, SASL Factories, or Other Factories, choose the appropriate factory type, and click Add to configure a new factory.

1.1.3.7. Create an Elytron Keystore

A key-store is the definition of a keystore or truststore including the type of keystore, its location, and the credential for accessing it.

To generate an example keystore for use with the elytron subsystem, use the following command:

```bash
$ keytool -genkeypair -alias localhost -keyalg RSA -keysize 1024 -validity 365 -keystore keystore.jks -dname "CN=localhost" -keypass secret -storepass secret
```

Add a Keystore Using the Management CLI
To define a key-store in Elytron that references the newly made keystore, execute the following management CLI command. This command species the path to the keystore, relative to the file system path provided, the credential reference used for accessing the keystore, and the type of keystore.

/ subsystem=elytron/key-store=newKeyStore:add(path=keystore.jks,relative-to=jboss.server.config.dir,credential-reference={clear-text=secret},type=JKS)

NOTE

The above command uses relative-to to reference the location of the keystore file. Alternatively, you can specify the full path to the keystore in path and omit relative-to.

Add a Keystore Using the Management Console

1. Access the management console. For more information, see the Management Console section in the JBoss EAP Configuration Guide.

2. Navigate to Configuration → Subsystems → Security (Elytron) → Other Settings and click View.

3. Click on Stores → Key Store and click Add to configure a new keystore.

1.1.3.8. Create an Elytron Key Manager

A key-manager references a key-store, and is used in conjunction with an SSL context.

Add a Key Manager Using the Management CLI

The following command specifies the underlying keystore to reference, the algorithm to use when initializing the key manager, and the credential reference for accessing the entries in the underlying keystore.

/ subsystem=elytron/key-manager=newKeyManager:add(key-store=KEY_STORE,credential-reference={clear-text=secret})

IMPORTANT

Red Hat did not specify the algorithm attribute in the previous command, because the Elytron subsystem uses KeyManagerFactory.getDefaultAlgorithm() to determine an algorithm by default. However, you can specify the algorithm attribute. To specify the algorithm attribute, you need to know what key manager algorithms are provided by the JDK you are using. For example, a JDK that uses SunJSSE provides the PKIX and SunX509 algorithms.

In the previous command you can specify SunX509 as the key manager algorithm attribute.

Add a Key Manager Using the Management Console

1. Access the management console. For more information, see the Management Console section in the JBoss EAP Configuration Guide.

2. Navigate to Configuration → Subsystems → Security (Elytron) → Other Settings and click View.
3. Click on SSL → Key Manager and click Add to configure a new key manager.

1.1.3.9. Create an Elytron Truststore

To create a truststore in Elytron execute the following CLI command.

```
/subsystem=elytron/key-store=default-trust-store:add(type=JKS, relative-to=jboss.server.config.dir, path=application.truststore, credential-reference={clear-text=password})
```

In order to successfully execute the command above you must have an `application.truststore` file inside your EAP_HOME/standalone/configuration directory. The truststore must contain the certificates associated with the endpoint or a certificate chain in case the end point’s certificate is signed by a CA.

Red Hat recommends you to avoid using self-signed certificates. Ideally, certificates should be signed by a CA and your truststore should contain a certificate chain representing your ROOT and intermediary CAs.

1.1.3.10. Create an Elytron Trust Manager

To define a trust manager in Elytron execute the following CLI command.

```
/subsystem=elytron/trust-manager=default-trust-manager:add(key-store=TRUST-STORE-NAME)
```

This sets the defined truststore as the source of the certificates that the application server trusts.

1.1.3.11. Using the Out of the Box Elytron Components

JBoss EAP provides a default set of Elytron components configured in the `elytron` subsystem. You can find more details on these pre-configured components in the Out of the Box section of the Security Architecture guide.

1.1.3.11.1. Securing Management Interfaces

You can find more details on the enabling JBoss EAP to use the out of the box Elytron components for securing the management interfaces in the User Authentication with Elytron section.

1.1.3.11.2. Securing Applications

The `elytron` subsystem provides `application-http-authentication` for `http-authentication-factory` by default, which can be used to secure applications. For more information on how to configure `application-http-authentication`, see the Out of the Box section of the Security Architecture guide.

To configure applications to use `application-http-authentication`, see Configure Web Applications to Use Elytron or Legacy Security for Authentication in How to Configure Identity Management Guide. You can also override the default behavior of all applications using the steps in the Override an Application’s Authentication Configuration section of the JBoss EAP How to Configure Identity Management Guide.

1.1.3.11.3. Using SSL/TLS

JBoss EAP does provide a default one-way SSL/TLS configuration using the legacy core management authentication, but it does not provide one in the `elytron` subsystem. You can find more details on configuring SSL/TLS using the `elytron` subsystem for both the management interfaces as well as for applications in the following sections:
- Enable One-way SSL/TLS for the Management Interfaces Using the Elytron Subsystem
- Enable Two-Way SSL/TLS for the Management Interfaces using the Elytron Subsystem
- Enable One-way SSL/TLS for Applications using the Elytron Subsystem
- Enable Two-Way SSL/TLS for Applications using the Elytron Subsystem

1.1.3.11.4. Using Elytron with Other Subsystems

In addition to securing applications and management interfaces, Elytron also integrates with other subsystems in JBoss EAP.

**batch-jberet**

You can configure the **batch-jberet** subsystem to run batch jobs using an Elytron security domain. For more information, see [Configure Security for Batch Jobs](#) in the *Configuration Guide*.

**datasources**

You can use a credential store or an Elytron security domain to provide authentication information in a datasource definition. For more information, see [Datasource Security](#) in the *Configuration Guide*.

**ejb3**

You can create mappings for Elytron security domains in the **ejb3** subsystem to be referenced by deployments. For more information, see [Elytron Integration with the EJB Subsystem](#) in *Developing Jakarta Enterprise Beans Applications*.

**iiop-openjdk**

You can use the **elytron** subsystem to configure SSL/TLS between clients and servers using the **iiop-openjdk** subsystem. For more information, see [Configure IIOP to use SSL/TLS with the Elytron Subsystem](#) in the *Configuration Guide*.

**jca**

You can use the **elytron-enabled** attribute to enable Elytron security for a work manager. For more information, see [Configuring the JCA Subsystem](#) in the *Configuration Guide*.

**jgroups**

You can configure the **SYM_ENCRYPT** and **ASYM_ENCRYPT** protocols to reference keystores or credential references defined in the **elytron** subsystem. For more information, see [Securing a Cluster](#) in the *Configuration Guide*.

**mail**

You can use a credential store to provide authentication information in a server definition in the **mail** subsystem. For more information, see [Use a Credential Store for Passwords](#) in the *Configuration Guide*.

**messaging-activemq**

You can secure remote connections to the remote connections used by the **messaging-activemq** subsystem. For more information, see the **Using the Elytron Subsystem** section of *Configuring Messaging*.

**modcluster**

You can use an Elytron client **ssl-context** to communicate with a load balancer using SSL/TLS. For more information, see [Elytron Integration with the ModCluster Subsystem](#).

**remoting**

You can configure inbound and outbound connections in the **remoting** subsystem to reference authentication contexts, SASL authentication factories, and SSL contexts defined in the **elytron** subsystem. For full details on configuring each type of connection, see [Elytron Integration with the](#)
Remoting Subsystem.

You can secure connections to the resource adapter using Elytron. You can enable security inflow to establish security credentials when submitting work to be executed by the work manager. For more information, see Configure Resource Adapters to Use the Elytron Subsystem in the Configuration Guide.

undertow

You can use the elytron subsystem to configure both SSL/TLS and application authentication. For more information on configuring application authentication, see Using SSL/TLS and Configure Web Applications to Use Elytron or Legacy Security for Authentication in How to Configure Identity Management.

1.1.3.12. Enable and Disable the Elytron Subsystem

The elytron subsystem comes pre-configured with the default JBoss EAP profiles alongside the legacy security subsystem.

If you are using a profile where the elytron subsystem has not been configured, you can add it by adding the elytron extension and enabling the elytron subsystem.

To add the elytron extension required for the elytron subsystem:

```
(extension=org.wildfly.extension.elytron:add)
```

To enable the elytron subsystem in JBoss EAP:

```
(subsystem=elytron:add
 reload
```

To disable the elytron subsystem in JBoss EAP:

```
(subsystem=elytron:remove
 reload
```

IMPORTANT

Other subsystems within JBoss EAP may have dependencies on the elytron subsystem. If these dependencies are not resolved before disabling it, you will see errors when starting JBoss EAP.

1.1.4. Legacy Security Subsystem

1.1.4.1. Disabling the security subsystem

You can disable the security subsystem in JBoss EAP by executing the remove operation of the subsystem.

Procedure

- Disable the security subsystem in JBoss EAP:
IMPORTANT

Other subsystems within JBoss EAP may have dependencies on the `security` subsystem. If these dependencies are not resolved before disabling it, you will see errors when starting JBoss EAP.

1.1.4.2. Enabling the security subsystem

You can enable the `security` subsystem in JBoss EAP by executing the `add` operation of the subsystem.

**Procedure**

- Enable the `security` subsystem in JBoss EAP:

  ```
  /subsystem=security:add
  ```

1.1.5. Legacy security realms

JBoss EAP uses security realms to define authentication and authorization mechanisms, such as local, LDAP properties, which can then be used by the management interfaces.

**Example: Security realms**

```xml
<security-realms>
  <security-realm name="ManagementRealm">
    <authentication>
      <local default-user="$local" skip-group-loading="true"/>
      <properties path="mgmt-users.properties" relative-to="jboss.server.config.dir"/>
    </authentication>
    <authorization map-groups-to-roles="false">
      <properties path="mgmt-groups.properties" relative-to="jboss.server.config.dir"/>
    </authorization>
  </security-realm>

  <security-realm name="ApplicationRealm">
    <authentication>
      <local default-user="$local" allowed-users="*" skip-group-loading="true"/>
      <properties path="application-users.properties" relative-to="jboss.server.config.dir"/>
    </authentication>
    <authorization>
      <properties path="application-roles.properties" relative-to="jboss.server.config.dir"/>
    </authorization>
  </security-realm>
</security-realms>
```
**NOTE**

In addition to updating the existing security realms, JBoss EAP also allows you to create new security realms. You can create new security realms via the management console as well as invoking the following command from the management CLI:

```
/core-service=management/security-realm=<new_realm_name>:add()
```

If you create a new security realm and want to use a properties file for authentication or authorization, you must create a new properties file specifically for the new security domain. JBoss EAP does not reuse existing files used by other security domains nor does it automatically create new files specified in the configuration if they do not exist.

**Additional resources**

- For more information on security realms, see [Security Realms](#).

### 1.1.6. Using authentication and socket bindings for securing the management interfaces

You can use a combination of **socket-binding**, **http-authentication-factory**, and **http-upgrade** to secure the management interfaces using the **elytron** subsystem. Alternatively, you can use **socket-binding with security-realm** to secure the management interfaces with the legacy core management authentication. You can also disable the management interfaces, and configure users of the interfaces to have various roles and access rights.

By default, JBoss EAP defines an **http-interface** to connect to the management interfaces.

**Procedure**

- Display server management interfaces settings:

  ```
  [standalone@localhost:9990 ] /core-service=management:read-resource(recursive=true)
  {
    "outcome" => "success",
    "result" => {
      "access" => {...},
      "ldap-connection" => undefined,
      "management-interface" => {
        "http-interface" => {
          "allowed-origins" => undefined,
          "console-enabled" => true,
          "http-authentication-factory" => "management-http-authentication",
          "http-upgrade" => {
            "enabled" => true,
            "sasl-authentication-factory" => "management-sasl-authentication"
          },
          "http-upgrade-enabled" => true,
          "sasl-protocol" => "remote",
          "secure-socket-binding" => undefined,
          "security-realm" => undefined,
          "server-name" => undefined,
          "socket-binding" => "management-http",
          "ssl-context" => undefined
        }
      }
    },
    "security-realm" => {...},
  }
  ```

[23]
1.2. HOW TO SECURE THE MANAGEMENT INTERFACES

The following sections show how to perform various operations related to securing the JBoss EAP management interfaces and related subsystems.

NOTE

The management CLI commands shown assume that you are running a JBoss EAP standalone server. For more details on using the management CLI for a JBoss EAP managed domain, see the JBoss EAP Management CLI Guide.

Elytron Integration with the Management CLI

The management interfaces can be secured using resources from the elytron subsystem in the same way as it is done by the legacy security realms.

The SSL configuration for connections comes from one of these locations:

- Any SSL configuration within the CLI specific configuration.
- The default SSL configuration that automatically prompts users to accept the server's certificate.
- The java system property.

Client configuration can be modified using the wildfly-config.xml file.

NOTE

If you set the -Dwildfly.config.url property, any file can be used by the client for configuration.

1.2.1. Configure networking and ports

Depending on the configuration of the host, JBoss EAP may be configured to use various network interfaces and ports. This allows JBoss EAP to work with different host, networking, and firewall requirements.

Additional resources

- For more information on the networking and ports used by JBoss EAP, as well as how to configure these settings, see the Network and Port Configuration section of the JBoss EAP Configuration Guide.

1.2.2. Disabling the management console

Other clients, such as JBoss Operations Network, operate using the HTTP interface for managing JBoss EAP. In order to continue using these services, just the web-based management console itself may be disabled. This is accomplished by setting the console-enabled attribute to false.
Procedure

- To disable the web-based management console in JBoss EAP:

```
/core-service=management/management-interface=http-interface/:write-attribute(name=console-enabled,value=false)
```

1.2.3. Disabling remote access to JMX

Remote access to the jmx subsystem allows for JDK and application management operations to be triggered remotely.

Procedure

- To disable remote access to JMX in JBoss EAP, remove the remoting connector in the jmx subsystem:

```
/subsystem=jmx/remoting-connector=jmx/ remove
```

1.2.4. Silent authentication

The default installation of JBoss EAP contains a method of silent authentication for a local management CLI user. This allows the local user the ability to access the management CLI without username or password authentication. This functionality can be enabled to allow local users run the management CLI scripts without requiring authentication. It is considered a useful feature given that access to the local configuration typically also gives the user the ability to add their own user details or otherwise disable security checks.

Silent authentication can be disabled where greater security control is required. This can be achieved by removing the local element within the security-realm attribute of the configuration file. This is applicable to both standalone instance as well as managed domain.

**IMPORTANT**

The removal of the local element should only be done if the impact on the JBoss EAP instance and its configuration is fully understood.

Procedure

1. To remove silent authentication when using the elytron subsystem:

```
[standalone@localhost:9990 ] /subsystem=elytron/sasl-authentication-factory=managenet-sasl-authentication:read-resource
{
    "outcome" => "success",
    "result" => {
        "mechanism-configurations" => [
            { "mechanism-name" => "JBOSS-LOCAL-USER",
                "realm-mapper" => "local"
            },
            { "mechanism-name" => "DIGEST-MD5",
                "mechanism-realm-configurations" => [{"realm-name" => "ManagementRealm"}]
            }
        ]
    }
}```
2. To remove silent authentication when using a legacy security realm:

/core-service=management/security-realm=<realm_name>/authentication=local:remove

1.2.5. One-way SSL/TLS for the management interfaces using the Elytron subsystem

In JBoss EAP, you can enable one-way SSL/TLS for the management interfaces using the JBoss EAP management CLI or the management console.

In the management CLI, one-way SSL/TLS can be enabled in two ways:

- Using security command.
- Using elytron subsystem commands.

In the management console, one-way SSL/TLS can be enabled in as follows:

- Using the management console

1.2.5.1. Enabling one-way SSL/TLS using a security command

The security enable-ssl-management command can be used to enable one-way SSL/TLS for the management interfaces.

Procedure

- Enter the security enable-ssl-management --interactive command in the CLI.

Example

security enable-ssl-management --interactive

Please provide required pieces of information to enable SSL:
Key-store file name (default management.keystore): keystore.jks
Password (blank generated): secret
What is your first and last name? [Unknown]: localhost
What is the name of your organizational unit? [Unknown]:
What is the name of your organization? [Unknown]:
What is the name of your City or Locality? [Unknown]:
What is the name of your State or Province? [Unknown]:
What is the two-letter country code for this unit? [Unknown]:
Is CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correct y/n [y]? 
Validity (in days, blank default): 365 
Alias (blank generated): localhost 
Enable SSL Mutual Authentication y/n (blank n): n 

SSL options: 
key store file: keystore.jks 
distinguished name: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown 
password: secret 
validity: 365 
alias: localhost 
Server keystore file keystore.jks, certificate file keystore.pem and keystore.csr file will be generated in server configuration directory. 
Do you confirm y/n :y 

NOTE 
- Once the command is executed, the management CLI will reload the server and reconnect to it. 
- You can disable one-way SSL/TLS for the management interfaces using the disable-ssl-management command. 

security disable-ssl-management 

This command does not delete the Elytron resources. It configures the system to use the ApplicationRealm legacy security realm for its SSL configuration.

1.2.5.2. Enabling one-way SSL/TLS using the Elytron subsystem commands 
You can enable one-way SSL/TLS for the management interfaces using the elytron subsystem commands.

Procedure
1. Configure a key-store. 

/subsystem=elytron/key-store=httpsKS:add(path=keystore.jks,relative-to=jboss.server.config.dir,credential-reference={clear-text=secret},type=JKS) 

NOTE 
The above command uses relative-to to reference the location of the keystore file. Alternatively, you can specify the full path to the keystore in path and omit relative-to.

If the keystore file does not exist yet, the following commands can be used to generate an example key pair:

/subsystem=elytron/key-store=httpsKS:generate-key-pair(alias=localhost,algorithm=RSA,key-size=1024,validity=365,credential-reference={clear-
2. Create a **key-manager** and **server-ssl-context**.

```plaintext
/subsystem=elytron/key-manager=httpsKM:add(key-store=httpsKS,credential-reference=
{clear-text=secret})
/subsystem=elytron/server-ssl-context=httpsSSC:add(key-manager=httpsKM,protocols=
["TLSv1.2"])
```

**IMPORTANT**

Red Hat did not specify the algorithm attribute in the previous command, because the Elytron subsystem uses `KeyManagerFactory.getDefaultAlgorithm()` to determine an algorithm by default. However, you can specify the algorithm attribute. To specify the algorithm attribute, you need to know what key manager algorithms are provided by the JDK you are using. For example, a JDK that uses SunJSSE provides the PKIX and SunX509 algorithms.

In the previous command you can specify SunX509 as the key manager algorithm attribute.

You also need to determine what HTTPS protocols you want to support. The example commands above use TLSv1.2.

You can use the `cipher-suite-filter` to specify cipher suites, and the `use-cipher-suites-order` argument to honor server cipher suite order. The `use-cipher-suites-order` attribute by default is set to `true`. This differs from the legacy security subsystem behavior, which defaults to honoring client cipher suite order.

3. Enable HTTPS on the management interface.

```plaintext
/core-service=management/management-interface=http-interface:write-attribute(name=ssl-context, value=httpsSSC)
/core-service=management/management-interface=http-interface:write-attribute(name=secure-socket-binding, value=management-https)
```

4. Reload the JBoss EAP instance.

```plaintext
reload
```

One-way SSL/TLS is now enabled for the management interfaces.

**IMPORTANT**

In cases where you have both a **security-realm** and **ssl-context** defined, JBoss EAP will use the SSL/TLS configuration provided by **ssl-context**.

**Additional resources**

Red Hat JBoss Enterprise Application Platform 7.4 How to Configure Server Security
1.2.5.3. Enabling one-way SSL/TLS using the management console

You can enable SSL for the management interface used by the management console using an SSL wizard in the management console.

Procedure

1. Access the management console. For more information, see the Management Console section in the JBoss EAP Configuration Guide.
2. Navigate to Runtime, click the appropriate server name.
3. Click View next to server name.
4. Click HTTP Management... to open the HTTP Management Interface configuration page.
5. Click Enable SSL to launch the wizard.
   The wizard guides you through the following scenarios for enabling SSL:
   - You want to create a certificate store and generate a self-signed certificate.
   - You want to obtain a certificate from Let’s Encrypt Certificate Authority.
   - You already have the certificate store on the file system, but no keystore configuration.
   - You already have a keystore configuration that uses a valid certificate store.

Using the wizard, you can optionally create a truststore for mutual authentication.

1.2.6. Two-way SSL/TLS for the management interfaces using the Elytron Subsystem

In JBoss EAP, two-way SSL/TLS for the management interfaces can be enabled either by using a security command or by using the elytron subsystem commands.

To enable two-way SSL/TLS, first you must obtain or generate a client certificate. You can generate a client certificate by using the following procedure:

- Generating client certificates

You can then enable two-way SSL/TLS for the management interfaces using one of the following methods:

- Enabling two-way SSL/TLS using a security command
- Enabling two-way SSL/TLS using the Elytron subsystem commands

1.2.6.1. Generating client certificates

You can generate client certificates using the keytool command in the CLI.
Procedure

1. Generate your client certificate:

   $ keytool -genkeypair -alias client -keyalg RSA -keysize 1024 -validity 365 -keystore client.keystore.jks -dname "CN=client" -keypass secret -storepass secret

2. Export the client certificate:

   $ keytool -exportcert -keystore client.keystore.jks -alias client -keypass secret -storepass secret -file /path/to/client.cer

1.2.6.2. Enabling two-way SSL/TLS using a security command

The `security enable-ssl-management` command can be used to enable two-way SSL/TLS for the management interfaces.

**NOTE**

The following example does not validate the certificate as no chain of trust exists. If you are using a trusted certificate, then the client certificate can be validated without issue.

Prerequisites

- You have configured a client keystore.
- You have exported a certificate for a server trust store.
  
  For more information, see Generating client certificates.

Procedure

- Enter the `security enable-ssl-management --interactive` command in the CLI.

Example

```
security enable-ssl-management --interactive

Please provide required pieces of information to enable SSL:
Key-store file name (default management.keystore): server.keystore.jks
Password (blank generated): secret
What is your first and last name? [Unknown]: localhost
What is the name of your organizational unit? [Unknown]:
What is the name of your organization? [Unknown]:
What is the name of your City or Locality? [Unknown]:
What is the name of your State or Province? [Unknown]:
What is the two-letter country code for this unit? [Unknown]:
Is CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correct y/n [y]? y
Validity (in days, blank default): 365
Alias (blank generated): localhost
Enable SSL Mutual Authentication y/n (blank n): y
Client certificate (path to pem file): /path/to/client.cer
Validate certificate y/n (blank y): n
Trust-store file name (management.truststore): server.truststore.jks
```
Password (blank generated): secret

SSL options:
key store file: server.keystore.jks
distinguished name: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown,
C=Unknown
password: secret
validity: 365
alias: localhost
client certificate: /path/to/client.cer
trust store file: server.trustore.jks
trust store password: secret

Server keystore file server.keystore.jks, certificate file server.pem and server.csr file will be
generated in server configuration directory.
Server truststore file server.trustore.jks will be generated in server configuration directory.
Do you confirm y/n: y

NOTE

- Once the command is executed, the management CLI will reload the server
  and attempt to reconnect to it.
  To complete the two-way SSL/TLS authentication, you need to import the
  server certificate into the client truststore and configure your client to
  present the client certificate.

- You can disable two-way SSL/TLS for the management interfaces using the
  `disable-ssl-management` command.

security disable-ssl-management

This command does not delete the Elytron resources. It configures the
system to use the `ApplicationRealm` legacy security realm for its SSL
configuration.

1.2.6.3. Enabling two-way SSL/TLS using the Elytron subsystem commands

You can use the `elytron` subsystem commands to enable two-way SSL/TLS for the management
interfaces.

Prerequisites

- You have exported a certificate for a server trust store.
  For more information, see Generating client certificates.

Procedure

1. Obtain or generate your keystore.
   Before enabling one-way SSL/TLS in JBoss EAP, you must obtain or generate the keystores,
   truststores and certificates you plan on using. To generate an example set of keystores,
   truststores, and certificates, use the following commands.

   a. Configure a `key-store`.

      
      /subsystem=elytron/key-store=twoWayKS:add(path=server.keystore.jks,relative-
b. Export your server certificate.

```
/subsystem=elytron/key-store=twoWayKS:export-certificate(alias=localhost,path=/path/to/server.cer,pem=true)
```

b. Create a key-store for the server trust store and import the client certificate into the server truststore.

```
/subsystem=elytron/key-store=twoWayTS:add(path=server.truststore.jks,relative-to=jboss.server.config.dir,credential-reference={clear-text=secret},type=JKS)
/subsystem=elytron/key-store=twoWayTS:import-certificate(alias=client,path=/path/to/client.cer,credential-reference={clear-text=secret},trust-cacerts=true,validate=false)
/subsystem=elytron/key-store=twoWayTS:store()
```

2. Configure a key-manager, trust-manager, and server-ssl-context for the server keystore and truststore.

```
/subsystem=elytron/key-manager=twoWayKM:add(key-store=twoWayKS,credential-reference={clear-text=secret})
/subsystem=elytron/trust-manager=twoWayTM:add(key-store=twoWayTS)
```
Red Hat did not specify the algorithm attribute in the previous command, because the Elytron subsystem uses `KeyManagerFactory.getDefaultAlgorithm()` and `TrustManagerFactory.getDefaultAlgorithm()` to determine an algorithm by default. However, you can specify the algorithm attribute. To specify the algorithm attribute, you need to know what key manager algorithms are provided by the JDK you are using. For example, a JDK that uses SunJSSE provides the PKIX and SunX509 algorithms.

In the previous command you can specify SunX509 as the key manager algorithm attribute and PKIX as the trust manager algorithm attribute.

You also need to determine what HTTPS protocols you want to support. The example commands above use TLSv1.2.

You can use the `cipher-suite-filter` to specify cipher suites, and the `use-cipher-suites-order` argument to honor server cipher suite order. The `use-cipher-suites-order` attribute by default is set to `true`. This differs from the legacy security subsystem behavior, which defaults to honoring client cipher suite order.

3. Enable HTTPS on the management interface.

```
/core-service=management/management-interface=http-interface:write-attribute(name=ssl-context, value=twoWaySSC)
/core-service=management/management-interface=http-interface:write-attribute(name=secure-socket-binding, value=management-https)
```

4. Reload the JBoss EAP instance.

```
reload
```

NOTE

To complete the two-way SSL/TLS authentication, you need to import the server certificate into the client truststore and configure your client to present the client certificate.

5. Configure your client to use the client certificate.

You need to configure your client to present the trusted client certificate to the server to complete the two-way SSL/TLS authentication. For example, if using a browser, you need to import the trusted certificate into the browser’s trust store.

This results in a forced two-way SSL/TLS authentication, without changing the original authentication to the server management.

If you want to change the original authentication method, see Configure Authentication with Certificates in How to Configure Identity Management for JBoss EAP.

Two-way SSL/TLS is now enabled for the management interfaces.
IMPORTANT

In cases where you have both a security-realm and ssl-context defined, JBoss EAP will use the SSL/TLS configuration provided by ssl-context.

Additional resources

- key-store Attributes
- key-manager Attributes
- server-ssl-context Attributes
- trust-manager Attributes

1.2.7. SASL authentication for the management interfaces using the CLI security command

You can use the CLI security command to enable and disable SASL authentication for the management interfaces. You can also use the command to reorder SASL mechanisms.

Enable SASL authentication

In JBoss EAP, SASL authentication, using an elytron SASL authentication factory, can be enabled for the management interfaces with the security enable-sasl-management command. This command creates all of the non-existing resources required to configure authentication. By default this command associates the included SASL factory with the http-interface.

Example: Enable SASL Authentication

```bash
security enable-sasl-management

Server reloaded.
Command success.
Authentication configured for management http-interface
sasl authentication-factory=management-sasl-authentication
security-domain=ManagementDomain
```

NOTE

Once the command is executed, the management CLI will reload the server and reconnect to it.

If a SASL factory already exists, then the factory is updated to use the mechanism defined by the --mechanism argument.

For a list of arguments, see Authorization Security Arguments.

Disable SASL authentication

To remove the active SASL authentication factory use the following command:

```bash
security disable-sasl-management
```

Alternatively, to remove specific mechanisms from the active SASL authentication factory, use the following command:
security disable-sasl-management --mechanism=MECHANISM

Reorder SASL mechanisms
The order of defined SASL mechanisms dictate how the server authenticates the request, with the first matching mechanism being sent to the client.

You can change this order by passing a comma-separated to to the security reorder-sasl-management command, for example:

security reorder-sasl-management --mechanisms-order=MECHANISM1,MECHANISM2,...

Additional resources
- Security Authorization Arguments

1.2.8. HTTP authentication for the management interfaces using the CLI security command

You can use the CLI security command to enable and disable HTTP authentication for the management interfaces.

Enable HTTP authentication
In JBoss EAP, HTTP authentication, using an elytron HTTP authentication factory, can be enabled for the management interfaces with the security enable-http-auth-management command. This command can only target the http-interface, and with no additional arguments the included HTTP authentication factory will be associated with this interface.

Example: Enable HTTP Authentication

security enable-http-auth-management

Server reloaded.
Command success.
Authentication configured for management http-interface
http authentication-factory=management-http-authentication
security-domain=ManagementDomain

NOTE
Once the command is executed, the management CLI will reload the server and reconnect to it.

If an HTTP factory already exists, then the factory is updated to use the mechanism defined by the --mechanism argument.

For a list of arguments, see Authorization Security Arguments.

Disable HTTP Authentication
To remove the active HTTP authentication factory use the following command.

security disable-http-auth-management
Alternatively, you can use the following command to remove specific mechanisms from the active HTTP authentication factory.

```
security disable-http-auth-management --mechanism=MECHANISM
```

**Additional resources**

- Authorization Security Arguments

**1.2.9. Configuring the management interfaces for one-way SSL/TLS with legacy core management authentication**

Configuring the JBoss EAP management interfaces for communication only using one-way SSL/TLS provides increased security. All network traffic between the client and the management interfaces is encrypted, which reduces the risk of security attacks such as a man-in-the-middle attack.

In this procedure unencrypted communication with the JBoss EAP instance is disabled. This procedure applies to both standalone server and managed domain configurations. For a managed domain, prefix the management CLI commands with the name of the host, for example: `/host=master`.

**IMPORTANT**

While performing the steps for enabling one-way SSL/TLS on the management interfaces, do not reload the configuration unless explicitly instructed. Doing so may cause you to be locked out of the management interfaces.

1. Create a keystore to secure the management interfaces.  
   For more information, see [Creating a keystore to secure the management interfaces](#).

2. Ensure the management interfaces bind to HTTPS.  
   For more information, see [Ensuring the management interfaces bind to HTTPS](#).

3. Optional: Implement a custom [socket-binding-group](#).  
   For more information, see [Custom socket-binding-group](#).

4. Create a new security realm.  
   For more information, see [Creating a new security realm](#).

5. Configure the management interfaces to use the new security realm.  
   For more information, see [Configuring the management interfaces to use a security realm](#).

6. Configure the management interfaces to use the keystore.  
   For more information, see [Configuring the management interfaces to use a keystore](#).

7. Update the [jboss-cli.xml](#).  
   For more information, see [Updating the jboss-cli.xml file](#).

**1.2.9.1. Creating a keystore to secure the management interfaces**

Create a keystore to secure the management interfaces.

This keystore must be in JKS format as the management interfaces are not compatible with keystores in JCEKS format.
Procedure

- Create a keystore using the following CLI command:
  Replace the example values for the parameters, for example alias, keypass, keystore, storepass and dname, with the correct values for the environment.

```bash
```

**NOTE**
The parameter **validity** specifies for how many days the key is valid. A value of **730** equals two years.

### 1.2.9.2. Ensuring the management interfaces bind to HTTPS

Configure JBoss EAP to ensure management interfaces bind to HTTPS.

**Procedure**

- Configuration when running a Standalone Server
  To ensure the management interfaces bind to HTTPS, you must add the **management-https** configuration and remove the **management-http** configuration.

  Use the following CLI commands to bind the management interfaces to HTTPS:

  ```bash
  /core-service=management/management-interface=http-interface:write-attribute(name=secure-socket-binding, value=management-https)
  /core-service=management/management-interface=http-interface:undefine-attribute(name=socket-binding)
  ```

- Configuration when running a Managed Domain
  Change the socket element within the **management-interface** attribute by adding **secure-port** and removing port configuration.

  Use the following commands to bind the management interfaces to HTTPS:

  ```bash
  /host=master/core-service=management/management-interface=http-interface:write-attribute(name=secure-port, value=9993)
  /host=master/core-service=management/management-interface=http-interface:undefine-attribute(name=port)
  ```

### 1.2.9.3. Custom socket-binding-group

If you want to use a custom **socket-binding-group**, you must ensure the **management-https** binding is defined, which by default is bound to port **9993**. You can verify this from the **socket-binding-group** attribute of the server’s configuration file or using the management CLI:

```bash
/socket-binding-group=standard-sockets/socket-binding=management-https:read-resource(recursive=true)
```
1.2.9.4. Creating a new security realm

Create a new security realm.

In this procedure, the new security realm using HTTPS, ManagementRealmHTTPS, uses a properties file named https-mgmt-users.properties located in the EAP_HOME/standalone/configuration/ directory for storing user names and passwords.

Procedure

1. Create a properties file for storing user name and passwords. User names and passwords can be added to the file later, but for now, you need to create an empty file named https-mgmt-users.properties and save it to that location. The below example shows using the touch command, but you may also use other mechanisms, such as a text editor.

   Example: Using the touch Command to Create an Empty File

   $ touch EAP_HOME/standalone/configuration/https-mgmt-users.properties

2. Next, use the following management CLI commands to create a new security realm named ManagementRealmHTTPS:

   /core-service=management/security-realm=ManagementRealmHTTPS:add

   /core-service=management/security-realm=ManagementRealmHTTPS/authentication=properties:add(path=https-mgmt-users.properties,relative-to=jboss.server.config.dir)

3. Add users to the properties file. At this point, you have created a new security realm and configured it to use a properties file for authentication. You must now add users to that properties file using the add-user script, which is available in the EAP_HOME/bin/ directory. When running the add-user script, you must specify both the properties file and the security realm using the -up and -r options respectively. From there, the add-user script will interactively prompt you for the user name and password information to store in the https-mgmt-users.properties file.

   $ EAP_HOME/bin/add-user.sh -up EAP_HOME/standalone/configuration/https-mgmt-users.properties -r ManagementRealmHTTPS

   ...
Enter the details of the new user to add.  
Using realm 'ManagementRealmHTTPS' as specified on the command line.  

...  
Username : httpUser  
Password requirements are listed below. To modify these restrictions edit the add-user.properties configuration file.  
- The password must not be one of the following restricted values {root, admin, administrator}  
- The password must contain at least 8 characters, 1 alphabetic character(s), 1 digit(s), 1 non-alphanumeric symbol(s)  
- The password must be different from the username  
...  
Password :  
Re-enter Password :  
About to add user 'httpUser' for realm 'ManagementRealmHTTPS'  
...  
Is this correct yes/no? yes  
...  
Added user 'httpUser' to file 'EAP_HOME/configuration/https-mgmt-users.properties'  
...  
Is this new user going to be used for one AS process to connect to another AS process? e.g. for a slave host controller connecting to the master or for a Remoting connection for server to server EJB calls.  
yes/no? no

**IMPORTANT**

When configuring security realms that use properties files to store usernames and passwords, it is recommended that each realm use a distinct properties file that is not shared with another realm.

### 1.2.9.5. Configuring the management interfaces to use a security realm

You can configure the management interfaces to use a security realm by using a management CLI command.

**Procedure**

- Use the following management CLI command:

```
/core-service=management/management-interface=http-interface:write-attribute(name=security-realm,value=ManagementRealmHTTPS)
```

### 1.2.9.6. Configuring the management interfaces to use a keystore

Configure the management interfaces to use a keystore by using management CLI commands.

**Procedure**

1. Use the following management CLI command to configure the management interfaces to use the keystore.  
   For the parameters file, password and alias their values must be copied from the Create a Keystore to Secure the Management Interfaces step.
NOTE

To update the keystore password, use the following CLI command:

```
/core-service=management/security-realm=ManagementRealmHTTPS/server-
identity=ssl:write-attribute(name=keystore-password,value=newpassword)
```

2. Reload the server’s configuration:

```
reload
```

After reloading the server configuration, the log should contain the following, just before the text which states the number of services that are started:

```
13:50:54,160 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0061: Http
management interface listening on https://127.0.0.1:9993/management
13:50:54,162 INFO  [org.jboss.as] (Controller Boot Thread) WFLYSRV0052: Admin console
listening on https://127.0.0.1:9993
```

The management interfaces are now listening on port 9993, which confirms that the procedure was successful.

IMPORTANT

At this point, the CLI will disconnect and will be unable to reconnect since the port bindings have changed.

Proceed to the next step to update the `jboss-cli.xml` file to allow the management CLI to reconnect.

1.2.9.7. Updating the `jboss-cli.xml` file

If using the management CLI to perform management actions, you must update the `EAP_HOME/bin/jboss-cli.xml` file.

Procedure

- Update the `EAP_HOME/bin/jboss-cli.xml` file as following:
  - Update the value of `<default-protocol>` to `https-remoting`.
  - In `<default-controller>`, update the value of `<protocol>` to `https-remoting`.
  - In `<default-controller>`, update the value of `<port>` to 9993.

Example: `jboss-cli.xml`

```
<jboss-cli xmlns="urn:jboss:cli:2.0">
  <default-protocol use-legacy-override="true">https-remoting</default-protocol>
</jboss-cli>
```
The next time you connect to the management interface using the management CLI, you must accept the server certificate and authenticate against the `ManagementRealmHTTPS` security realm:

Example: Accepting Server Certificate and Authenticating

```
$ ./jboss-cli.sh -c
Unable to connect due to unrecognised server certificate
Subject    - CN=appserver,OU=Sales,O=Systems Inc,L=Raleigh,ST=NC,C=US
Issuer     - CN=appserver, OU=Sales, O=Systems Inc, L=Raleigh, ST=NC, C=US
Valid To   - Thu Jun 28 13:38:48 CDT 2018

```

Authenticating against security realm: `ManagementRealmHTTPS`
Username: `httpUser`
Password: [standalone@localhost:9993 ]

**IMPORTANT**

In cases where you have both a `security-realm` and `ssl-context` defined, JBoss EAP will use the SSL/TLS configuration provided by `ssl-context`.

1.2.10. Setting up two-way SSL/TLS for the management interfaces with legacy core management authentication

Two-way SSL/TLS authentication, also known as `client authentication`, authenticates both the client and the server using SSL/TLS certificates. This differs from the Configure the Management Interfaces for One-way SSL/TLS section in that both the client and server each have a certificate. This provides assurance that not only is the server who it says it is, but the client is also who it says it is.

In this section the following conventions are used:

- **HOST1**
  - The JBoss server hostname. For example: `jboss.redhat.com`.

- **HOST2**
  - A suitable name for the client. For example: `myclient`. Note this is not necessarily an actual hostname.

- **CA_HOST1**
  - The DN (distinguished name) to use for the HOST1 certificate. For example: `cn=jboss,dc=redhat,dc=com`. 
CA_HOST2

The DN (distinguished name) to use for the HOST2 certificate. For example: cn=myclient,dc=redhat,dc=com.

NOTE

If a password vault is used to store the keystore and truststore passwords, which is recommended, the password vault should already be created. For more information on the password vault, see the Password Vault section as well as the Password Vault System section of the Red Hat JBoss Enterprise Application Platform 7 Security Architecture guide.

WARNING

Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.

Procedure

1. Generate the keystores.

   ```bash
   $ keytool -genkeypair -alias HOST1_alias -keyalg RSA -keysize 1024 -validity 365 -keystore HOST1.keystore.jks -dname "CA_HOST1" -keypass secret -storepass secret
   $ keytool -genkeypair -alias HOST2_alias -keyalg RSA -keysize 1024 -validity 365 -keystore HOST2.keystore.jks -dname "CA_HOST2" -keypass secret -storepass secret
   ```

2. Export the certificates.

   ```bash
   $ keytool -exportcert -keystore HOST1.keystore.jks -alias HOST1_alias -keypass secret -storepass secret -file HOST1.cer
   $ keytool -exportcert -keystore HOST2.keystore.jks -alias HOST2_alias -keypass secret -storepass secret -file HOST2.cer
   ```

3. Import the certificates into the opposing truststores.

   ```bash
   $ keytool -importcert -keystore HOST1.truststore.jks -storepass secret -alias HOST2_alias -trustcacerts -file HOST2.cer
   $ keytool -importcert -keystore HOST2.truststore.jks -storepass secret -alias HOST1_alias -trustcacerts -file HOST1.cer
   ```


   Define a CertificateRealm in the configuration for the server (host.xml or standalone.xml) and point the interface to it. This can be done using the following commands:

   ```bash
   /core-service=management/security-realm=CertificateRealm:add()
   ```
5. Change the **security-realm** of the **http-interface** to the new CertificateRealm.

   `/core-service=management/security-realm=CertificateRealm/server-identity=ssl:add(keystore-path=/path/to/HOST1.keystore.jks,keystore-password=secret,alias=HOST1_alias)`

   `/core-service=management/security-realm=CertificateRealm/authentication=truststore:add(keystore-path=/path/to/HOST1.truststore.jks,keystore-password=secret)`

6. Add the SSL/TLS configuration for the CLI.

   **IMPORTANT**

   In addition to adding the two-way SSL/TLS, the management interface should also be configured to bind to HTTPS. For details, see **Ensure the Management Interfaces Bind to HTTPS** in the section entitled **Configure the Management Interfaces for One-way SSL/TLS with Legacy Core Management Authentication**.

   Add the SSL/TLS configuration for the CLI, which uses **EAP_HOME/bin/jboss-cli.xml** as a settings file.

   To store the keystore and truststore passwords in plain text, edit **EAP_HOME/bin/jboss-cli.xml** and add the SSL/TLS configuration using the appropriate values for the variables:

   **Example: jboss-cli.xml Storing Keystore and Truststore Passwords in Plain Text**

   ```xml
   <ssl>
     <alias>HOST2_alias</alias>
     <key-store>/path/to/HOST2.keystore.jks</key-store>
     <key-store-password>secret</key-store-password>
     <trust-store>/path/to/HOST2.truststore.jks</trust-store>
     <trust-store-password>secret</trust-store-password>
     <modify-trust-store>true</modify-trust-store>
   </ssl>
   ```

   To use the keystore and truststore passwords stored in a password vault, you need to add the vault configuration and appropriate vault values to **EAP_HOME/bin/jboss-cli.xml**:

   **Example: jboss-cli.xml Storing Keystore and Truststore Passwords in a Password Vault**

   ```xml
   <ssl>
     <vault>
       <vault-option name="KEYSTORE_URL" value="path-to/vault/vault.keystore"/>
       <vault-option name="KEYSTORE_PASSWORD" value="MASK-5WNXs8oEbrs"/>
       <vault-option name="KEYSTORE_ALIAS" value="vault"/>
       <vault-option name="SALT" value="12345678"/>
       <vault-option name="ITERATION_COUNT" value="50"/>
       <vault-option name="ENC_FILE_DIR" value="EAP_HOME/vault"/>
     </vault>
   </ssl>
   ```
IMPORTANT

In cases where you have both a security-realm and ssl-context defined, JBoss EAP will use the SSL/TLS configuration provided by ssl-context.

1.2.11. HTTPS Listener Reference

For a full list of attributes available for the HTTPS listener, see the Undertow Subsystem Attributes section in the JBoss EAP Configuration Guide.

1.2.11.1. About Cipher Suites

You can configure a list of the encryption ciphers which are allowed. For JSSE syntax, it must be a comma-separated list. For OpenSSL syntax, it must be a colon-separated list. Ensure that only one syntax is used. The default is the JVM default.

IMPORTANT

Using weak ciphers is a significant security risk. See NIST Guidelines for NIST recommendations on cipher suites.

See the OpenSSL documentation for a list of available OpenSSL ciphers. Note that the following are not supported:

- @SECLEVEL
- SUITEB128
- SUITEB128ONLY
- SUITEB192

See the Java documentation for a list of the standard JSSE ciphers.

To update the list of enabled cipher suites, use the enabled-cipher-suites attribute of the HTTPS listener in the undertow subsystem.

Example: Management CLI Command for Updating the List of Enabled Cipher Suites

```
/subsystem=undertow/server=default-server/https-listener=https:write-attribute(name=enabled-cipher-suites,value="TLS_RSA_WITH_AES_128_CBC_SHA,TLS_RSA_WITH_AES_256_CBC_SHA")
```
The example only lists two possible ciphers, but real-world examples will likely use more.

1.2.12. Enabling support for the TLS 1.3 protocol with the OpenSSL provider

You can enable support for the TLS 1.3 protocol with the OpenSSL provider for TLS by configuring the `cipher-suite-names` attribute in the `ssl-context` configuration. Choose one of the following methods for configuring JBoss EAP to use the OpenSSL TLS provider:

- Configure the Elytron subsystem to use the OpenSSL TLS provider by default.
- Configure the `providers` attribute of a `server-ssl-context` component or a `client-ssl-context` component to use the OpenSSL TLS provider.

**IMPORTANT**

Compared with TLS 1.2, you might experience reduced performance when running TLS 1.3 with JDK 11. This can occur when clients make a very large number of TLS 1.3 requests to a server. A system upgrade to a newer JDK version can improve performance. Test your setup with TLS 1.3 for performance degradation before enabling it in production.

**Prerequisites**

- Enable one-way SSL/TLS or two-way SSL/TLS for applications.

**Procedure**

1. Choose one of the following methods to configure your JBoss EAP 7.4 instance to use the OpenSSL TLS provider:

   a. Configure the `elytron` subsystem to use the OpenSSL TLS provider by default. To do this, remove the default `final-providers` configuration, which registers the OpenSSL TLS provider after all globally registered providers.

      ```
      /subsystem=elytron:undefine-attribute(name=final-providers)
      reload
      ```

      Next, register the OpenSSL TLS provider ahead of all globally registered providers.

      ```
      /subsystem=elytron:write-attribute(name=initial-providers, value=combined-providers)
      ```

   b. Configure the `providers` attribute of a `server-ssl-context` or a `client-ssl-context` to use the OpenSSL TLS provider.

   **Example of setting the `providers` attribute for an existing `server-ssl-context` called `serverSSC`**.

      ```
      /subsystem=elytron/server-ssl-context=serverSSC:write-attribute(name=providers, value=openssl)
      reload
      ```

2. **Optional:** If you configured your `ssl-context` to use a protocol other than the TLS 1.3 protocol, you must configure the `protocols` attribute in the `ssl-context` to include the TLS 1.3 protocol:
3. Enable support for the TLS 1.3 protocol with the OpenSSL provider by configuring the `cipher-suite-names` attribute in the `ssl-context` configuration. The following example sets `TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1305_SHA256:TLS_AES_128_GCM_SHA256` as the value for the `cipher-suite-names` attribute:

```
/subsystem=elytron/server-ssl-context=serverSSC:write-attribute(name=cipher-suite-names,value=TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1305_SHA256:TLS_AES_128_GCM_SHA256)
```

4. Reload your JBoss EAP instance:

```
reload
```

5. Optional: Test that you can successfully establish an SSL-encrypted connection with the server by using the TLS 1.3 protocol and the TLS 1.3 cipher suite. Use a tool, such as `curl`, to check the output of the configuration:

```
curl -v https://<ip_address>[:<ssl_port>]
```

Example output showing `TLS_AES_256_GCM_SHA384` with the TLS 1.3 protocol to secure the SSL connection.

SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384
* ALPN, server accepted to use h2
* Server certificate:
  * subject: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost
  * start date: Oct 6 14:58:16 2020 GMT
  * expire date: Nov 5 15:58:16 2020 GMT
  * issuer: C=Unknown; ST=Unknown; L=Unknown; O=Unknown; OU=Unknown; CN=localhost
  * SSL certificate verify result: self signed certificate (18), continuing anyway.

Additional resources

- For information about enabling one-way SSL/TLS or two-way SSL/TLS for applications, see Enable One-way SSL/TLS for Applications Using the Elytron Subsystem.
- For information about the `client-ssl-context`, see Using a `client-ssl-context`.
- For information about the `server-ssl-context`, see Using a `server-ssl-context`.

1.2.13. FIPS 140–2 Compliant Cryptography

It is possible to configure FIPS 140–2 compliant cryptography on Red Hat Enterprise Linux using either of the following methods.

- Using the SunPKCS11 provider with an NSS database
- Using the third party BouncyCastle providers
1.2.13.1. Enable FIPS 140-2 Cryptography for SSL/TLS on Red Hat Enterprise Linux 7 and Later

You can configure Undertow to use FIPS 140-2 compliant cryptography for SSL/TLS. The scope of this configuration example is limited to Red Hat Enterprise Linux 7 and later, using the Mozilla NSS library in FIPS mode.

**IMPORTANT**

The installed Red Hat Enterprise Linux must already be configured to be FIPS 140-2 compliant. For more information, see the solution titled *How can I make RHEL 6 or RHEL 7 FIPS 140-2 compliant?*, which is located on the Red Hat Customer Portal.

**WARNING**

Using the TLS 1.2 protocol when running JBoss EAP in FIPS mode can cause a `NoSuchAlgorithmException` to occur. More details on this issue can be found in the solution titled *NoSuchAlgorithmException: no such algorithm: SunTls12MasterSecret*, which is located on the Red Hat Customer Portal.

Therefore, it is not possible to configure HTTP/2 in FIPS mode because HTTP/2 requires the TLS 1.2 protocol. FIPS mode (PKCS11) supports the TLS 1 and the TLS 1.1 protocols so you can use:

- TLS 1.1 in case of Oracle/OpenJDK
- TLS 1 in case of IBM java

To configure Undertow to use FIPS 140-2 compliant cryptography for SSL/TLS, you must do the following:

- **Configure the NSS database.**
- **Configure the management CLI for FIPS 140-2 compliant cryptography for SSL/TLS.**
- **Configure the `undertow` subsystem to use either Elytron or the legacy core management authentication.**

**NOTE**

The OpenSSL provider requires a private key, but it is not possible to retrieve a private key from the PKCS11 store. FIPS does not allow the export of unencrypted keys from FIPS compliant cryptographic module. Therefore, for both the `elytron` subsystem as well as legacy security, it is not possible to use the OpenSSL provider for TLS when in FIPS mode.

**Configuring the NSS database**

1. Create a directory owned by the appropriate user to house the NSS database.
Example Commands for Creating the NSS Database Directory

```bash
$ mkdir -p /usr/share/jboss-as/nssdb
$ chown jboss /usr/share/jboss-as/nssdb
$ modutil -create -dbdir /usr/share/jboss-as/nssdb
```

**NOTE**
- DBM file format, the default database format in RHEL 7 and earlier, has been deprecated. **NSS now uses SQL by default.**
- The `jboss` user is only an example. Replace it with an active user on your operating system to run JBoss EAP.

2. Create the NSS configuration file: `/usr/share/jboss-as/nss_pkcsll_fips.cfg`. It must specify:
   - a name
   - the directory where the NSS library is located
   - the directory where the NSS database was created in the previous step

   **Example:** `nss_pkcsll_fips.cfg`

   ```
   name = nss-fips
   nssLibraryDirectory=/usr/lib64
   nssSecmodDirectory=/usr/share/jboss-as/nssdb
   nssDbMode = readOnly
   nssModule = fips
   ```

   **NOTE**
   If you are not running a 64-bit version of Red Hat Enterprise Linux 6 then set `nssLibraryDirectory` to `/usr/lib` instead of `/usr/lib64`.

3. Edit the Java security configuration file. This configuration file affects the entire JVM, and can be defined using either of the following methods.

   - A default configuration file, `java.security`, is provided in your JDK. This file is used if no other security configuration files are specified. See the JDK vendor’s documentation for the location of this file.

   - Define a custom Java security configuration file and reference it by using the `-Djava.security.properties=/path/to/java.security.properties`. When referenced in this manner it overrides the settings in the default security file. This option is useful when having multiple JVMs running on the same host that require different security settings. Add the following line to your Java security configuration file:

   **Example:** `java.security`

   ```
   security.provider.1=sun.security.pkcs11.SunPKCS11 /usr/share/jboss-as/nss_pkcsll_fips.cfg
   ```
The **nss_pkcs11_fips.cfg** configuration file specified in the above line is the file created in the previous step.

You also need to update the following link in your configuration file from:

```
security.provider.5=com.sun.net.ssl.internal.ssl.Provider
```

to

```
security.provider.5=com.sun.net.ssl.internal.ssl.Provider SunPKCS11-nss-fips
```

**IMPORTANT**

Any other `security.provider.X` lines in this file, for example `security.provider.2`, must have the value of their `X` increased by one to ensure that this provider is given priority.

4. Run the `modutil` command on the NSS database directory you created in the previous step to enable FIPS mode.

```
modutil -fips true -dbdir /usr/share/jboss-as/nssdb
```

**NOTE**

You may get a security library error at this point requiring you to regenerate the library signatures for some of the NSS shared objects.

5. Set the password on the FIPS token.

The name of the token *must* be **NSS FIPS 140-2 Certificate DB**.

```
modutil -changepw "NSS FIPS 140-2 Certificate DB" -dbdir /usr/share/jboss-as/nssdb
```

**IMPORTANT**

The password used for the FIPS token must be a FIPS compliant password. If the password is not strong enough, you may receive an error: **ERROR: Unable to change password on token "NSS FIPS 140-2 Certificate DB"**.

6. Create a certificate using the NSS tools.

**Example Command**

```
$ certutil -S -k rsa -n undertow -t "u,u,u" -x -s "CN=localhost, OU=MYOU, O=MYORG, L=MYCITY, ST=MYSTATE, C=MY" -d /usr/share/jboss-as/nssdb
```

7. Verify that the JVM can read the private key from the PKCS11 keystore by running the following command:
IMPORTANT

Once you have FIPS enabled, you may see the following error when starting JBoss EAP:

```
10:16:13,993 ERROR [org.jboss.msc.service.fail] (MSC service thread 1-1) MSC000001: Failed to start service
jboss.server.controller.management.security_realm.ApplicationRealm.key-manager: org.jboss.msc.service.StartException in service
jboss.server.controller.management.security_realm.ApplicationRealm.key-manager: WFLYDM0018: Unable to start service
at org.jboss.as.domain.management.security.AbstractKeyManagerService.start(AbstractKeyManagerService.java:85)
at org.jboss.msc.service.ServiceControllerImpl$StartTask.startService(ServiceControllerImpl.java:1963)
at org.jboss.msc.service.ServiceControllerImpl$StartTask.run(ServiceControllerImpl.java:1896)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
at java.lang.Thread.run(Thread.java:745)
Caused by: java.security.KeyStoreException: FIPS mode: KeyStore must be from provider SunPKCS11-nss-fips
at sun.security.ssl.KeyManagerFactoryImpl$SunX509.engineInit(KeyManagerFactoryImpl.java:67)
at javax.net.ssl.KeyManagerFactory.init(KeyManagerFactory.java:256)
at org.jboss.as.domain.management.security.AbstractKeyManagerService.createKeyManagers(AbstractKeyManagerService.java:130)
at org.jboss.as.domain.management.security.AbstractKeyManagerService.start(AbstractKeyManagerService.java:83)
... 5 more
```

This message will appear if you have any existing key managers configured, such as the default key manager in legacy core management authentication, that do not use FIPS 140-2 compliant cryptography.

Configure the Management CLI for FIPS 140-2 Compliant Cryptography for SSL/TLS

You must configure the JBoss EAP management CLI to work in an environment with FIPS 140-2 compliant cryptography for SSL/TLS enabled. By default, if you try to use the management CLI in such an environment, the following exception is thrown: `org.jboss.as.cli.CliInitializationException: java.security.KeyManagementException: FIPS mode: only SunJSSE TrustManagers may be used`.

- If you are using the legacy `security` subsystem:

  Update the `javax.net.ssl.keyStore` and `javax.net.ssl.trustStore` system properties in the `jboss-cli.sh` file, as shown below:
If you are using the **elytron** subsystem:

1. Create an XML configuration file for the management CLI with the following contents:

   **Example: cli-wildfly-config.xml**

   ```xml
   <configuration>
   <authentication-client xmlns="urn:elytron:client:1.2">
   <key-stores>
   <key-store name="truststore" type="PKCS11">
   <key-store-clear-password password="P@ssword123"/>
   </key-store>
   </key-stores>
   <ssl-contexts>
   <ssl-context name="client-cli-context">
   <trust-store key-store-name="truststore"/>
   <cipher-suite selector="${cipher.suite.filter}"/>
   <protocol names="TLSv1.1"/>
   </ssl-context>
   </ssl-contexts>
   <ssl-context-rules>
   <rule use-ssl-context="client-cli-context"/>
   </ssl-context-rules>
   </authentication-client>
   </configuration>
   
   **NOTE**

   If you are using the IBM JDK, see the IBM management CLI configuration example for the specific configuration required.

2. When starting the management CLI, pass the configuration file to the management CLI script using the `-Dwildfly.config.url` property. For example:

   ```
   $ jboss-cli.sh -Dwildfly.config.url=cli-wildfly-config.xml
   ```

Configure the Elytron and Undertow Subsystems

1. Add the FIPS 140-2 compliant cryptography **key-store**, **key-manager** and **ssl-context**.

   ```
   /subsystem=elytron/key-store=fipsKS:add(type=PKCS11,provider-name="SunPKCS11-nss-fips",credential-reference={clear-text="P@ssword123"})
   
   /subsystem=elytron/key-manager=fipsKM:add(key-store=fipsKS,algorithm="SunX509",provider-name=SunPKCS11-nss-fips,credential-reference={clear-text="P@ssword123"})
   
   /subsystem=elytron/server-ssl-context=fipsSSC:add(key-manager=fipsKM,protocols=["TLSv1.1"])```

```
2. Update the `undertow` subsystem to use the new `ssl-context`.

   **NOTE**

   `https-listener` must always have either a `security-realm` or `ssl-context` configured. When changing between the two configurations, the commands must be executed as a single batch, as shown below.

   ```
   batch
   /subsystem=undertow/server=default-server/https-listener=https:undefined-
   attribute(name=security-realm)
   /subsystem=undertow/server=default-server/https-listener=https:write-
   attribute(name=ssl-context,value=fipsSSC)
   run-batch
   reload
   ```

   In the `elytron` subsystem, OpenJDK and Oracle JDK in FIPS mode restrict the usage of any advanced features that are based on providing custom `KeyManager` or `TrustManager` implementations. The following configuration attributes do not work on the server:

   - `server-ssl-context.security-domain`
   - `trust-manager.certificate-revocation-list`

   **Configure Undertow with the Legacy Core Management Authentication**

   Optionally, you can still use the legacy core management authentication instead of the `elytron` subsystem to complete the setup of FIPS 140-2 compliant cryptography for SSL/TLS:

   1. Configure Undertow to use SSL/TLS.

      **NOTE**

      The following commands below must either be run in batch mode, or the server must be reloaded after adding the `ssl` server identity. The example below is shown using batch mode.

      ```
      batch
      /core-service=management/security-realm=HTTPSRealm:add
      /core-service=management/security-realm=HTTPSRealm/server-identity=ssl:add(keystore-
      provider=PKCS11, keystore-password="strongP@ssword1")
      /subsystem=undertow/server=default-server/https-listener=https:add(socket-binding=https,
      security-realm=HTTPSRealm, enabled-protocols="TLSv1.1")
      run-batch
      ```

      The basic details for configuring Undertow to SSL/TLS are covered in Setting up an SSL/TLS for Applications.

   2. Configure the cipher suites used by Undertow.
Once you have SSL/TLS configured, you need to configure the https listener and security realm to have a specific set of cipher suites enabled:

**Required Cipher Suites**

- SSL_RSA_WITH_3DES_EDE_CBC_SHA, SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA,
- TLS_RSA_WITH_AES_128_CBC_SHA, TLS_DHE_DSS_WITH_AES_128_CBC_SHA,
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA, TLS_RSA_WITH_AES_256_CBC_SHA,
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA,
- TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA,
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA,
- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA,
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA,
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA,
- TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA,
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA,
- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA,
- TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA,
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA,
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA,
- TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA,
- TLS_ECDH_anon_WITH_AES_128_CBC_SHA,
- TLS_ECDH_anon_WITH_AES_256_CBC_SHA

The basics behind enabling cipher suites for the https listener are covered in About Cipher Suites. To enable cipher suites on the https listener:

**Example Command for Enabling Cipher Suites on the Https Listener**

```
/subsystem=undertow/server=default-server/https-listener=https:write-
attribute(name=enabled-cipher-
suites,value="SSL_RSA_WITH_3DES_EDE_CBC_SHA,SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA,
TLS_RSA_WITH_AES_128_CBC_SHA,TLS_DHE_DSS_WITH_AES_128_CBC_SHA,
TLS_DHE_RSA_WITH_AES_128_CBC_SHA,TLS_RSA_WITH_AES_256_CBC_SHA,
TLS_DHE_RSA_WITH_AES_256_CBC_SHA,
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA,
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA,
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA,
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA,
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA,
TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA,
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA,
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA,
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA,
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA,
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA,
TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA,
TLS_ECDH_anon_WITH_AES_128_CBC_SHA,
TLS_ECDH_anon_WITH_AES_256_CBC_SHA")
```

3. Enable cipher suites on the security realm.

**Example Command for Enabling Cipher Suites on the Security Realm**

```
/core-service=management/security-realm=HTTPSRealm/server-identity=ssl/write-
attribute(name=enabled-cipher-suites, value=[SSL_RSA_WITH_3DES_EDE_CBC_SHA,
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA, TLS_RSA_WITH_AES_128_CBC_SHA,
TLS_DHE_DSS_WITH_AES_128_CBC_SHA])
```
1.2.13.2. Enable FIPS 140-2 Cryptography for SSL/TLS Using Bouncy Castle

You can configure Undertow to use FIPS 140-2 compliant cryptography for SSL/TLS. The scope of this configuration example is limited to Red Hat Enterprise Linux 7 and later. The Bouncy Castle JARs are not provided by Red Hat, and must be obtained directly from Bouncy Castle.

Prerequisites

- Ensure your environment is configured to use the BouncyCastle provider.
- A Bouncy Castle keystore must exist on the server. If one does not exist, it can be created using the following command.

  ```
  $ keytool -genkeypair -alias ALIAS -keyalg RSA -keysize 2048 -keypass PASSWORD -keystore KEYSTORE -storetype BCFKS -storepass STORE_PASSWORD
  ```

Configure the Management CLI for FIPS 140-2 Compliant Cryptography for SSL/TLS Using Elytron

You must configure the JBoss EAP management CLI to work in an environment with FIPS 140-2 compliant cryptography for SSL/TLS enabled.

1. Create an XML configuration file for the management CLI with the following contents:

   Example: cli-wildfly-config.xml

   ```xml
   <configuration>
   <authentication-client xmlns="urn:elytron:client:1.2">
   <key-stores>
   <key-store name="truststore" type="BCFKS">
   <file name="${truststore.location}" />
   <key-store-clear-password password="${password}" />
   </key-store>
   <key-store name="keystore" type="BCFKS">
   <file name="${keystore.location}" />
   <key-store-clear-password password="${password}" />
   </key-store>
   </key-stores>
   </authentication-client>
   ```
When starting the management CLI, pass the configuration file to the management CLI script using the `-Dwildfly.config.url` property. For example:

```
$ jboss-cli.sh -Dwildfly.config.url=cli-wildfly-config.xml
```

Configure the Elytron and Undertow Subsystems

1. Add the FIPS 140-2 compliant cryptography `key-store`, `key-manager` and `ssl-context`. When defining the keystore, the type must be `BCFKS`.

```
<key-stores>
<ssl-contexts>
  <ssl-context name="client-cli-context">
    <key-store-ssl-certificate algorithm="PKIX" key-store-name="keystore">
      <key-store-clear-password password="${password}" />
    </key-store-ssl-certificate>
    <trust-store key-store-name="truststore"/>
    <trust-manager algorithm="PKIX"/>
    <cipher-suite selector="TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA,TLS_DHE_DSS_WITH_AES_128_CBC_SHA,TLS_DHE_DSS_WITH_AES_256_CBC_SHA,TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA,TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA,TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256,TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA,TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA256,TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384,TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA,TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA,TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256,TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA,TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA256,TLS_RSA_WITH_AES_128_CBC_SHA,TLS_RSA_WITH_AES_128_CBC_SHA256,TLS_RSA_WITH_AES_256_CBC_SHA,TLS_RSA_WITH_AES_256_CBC_SHA256,TLS_RSA_WITH_AES_256_CCM,TLS_RSA_WITH_AES_128_CCM,TLSSubsystem=elytron/key-store=fipsKS:add(path=KEYSTORE,relative-to=jboss.server.config.dir,credential-reference={clear-text=STORE_PASSWORD},type="BCFKS")
    </cipher-suite>
  </ssl-context>
</ssl-contexts>
<ssl-context-rules>
  <rule use-ssl-context="client-cli-context"/>
</ssl-context-rules>
</authentication-client>
</configuration>
```

2. When starting the management CLI, pass the configuration file to the management CLI script using the `-Dwildfly.config.url` property. For example:

```
$ jboss-cli.sh -Dwildfly.config.url=cli-wildfly-config.xml
```

Configure the Elytron and Undertow Subsystems
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA,
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384,
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA,
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA,
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256,
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA,
TLS_RSA_WITH_3DES_EDE_CBC_SHA,
TLS_RSA_WITH_AES_128_CBC_SHA,
TLS_RSA_WITH_AES_128_CBC_SHA256,
TLS_RSA_WITH_AES_256_CBC_SHA,
TLS_RSA_WITH_AES_256_CBC_SHA256,
TLS_RSA_WITH_AES_128_CCM

2. Update the **undertow** subsystem to use the new **ssl-context**.

**NOTE**

**https-listener** must always have either a **security-realm** or **ssl-context** configured. When changing between the two configurations, the commands must be executed as a single batch, as shown below.

```bash
batch
/subsystem=undertow/server=default-server/https-listener=https:undefined-attribute(name=security-realm)
/subsystem=undertow/server=default-server/https-listener=https:write-attribute(name=ssl-context,value=fipsSSC)
run-batch
reload
```

### 1.2.14. FIPS 140-2 Compliant Cryptography on IBM JDK

On the IBM JDK, the IBM Java Cryptographic Extension (JCE) IBMJCEFIPS provider and the IBM Java Secure Sockets Extension (JSSE) FIPS 140-2 Cryptographic Module (IBMJSSE2) for multi-platforms provide FIPS 140-2 compliant cryptography.

For more information on the IBMJCEFIPS provider, see the [IBM Documentation for IBM JCEFIPS](https://www.ibm.com/support/pages/node/721779) and [NIST IBMJCEFIPS – Security Policy](https://csrc.nist.gov/projects/fips-140-2). For more information on IBMJSSE2, see the [Running IBMJSSE2 in FIPS mode](https://www-01.ibm.com/support/docview.wss?uid=swg21520488).

#### 1.2.14.1. Key Storage

The IBM JCE does not provide a keystore. The keys are stored on the computer and do not leave its physical boundary. If the keys are moved between computers they must be encrypted.

To run **keytool** in FIPS-compliant mode use the `-providerClass` option on each command like this:

```bash
keytool -list -storetype JCEKS -keystore mystore.jck -storepass mystorepass -providerClass com.ibm.crypto.fips.provider.IBMJCEFIPS
```

#### 1.2.14.2. Management CLI Configuration

To **configure the management CLI for FIPS 140-2 compliant cryptography** on the IBM JDK, you must use a management CLI configuration file specifically for the IBM JDK, such as the following:
Example: cli-wildfly-config-ibm.xml

```xml
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <key-stores>
      <key-store name="truststore" type="JKS">
        <file name="/path/to/truststore"/>
        <key-store-clear-password password="P@ssword123"/>
      </key-store>
    </key-stores>
    <ssl-contexts>
      <ssl-context name="client-cli-context">
        <trust-store key-store-name="truststore"/>
        <cipher-suite selector="${cipher.suite.filter}"/>
        <protocol names="TLSv1"/>
      </ssl-context>
    </ssl-contexts>
    <ssl-context-rules>
      <rule use-ssl-context="client-cli-context"/>
    </ssl-context-rules>
  </authentication-client>
</configuration>
```

1.2.14.3. Examine FIPS Provider Information

To examine information about the IBMJCEFIPS used by the server, enable debug-level logging by adding `-Djavax.net.debug=true` to the `standalone.conf` or `domain.conf` files. Information about the FIPS provider is logged to the `server.log` file, for example:

```
04:22:45,689 INFO [stdout] (http-/127.0.0.1:8443-1) DHCrypt: DH KeyPairGenerator from provider from init IBMJCEFIPS version 1.7
04:22:45,754 INFO [stdout] (http-/127.0.0.1:8443-1) JsseJCE: Using KeyAgreement DiffieHellman from provider IBMJCEFIPS version 1.7
04:22:45,754 INFO [stdout] (http-/127.0.0.1:8443-1) DHCrypt: DH KeyAgreement from provider IBMJCEFIPS version 1.7
04:22:45,754 INFO [stdout] (http-/127.0.0.1:8443-1) DHCrypt: DH KeyAgreement from provider from initIBMJCEFIPS version 1.7
```

1.2.15. Starting a Managed Domain when the JVM is Running in FIPS Mode

Update each host controller and the master domain controller to use SSL/TLS for communication.

Prerequisites

Before you begin, make sure you have completed the following prerequisites.

- You have implemented a managed domain.
  For details about configuring a managed domain, see the Domain Management section in the JBoss EAP Configuration Guide.
- You have configured FIPS.
For details about configuring FIPS, see Enable FIPS 140-2 Cryptography for SSL/TLS on Red Hat Enterprise Linux 7 and later.

- You have created all necessary certificates and have imported the domain controller’s certificate into each controller’s truststore.

**WARNING**

Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 in all affected packages.

1. On the master domain controller, create an SSL/TLS security realm that is configured to use your NSS database as a PKCS11 provider.

   **Example: Security Realm on the Master Domain Controller**

   ```xml
   <security-realm name="HTTPSRealm">
   <server-identities>
   <ssl>
   <engine enabled-protocols="TLSv1.1"/>
   <keystore provider="PKCS11" keystore-password="strongP@ssword1"/>
   </ssl>
   </server-identities>
   <authentication>
   <local default-user="\$local"/>
   <properties path="https-users.properties" relative-to="jboss.domain.config.dir"/>
   </authentication>
   </security-realm>
   
   </security-realm>
   
   <management-interfaces>
   
   </management-interfaces>
   ```

2. On each host controller, create a security realm with an SSL/TLS truststore for authentication.

   **Example: Security Realm on Each Host Controller**

   ```xml
   <security-realm name="HTTPSRealm">
   <authentication>
   <truststore provider="PKCS11" keystore-password="strongP@ssword1"/>
   </authentication>
   </security-realm>
   ```

   **NOTE**

   Repeat this process on each host.

3. Secure the HTTP interface on the master domain controller with the security realm you just created.

   **Example: HTTP Interface**

   ```xml
   <management-interfaces>
   ```
4. Use the SSL/TLS realm on each host controller to connect to the master domain controller.
   Update the security realm used for connecting to the master domain controller. Modify the host
   controller’s configuration file, for example `host.xml` or `host-slave.xml`, while the server is not
   running.

   **Example: Host Controller Configuration File**

   ```xml
   <domain-controller>
   <remote security-realm="HTTPSRealm">
   <discovery-options>
     <static-discovery name="primary" protocol="${jboss.domain.master.protocol:remote}"
     host="${jboss.domain.master.address}" port="${jboss.domain.master.port:9990}"/>
   </discovery-options>
   </remote>
   </domain-controller>
   ```

5. Update how each server connects back to its host controller.

   **Example: Server Configuration**

   ```xml
   <server name="my-server" group="my-server-group">
   <ssl ssl-protocol="TLS" trust-manager-algorithm="PKIX" truststore-type="PKCS11"
   truststore-password="strongP@ssword1"/>
   </server>
   ```

6. Configure two-way SSL/TLS in a managed domain.
   To enable two-way SSL/TLS, add a truststore authentication method to the SSL/TLS security
   realm for the master domain controller, execute the following management CLI commands:

   ```
   /host=master/core-service=management/security-realm=HTTPSRealm/authentication=truststore:add(keystore-provider="PKCS11",keystore-password="strongP@ssword1")
   reload --host=master
   ```

   You also need to update each host controller’s security realm to have an SSL server identity,
   execute the following management CLI commands:

   ```
   /host=host1/core-service=management/security-realm=HTTPSRealm/server-identity=ssl:add(keystore-provider=PKCS11, keystore-password="strongP@ssword1",enabled-protocols="[TLSv1.1]")
   reload --host=host1
   ```
You also need to ensure that each host’s certificate is imported into the domain controller’s truststore.

1.2.16. Secure the Management Console with Red Hat Single Sign-On

You can secure the JBoss EAP management console with Red Hat Single Sign-On using the elytron subsystem.

**NOTE**

This feature is only available when running a standalone server and is not supported when running a managed domain. It is not supported to use Red Hat Single Sign-On to secure the management CLI.

Use the following steps to set up Red Hat Single Sign-On to authenticate users for the JBoss EAP management console.

2. Install the Red Hat Single Sign-On client adapter on JBoss EAP.

**Configure a Red Hat Single Sign-On Server for JBoss EAP Management**

2. Start the Red Hat Single Sign-On server. This procedure assumes that you started the server with a port offset of 100.
   
   `$ RHSSO_HOME/bin/standalone.sh -Djboss.socket.binding.port-offset=100`

3. Log in to the Red Hat Single Sign-On administration console at `http://localhost:8180/auth/`. If this is the first time you have accessed the Red Hat Single Sign-On administration console, you are prompted to create an initial administration user.
4. Create a new realm called `wildfly-infra`.
   a. From the drop down next to the realm name, click Add realm, enter `wildfly-infra` in the Name field, and click Create.
5. Create a client application called `wildfly-console`.
   
   **IMPORTANT**
   
   The name of this client application must be `wildfly-console`.
   a. Select Clients and click Create.
   b. Enter `wildfly-console` in the Client ID field and click Save.
c. In the **Settings** screen that appears, set **Access Type** to **public**, Valid Redirect URIs to `http://localhost:9990/console/*`, Web Origins to `http://localhost:9990`, and click **Save**.

6. Create a client application called **wildfly-management**.
   a. Select **Clients** and click **Create**.
   b. Enter **wildfly-management** in the **Client ID** field and click **Save**.
   c. In the **Settings** screen that appears, set **Access Type** to **bearer-only** and click **Save**.

7. Create a role to grant access to the JBoss EAP management console.
   a. Select **Roles** and click **Add Role**.
   b. Enter **ADMINISTRATOR** in uppercase in the **Role Name** field and click **Save**. This procedure uses the **ADMINISTRATOR** role, but other roles are supported. For more information, see the **Role-Based Access Control** section of JBoss EAP’s **Security Architecture**.

8. Create a user and assign the **ADMINISTRATOR** role to them.
   a. Select **Users** and click **Add user**.
   b. Enter **jboss** in the **Username** field and click **Save**.
   c. Select the **Credentials** tab and set a password for this user.
   d. Select the **Role Mappings** tab, select **ADMINISTRATOR** and click **Add selected** to add the role to this user.

**Install the Red Hat Single Sign-On Client Adapter on JBoss EAP**

1. Download the Red Hat Single Sign-On client adapter for JBoss EAP 7 from the software downloads page.

2. Unzip this file into the root directory of your JBoss EAP installation.

3. Execute the **adapter-elytron-install-offline.cli** script to configure your JBoss EAP installation.

   ```
   $ EAP_HOME/bin/jboss-cli.sh --file=adapter-elytron-install-offline.cli
   ```

**IMPORTANT**

This script adds the **keycloak** subsystem and other required resources in the **elytron** and **undertow** subsystems to **standalone.xml**. If you need to use a different configuration file, modify the script as needed.

**Configure JBoss EAP to Use Red Hat Single Sign-On**

1. In the **EAP_HOME/bin/** directory, create a file called **protect-eap-mgmt-services.cli** with the following contents.

   ```
   # Create a realm for both JBoss EAP console and mgmt interface
   ```
2. In this file, replace `REALM_PUBLIC_KEY` with the value of the public key. To obtain this value, log in to the Red Hat Single Sign-On administration console, select the `wildfly-infra` realm, navigate to `Realm Settings → Keys` and click `Public key`.

3. Start JBoss EAP.

   $ EAP_HOME/bin/standalone.sh

   **IMPORTANT**

   If you modified the `adapter-elytron-install-offline.cli` script when installing the Red Hat Single Sign-On client adapter to use a configuration file other than `standalone.xml`, be sure to start the JBoss EAP using that configuration.

4. Execute the `protect-eap-mgmt-services.cli` script.

   $ EAP_HOME/bin/jboss-cli.sh --connect --file=protect-eap-mgmt-services.cli

Now, when you access the JBoss EAP management console at `http://localhost:9990/console/`, you are redirected to Red Hat Single Sign-On to log in, and then redirected back to the JBoss EAP management console upon successful authentication.
1.3. CONFIGURING SECURITY AUDITING FOR A LEGACY SECURITY DOMAIN

You can use an audit module to monitor the events in the security subsystem. Auditing uses provider modules, custom implementations, or both to monitor events. After monitoring events, the audit module writes a log file, reads email notifications, or uses other measurable auditing mechanisms.

Use the management console to configure security auditing settings for a security domain.

Procedure

1. Click on the Configuration tab.
3. Select an editable security domain and click View.
4. Select the Audit tab and press Add to add a new audit module.
5. Set a name for the module and fill in the Code field with the class name of the provider module.
6. Optional: Add module options by editing the module and adding key/value pairs in the Module Options field. Press Enter to add a new value and press Backspace to remove an existing value.

1.4. SECURITY AUDITING WITH ELYTRON

You can use Elytron to complete security audits on triggering events. Security auditing refers to triggering events, such as writing to a log, in response to an authorization or authentication attempt.

The type of security audit performed on events depends on your security realm configuration.

1.4.1. Elytron audit logging

After you enable audit logging with the elytron subsystem, you can log Elytron authentication and authorization events within the application server. Elytron stores audit log entries in either JSON for storing individual events or SIMPLE for human readable text format.

Elytron audit logging differs from other types of audit logging, such as audit logging for the JBoss EAP management interfaces.

Elytron disables audit logging by default. You can enable audit logging by configuring any of the following log handlers for Elytron. You can add a log handler to a security domain.

- File audit logging
- Periodic rotating file audit logging
- Size rotating file audit logging
- syslog audit logging
- Custom audit logging
You can use the `aggregate-security-event-listener resource` to send security events to more destinations, such as loggers. The `aggregate-security-event-listener resource` delivers all events to all listeners specified in the aggregate listener definition.

You can use an audit module to monitor events for a legacy security domain. You can use the management console to configure security auditing settings for a legacy security domain.

### Additional resources

- For information about configuring auditing with the legacy security system, see [Configuring security auditing for a legacy security domain](#).
- For more information about management interface audit logging options, see [Management audit logging](#) in the Configuration Guide.
- For more information about file audit logging, see [Enabling file audit logging](#).
- For more information about periodic rotating file audit logging, see [Periodic Rotating File Audit Logging](#).
- For more information about size rotating file audit logging, see [Size rotating file audit logging](#).
- For more information about `syslog` audit logging, see [syslog audit logging](#).
- For more information about custom audit logging, see [Using custom security event listeners in Elytron](#).

### 1.4.2. Enabling file audit logging

You can use the `elytron` subsystem to enable file audit logging for your standalone server or a server in a managed domain.

File audit logging stores audit log messages in a single file within your file system. By default, Elytron specifies `local-audit` as the file audit logger. You must enable `local-audit` so that it can write Elytron audit logs to `EAP_HOME/standalone/log/audit.log` on a standalone server or `EAP_HOME/domain/log/audit.log` for a managed domain.

**Procedure**

1. Create a file audit log.

   **Example of creating a file audit log by using the `elytron` subsystem:**

   ```
   /subsystem=elytron/file-audit-log=<audit_log_name>:add(path="<path_to_log_file>", relative-to="<base_for_path_to_log_file>", format=<format_type>, synchronized=<whether_to_log_immediately>)
   ```

2. Add the file audit log to a security domain.

   **Example command adding file audit log to a security domain**

   ```
   /subsystem=elytron/security-domain=<security_domain_name>:write-attribute(name=security-event-listener , value=<audit_log_name>)
   ```
Additional resources

- For more information about file audit logger attributes, see File audit logger attributes.

1.4.3. Enabling periodic rotating file audit logging

You can use the elytron subsystem to enable periodic rotating file audit logging for your standalone server or a server in a domain domain.

Periodic rotating file audit logging automatically rotates audit log files based on your configured schedule. Periodic rotating file audit logging is similar to default file audit logger, but periodic rotating file audit logging contains an additional attribute: suffix.

The value of the suffix attribute is a date specified using the java.time.format.DateTimeFormatter format, such as yyyy-MM-dd. Elytron automatically calculates the period of the rotation from the value provided with the suffix. The elytron subsystem appends the suffix to the end of a log file name.

Procedure

1. Create a periodic rotating file audit log.

   Example of creating periodic rotating file audit log in the elytron subsystem

   ```
   /subsystem=elytron/periodic-rotating-file-audit-log=<periodic_audit_log_name>:add(path="<periodic_audit_log_filename>", relative-to="<path_to_audit_log_directory>", format="<record_format>", synchronized="<whether_to_log_immediately>", suffix="<suffix_in_DateTimeFormatter_format>")
   ```

2. Add the periodic rotating file audit logger to a security domain.

   Example adding a periodic rotating file audit logger to a security domain

   ```
   /subsystem=elytron/security-domain=<security_domain_name>:write-attribute(name=security-event-listener, value=<periodic_audit_log_name>)
   ```

Additional resources

- For information about periodic rotating file audit logger attributes, see the periodic-rotating-file-audit-log Attributes table.

1.4.4. Enabling size rotating file audit logging

You can use the elytron subsystem to enable size rotating file audit logging for your standalone server or a server in a domain managed.

Size rotating file audit logging automatically rotates audit log files when the log file reaches a configured file size. Size rotating file audit logging is similar to default file audit logger, but the size rotating file audit logging contains additional attributes.

When the log file size exceeds the limit defined by the rotate-size attribute, Elytron appends the suffix .1 to the end of the current file, and Elytron creates a new log file. Elytron increments a suffix by one for existing log files. For example, Elytron renames audit_log.1 to audit_log.2. Elytron continues the...
increments until a log file amount reaches the maximum number of log files, defined by `max-backup-index`. When a log files exceed the `max-backup-index` value, Elytron removes the file, for example `audit_log.99`, that is the file that is over limit.

**Procedure**

1. Create a size rotating file audit log.

   **Example of creating a size rotating file audit log by using the elytron subsystem:**
   ```xml
   /subsystem=elytron/size-rotating-file-audit-log=<audit_log_name>:add(path="<path_to_log_file>",relative-to="<base_for_path_to_log_file>",format="<record_format>",synchronized="<whether_to_log_immediately>",rotate-size="<max_file_size_before_rotation>",max-backup-index="<max_number_of_backup_files>"
   ```

2. Add the size rotating audit logger to a security domain.

   **Example of enabling a size rotating file audit log by using the elytron subsystem:**
   ```xml
   /subsystem=elytron/security-domain=<domain_size_logger>:write-attribute(name=security-event-listener, value=<audit_log_name>)
   ```

**Additional resources**

- For information about size rotating file audit logging attributes, see the [Size rotating file audit logging attributes](#) table.

### 1.4.5. Enabling syslog audit logging

You can use the `elytron` subsystem to enable `syslog` audit logging for your standalone server or a server in a domain managed. When you use `syslog` audit logging, you send the logging results to a `syslog` server, which provides more security options than logging to a local file.

The `syslog` handler specify parameters used to connect to a `syslog` server, such as the `syslog` server’s host name and the port on which the `syslog` server listens. You can define multiple `syslog` handlers and activate them simultaneously.

Supported log formats include `RFC5424` and `RFC3164`. Supported transmission protocols include UDP, TCP, and TCP with SSL.

When you define a `syslog` for the first instance, the logger sends an `INFORMATIONAL` priority event to `syslog` server containing the message as demonstrated in the following example:

```
"Elytron audit logging enabled with RFC format: <format>"
```

`<format>` refers to the RFC format configured for the audit logging handler, which defaults to `RFC5424` value.

**Procedure**

1. Add a `syslog` handler.
Example of adding a syslog handler by using the elytron subsystem:

```
/subsystem=elytron/syslog-audit-log=<syslog_audit_log_name>:add(host-name=<record_host_name>, port=<syslog_server_port_number>, server-address=<syslog_server_address>, format=<record_format>, transport=<transport_layer_protocol>)
```

You can also send logs to a syslog server over TLS:

Example syslog configuration to send logs over TLS

```
/subsystem=elytron/syslog-audit-log=<syslog_audit_log_name>:add(transport=SSL_TCP, server-address=<syslog_server_address>, port=<syslog_server_port_number>, host-name=<record_host_name>, ssl-context=<client_ssl_context>)
```

2. Add the syslog audit logger to a security domain.

Example of adding a syslog audit logger to a security domain

```
/subsystem=elytron/security-domain=<security_domain_name>:write-attribute(name=security-event-listener, value=<syslog_audit_log_name>)
```

Additional resources

- For information about syslog-audit-log attributes, see the syslog-audit-log Attributes table.
- For more information enabling support for TLS by setting the ssl-context configuration, see Using a client-ssl-context.
- For more information about RFC5424, see The Syslog Protocol.
- For more information about RFC3164, see The BSD syslog Protocol.

1.4.6. Using custom security event listeners in Elytron

You can use Elytron to define a custom event listener. A custom event listener manages the processing incoming security events. You can use the event listener for custom audit logging purposes, or you can use the event listener to authenticate users against your internal identity storage.
IMPORTANT

Using the `module` management CLI command to add and remove modules is provided as a Technology Preview feature only. The `module` command is not appropriate for use in a managed domain or when connecting with a remote management CLI. You must manually add or remove modules in a production environment.

Technology Preview features are not supported with Red Hat production service level agreements (SLAs), might not be functionally complete, and Red Hat does not recommend to use them for production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

See Technology Preview Features Support Scope on the Red Hat Customer Portal for information about the support scope for Technology Preview features.

Procedure

1. Create a class that implements the `java.util.function.Consumer<org.wildfly.security.auth.server.event.SecurityEvent>` interface. For example, the following prints a message whenever a user succeeds or fails authentication.

Example of creating a Java class that uses the specified interface:

```java
public class MySecurityEventListener implements Consumer<SecurityEvent> {
    public void accept(SecurityEvent securityEvent) {
        if (securityEvent instanceof SecurityAuthenticationSuccessfulEvent) {
            System.err.printf("Authenticated user %s\n", securityEvent.getSecurityIdentity().getPrincipal());
        } else if (securityEvent instanceof SecurityAuthenticationFailedEvent) {
            System.err.printf("Failed authentication as user %s\n", ((SecurityAuthenticationFailedEvent)securityEvent).getPrincipal());
        }
    }
}
```

The Java class in the example prints a message whenever a user succeeds or fails authentication.

2. Add the JAR that provides the custom event listener as a module to JBoss EAP, The following is an example of the management CLI command that adds a custom event listener as a module to Elytron.

Example of using the `module` command to add a custom event listener as a module to Elytron:

```
/subsystem=elytron/custom-security-event-listener=<listener_name>:add(module=<module_name>, class-name=<class_name>)
```

3. Reference the custom event listener in the security domain.

Example of referencing a custom event listener in ApplicationDomain:
4. Restart the server.

```sh
$ reload
```

The event listener receives security events from the specified security domain.

**Additional resources**

- For information about manually adding or removing modules in a production environment, see [Create a Custom Module Manually](#) and [Remove a Custom Module Manually](#) in the *Configuration Guide*.

- For information about adding a custom event listener as a module, see [Add a Custom Component to Elytron](#).

**1.5. CONFIGURE ONE-WAY AND TWO-WAY SSL/TLS FOR APPLICATIONS**

### 1.5.1. Automatic Self-signed Certificate Creation for Applications

When using the legacy security realms, JBoss EAP provides automatic generation of self-signed certificate for development purposes.

**Example: Server Log Showing Self-signed Certificate Creation**

```
15:26:09,031 WARN  [org.jboss.as.domain.management.security] (MSC service thread 1-7)
WFLYDM0111: Keystore /path/to/jboss/standalone/configuration/application.keystore not found, it will
be auto generated on first use with a self signed certificate for host localhost
...
15:26:10,076 WARN  [org.jboss.as.domain.management.security] (MSC service thread 1-2)
WFLYDM0113: Generated self signed certificate at /path/to/jboss/configuration/application.keystore.
Please note that self signed certificates are not secure, and should only be used for testing purposes.
Do not use this self signed certificate in production.
SHA-1 fingerprint of the generated key is
SHA-256 fingerprint of the generated key is
```

This certificate is created for testing purposes and is assigned to the HTTPS interface used by applications. The keystore containing the certificate will be generated if the file does not exist the first time the HTTPS interface is accessed.

**Example: Default ApplicationRealm Using the Self-signed Certificate**

```xml
<security-realm name="ApplicationRealm">
  <server-identities>
    <ssl>
      <keystore path="application.keystore" relative-to="/jboss.server.config.dir" keystore-password="password" alias="server" key-password="password" generate-self-signed-certificate-```
Example: Default HTTPS Interface Configuration

```xml
<subsystem xmlns="urn:jboss:domain:undertow:10.0">
  ...
  <server name="default-server">
    ...
    <https-listener name="https" socket-binding="https" security-realm="ApplicationRealm" enable-http2="true">
      <host name="default-host" alias="localhost">
        ...
      </host>
    </https-listener>
  </server>
  ...
</subsystem>
```

NOTE

If you want to disable the self-signed certificate creation, you will need to remove the `generate-self-signed-certificate-host="localhost"` from the server keystore configuration. The `generate-self-signed-certificate-host` attribute holds the host name for which the self-signed certificate should be generated.

WARNING

This self-signed certificate is intended for testing purposes only and is not intended for use in production environments. For more information on configuring SSL/TLS for applications with Elytron, see the Enable One-way SSL/TLS for Applications using the Elytron Subsystem and Enable Two-way SSL/TLS for Applications using the Elytron Subsystem sections. For more information on configuring SSL/TLS for applications with legacy security, see the Enable One-way SSL/TLS for Applications Using Legacy Security Realms and Enable Two-way SSL/TLS for Applications Using Legacy Security Realms sections.

1.5.2. Using Elytron

1.5.2.1. Enable One-way SSL/TLS for Applications Using the Elytron Subsystem

In JBoss EAP, you can enable one-way SSL/TLS for the for deployed applications using the JBoss EAP management CLI or the management console.

In the management CLI, one-way SSL/TLS can be enabled in two ways:

- Using `security command`.
- Using `elytron` subsystem commands.

Using a Security Command
The **security enable-ssl-http-server** command can be used to enable one-way SSL/TLS for deployed applications.

### Example: Wizard Usage

```
security enable-ssl-http-server --interactive

Please provide required pieces of information to enable SSL:
Key-store file name (**default** **default**-server.keystore): keystore.jks
Password (blank generated): secret
What is your first and last name? [Unknown]: localhost
What is the name of your organizational unit? [Unknown]:
What is the name of your organization? [Unknown]:
What is the name of your City or Locality? [Unknown]:
What is the name of your State or Province? [Unknown]:
What is the two-letter country code for this unit? [Unknown]:
Is CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correct y/n [y]? Validity (in days, blank default): 365
Alias (blank generated): localhost
Enable SSL Mutual Authentication y/n (blank n): n

SSL options:
key store file: keystore.jks
distinguished name: CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
password: secret
validity: 365
alias: localhost
Server keystore file keystore.jks, certificate file keystore.pem and keystore.csr file will be generated in server configuration directory.
Do you confirm y/n: y
```

**NOTE**

Once the command is executed, the management CLI will reload the server.

One-way SSL/TLS is now enabled for applications.

### Using Elytron Subsystem Commands

In JBoss EAP, you can use the **elytron** subsystem, along with the **undertow** subsystem, to enable one-way SSL/TLS for deployed applications.

1. Configure a **key-store** in JBoss EAP:

```
/subsystem=elytron/key-store=httpsKS:add(path=/path/to/keystore.jks, credential-reference={clear-text=secret}, type=JKS)
```

If the keystore file does not exist yet, the following commands can be used to generate an example key pair:

```
/subsystem=elytron/key-store=httpsKS:generate-key-pair(alias=localhost,algorithm=RSA,key-size=1024,validity=365,credential-reference={clear-text=secret},distinguished-name="CN=localhost")
```
Configure a **key-manager** that references your **key-store**.

```xml
/subsystem=elytron/key-manager=httpsKM:add(key-store=httpsKS,credential-reference={clear-text=secret})
```

**IMPORTANT**

Red Hat did not specify the algorithm attribute in the previous command, because the Elytron subsystem uses `KeyManagerFactory.getDefaultAlgorithm()` to determine an algorithm by default. However, you can specify the algorithm attribute. To specify the algorithm attribute, you need to know what key manager algorithms are provided by the JDK you are using. For example, a JDK that uses SunJSSE provides the **PKIX** and **SunX509** algorithms.

In the previous command you can specify **SunX509** as the key manager algorithm attribute.

Configure a **server-ssl-context** that references your **key-manager**.

```xml
/subsystem=elytron/server-ssl-context=httpsSSC:add(key-manager=httpsKM, protocols=["TLSv1.2"])`)
```

**IMPORTANT**

You need to determine what SSL/TLS protocols you want to support. The example command above uses **TLSv1.2**. You can use the `cipher-suite-filter` argument to specify which cipher suites are allowed, and the `use-cipher-suites-order` argument to honor server cipher suite order. The `use-cipher-suites-order` attribute by default is set to **true**. This differs from the legacy **security** subsystem behavior, which defaults to honoring client cipher suite order.

**WARNING**

Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.

Check and see if the **https-listener** is configured to use a legacy security realm for its SSL configuration.

```xml
/subsystem=undertow/server=default-server/https-listener=https:read-attribute(name=security-realm)
```
The above command shows that the **https-listener** is configured to use the **ApplicationRealm** legacy security realm for its SSL configuration. Undertow cannot reference both a legacy security realm and an **ssl-context** in Elytron at the same time so you must remove the reference to the legacy security realm.

**NOTE**

If the result is **undefined**, you do not need to remove the reference to the security realm in the next step.

5. Remove the reference to the legacy security realm, and update the **https-listener** to use the **ssl-context** from Elytron.

**NOTE**

**https-listener** must always have either a **security-realm** or **ssl-context** configured. When changing between the two configurations, the commands must be executed as a single batch, as shown below.

```bash
batch
/subsystem=undertow/server=default-server/https-listener=https:undefine-attribute(name=security-realm)
/subsystem=undertow/server=default-server/https-listener=https:write-attribute(name=ssl-context, value=httpsSSC)
run-batch
```

6. Reload the server.

```bash
reload
```

One-way SSL/TLS is now enabled for applications.

**NOTE**

You can disable one-way SSL/TLS for deployed applications using the **disable-ssl-http-server** command.

```bash
security disable-ssl-http-server
```

This command does not delete the Elytron resources. It configures the system to use the **ApplicationRealm** legacy security realm for its SSL configuration.

**Using Management Console**

You can enable SSL for applications by configuring the **undertow** subsystem using an SSL wizard in the management console.

To access the wizard:
1. Access the management console. For more information, see the Management Console section in the JBoss EAP Configuration Guide.


3. Click the name of the server to configure.

4. Click View.

5. Navigate to Listener → HTTPS Listener.

6. Select the listener for which SSL is to be enabled, and click Enable SSL to launch the wizard. The wizard guides you through the following scenarios for enabling SSL:
   - You want to create a certificate store and generate a self-signed certificate.
   - You want to obtain a certificate from Let’s Encrypt Certificate Authority.
   - You already have the certificate store on the file system, but no keystore configuration.
   - You already have a keystore configuration that uses a valid certificate store.

Using the wizard, you can optionally create a truststore for mutual authentication.

1.5.2.2. Enable Two-way SSL/TLS for Applications Using the Elytron Subsystem

1. Obtain or generate your client keystores:

   $ keytool -genkeypair -alias client -keyalg RSA -keysize 1024 -validity 365 -keystore client.keystore.jks -dname "CN=client" -keypass secret -storepass secret

2. Export the client certificate:

   keytool -exportcert -keystore client.keystore.jks -alias client -keypass secret -storepass secret -file /path/to/client.cer

3. Enable two-way SSL/TLS for deployed applications.
   In JBoss EAP, two-way SSL/TLS for deployed applications can be enabled either by using a security command or by using the elytron subsystem commands.

   a. Using a security command.

      The security enable-ssl-http-server command can be used to enable two-way SSL/TLS for the deployed applications.

      **NOTE**

      The following example does not validate the certificate as no chain of trust exists. If you are using a trusted certificate, then the client certificate can be validated without issue.

      **Example: Wizard Usage**

      security enable-ssl-http-server --interactive
NOTE

Once the command is executed, the management CLI will reload the server.

To complete the two-way SSL/TLS authentication, you need to import the server certificate into the client truststore and configure your client to present the client certificate.

b. Using elytron subsystem commands.

In JBoss EAP, you can also use the elytron subsystem, along with the undertow subsystem, to enable two-way SSL/TLS for deployed applications.

i. Obtain or generate your keystore.

Before enabling two-way SSL/TLS in JBoss EAP, you must obtain or generate the keystores, truststores and certificates you plan on using.

A. Create a server keystore:

```bash
/subsystem=elytron/key-store=twoWayKS:add(path=/PATH/TO/server.keystore.jks,credential-reference=
```
(clear-text=secret),type=JKS)

/subsystem=elytron/key-store=twoWayKS:generate-key-pair(alias=localhost,algorithm=RSA,key-size=1024,validity=365,credential-reference={clear-text=secret},distinguished-name="CN=localhost")

/subsystem=elytron/key-store=twoWayKS:store()

NOTE

The command above uses an absolute path to the keystore. Alternatively you can use the relative-to attribute to specify the base directory variable and path specify a relative path.

/subsystem=elytron/key-store=twoWayKS:add(path=server.keystore.jks,relative-to=jboss.server.config.dir,credential-reference={clear-text=secret},type=JKS)

B. Export the server certificate:

/subsystem=elytron/key-store=twoWayKS:export-certificate(alias=localhost,path=/path/to/server.cer,pem=true)

ii. Create a keystore for the server truststore and import the client certificate into the server truststore.

NOTE

The following example does not validate the certificate as no chain of trust exists. If you are using a trusted certificate, then the client certificate can be validated without issue.

/subsystem=elytron/key-store=twoWayTS:add(path=/path/to/server.truststore.jks,credential-reference={clear-text=secret},type=JKS)

/subsystem=elytron/key-store=twoWayTS:import-certificate(alias=client,path=/path/to/client.cer,credential-reference={clear-text=secret},trust-cacerts=true,validate=false)

/subsystem=elytron/key-store=twoWayTS:store()

iii. Configure a key-manager that references your keystore key-store.

/subsystem=elytron/key-manager=twoWayKM:add(key-store=twoWayKS, credential-reference={clear-text=secret})
**IMPORTANT**

Red Hat did not specify the algorithm attribute in the previous command, because the Elytron subsystem uses `KeyManagerFactory.getDefaultAlgorithm()` to determine an algorithm by default. However, you can specify the algorithm attribute. To specify the algorithm attribute, you need to know what key manager algorithms are provided by the JDK you are using.

For example, a JDK that uses SunJSSE provides the PKIX and SunX509 algorithms.

In previous command you can specify SunX509 as the key manager algorithm attribute.

iv. Configure a **trust-manager** that references your truststore **key-store**.

```
/subsystem=elytron/trust-manager=twoWayTM:add(key-store=twoWayTS)
```

**IMPORTANT**

Red Hat did not specify the algorithm attribute in the previous command, because the Elytron subsystem uses `TrustManagerFactory.getDefaultAlgorithm()` to determine an algorithm by default. However, you can specify the algorithm attribute. To specify the algorithm attribute, you need to know what trust manager algorithms are provided by the JDK you are using. For example, a JDK that uses SunJSSE provides the PKIX and SunX509 algorithms.

In the previous command you can specify PKIX as the trust manager algorithm attribute.

v. Configure a **server-ssl-context** that references your **key-manager**, **trust-manager**, and enables client authentication:

```
/subsystem=elytron/server-ssl-context=twoWaySSC:add(key-manager=twoWayKM, protocols=["TLSv1.2"], trust-manager=twoWayTM, need-client-auth=true)
```

**IMPORTANT**

You need to determine what SSL/TLS protocols you want to support. The example command above uses TLSv1.2. You can use the cipher-suite-filter argument to specify which cipher suites are allowed, and the use-cipher-suites-order argument to honor server cipher suite order. The use-cipher-suites-order attribute by default is set to true. This differs from the legacy security subsystem behavior, which defaults to honoring client cipher suite order.
WARNING

Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.

vi. Check and see if the **https-listener** is configured to use a legacy security realm for its SSL configuration.

```
/subsystem=undertow/server=default-server/https-listener=https:read-attribute(name=security-realm)
{
    "outcome" => "success",
    "result" => "ApplicationRealm"
}
```

The above command shows that the **https-listener** is configured to use the **ApplicationRealm** legacy security realm for its SSL configuration. Undertow cannot reference both a legacy security realm and an **ssl-context** in the **elytron** subsystem at the same time. So you must remove the reference to the legacy security realm.

**NOTE**

If the result is undefined, you do not need to remove the reference to the security realm in the next step.

vii. Remove the reference to the legacy security realm, and update the **https-listener** to use the **ssl-context** from Elytron.

**NOTE**

**https-listener** must always have either a **security-realm** or **ssl-context** configured. When changing between the two configurations, the commands must be executed as a single batch, as shown below.

``` batch
/subsystem=undertow/server=default-server/https-listener=https:undefine-attribute(name=security-realm)
/subsystem=undertow/server=default-server/https-listener=https:write-attribute(name=ssl-context, value=twoWaySSC)
run-batch
```

viii. Reload the server.

``` reload ```
To complete the two-way SSL/TLS authentication, you need to import the server certificate into the client truststore and configure your client to present the client certificate.

```
$ keytool -importcert -keystore client.truststore.jks -storepass secret -alias localhost -trustcacerts -file /path/to/server.cer
```

ix. Configure your client to use the client certificate.

You need to configure your client to present the trusted client certificate to the server to complete the two-way SSL/TLS authentication. For example, if using a browser, you need to import the trusted certificate into the browser’s trust store.

This procedure forces a two-way SSL/TLS but it does not change the original authentication method of the application.

If you want to change the original authentication method, see Configure Authentication with Certificates in How to Configure Identity Management for JBoss EAP.

Two-way SSL/TLS is now enabled for applications.

**NOTE**

You can disable two-way SSL/TLS for deployed applications using the `security disable-ssl-http-server` command.

```
security disable-ssl-http-server
```

This command does not delete the Elytron resources. It configures the system to use the ApplicationRealm legacy security realm for its SSL configuration.

### 1.5.3. Configuring Certificate Revocation Using CRL in Elytron

Configure the trust manager used for enabling two-way SSL/TLS to use Certificate Revocation List (CRL) for certificate revocation in Elytron.

**Prerequisites**

- The trust manager is configured to use two-way SSL/TLS.
- The trust manager contains the certificate chain to be checked for revocation.

**Procedure**

1. Configure the trust manager to use CRLs obtained from distribution points referenced in your certificates.

   ```
   /subsystem=elytron/trust-manager=twoWayTM:write-attribute(name=certificate-revocation-list,value=[]) 
   ```

2. Override the CRL obtained from distribution points referenced in your certificates.
3. Configure **trust-manager** to use CRL for certificate revocation.
   
   - If an OCSP responder is also configured for certificate revocation, add attribute `ocsp.prefer-crls` with the value `true` in the trust manager to use CRL for certificate revocation:
     
     ```
     /subsystem=elytron/trust-manager=twoWayTM:write-attribute(name=ocsp.prefer-crls,value="true")
     ```
   
   - If no OCSP responder is configured for certificate revocation, the configuration is complete.

**Additional Information**

- For a complete list of CRL attributes, see [trust-manager Attributes](#).

### 1.5.4. Configuring Certification Revocation Using OCSP in Elytron

Configure the trust manager used for enabling two-way SSL/TLS to use an Online Certificate Status Protocol (OCSP) responder for certificate revocation. OCSP is defined in [RFC6960](#).

When both OCSP responder and CRL are configured for certificate revocation, the OCSP responder is invoked by default.

**Prerequisites**

- The trust manager is configured to use two-way SSL/TLS.

**Procedure**

1. Configure the trust manager to use the OCSP responder defined in the certificate for certificate revocation.

   ```
   /subsystem=elytron/trust-manager=twoWayTM:write-attribute(name=ocsp,value={})
   ```

2. Override OCSP responder defined in the certificate.

   ```
   /subsystem=elytron/trust-manager=twoWayTM:write-attribute(name=ocsp.responder,value="http://example.com/ocsp-responder")
   ```

**Additional Information**

- For a complete list of attributes, see [online-certificate-status Attributes](#).

### 1.5.5. Using Legacy Security Realms

---

Red Hat JBoss Enterprise Application Platform 7.4 How to Configure Server Security

80
IMPORTANT

As a prerequisite, an SSL/TLS encryption key and certificate should be created and placed in an accessible directory. Additionally, relevant information, such as keystore aliases and passwords, desired cipher suites, should also be accessible. For examples on generating SSL/TLS Keys and Certificates, see the first two steps in the Setting up Two-way SSL/TLS for the Management Interfaces section. For more information about the HTTPS listener, including cipher suites, see the HTTPS Listener Reference section.

1.5.5.1. Enable One-way SSL/TLS for Applications Using Legacy Security Realms

This example assumes that the keystore, identity.jks, has been copied to the server configuration directory and configured with the given properties. Administrators should substitute their own values for the example ones.

NOTE

The management CLI commands shown assume that you are running a JBoss EAP standalone server. For more details on using the management CLI for a JBoss EAP managed domain, see the JBoss EAP Management CLI Guide.

1. Add and configure an HTTPS security realm first. Once the HTTPS security realm has been configured, configure an https-listener in the undertow subsystem that references the security realm:

```
batch
/core-service=management/security-realm=HTTPSRealm:add
/core-service=management/security-realm=HTTPSRealm/server-identity=ssl:add(keystore-path=identity.jks, keystore-relative-to=jboss.server.config.dir, keystore-password=password1, alias=appserver)
/subsystem=undertow/server=default-server/https-listener=https:write-attribute(name=security-realm, value=HTTPSRealm)
run-batch
```

WARNING

Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.

2. Restart the JBoss EAP instance for the changes to take effect.

1.5.5.2. Enable Two-way SSL/TLS for Applications Using Legacy Security Realms
Setting up two-way SSL/TLS for applications follows many of the same procedures outlined in Setting up Two-way SSL/TLS for the Management Interfaces. To set up two-way SSL/TLS for applications, you need to do the following:

1. Generate the stores for both the client and server
2. Export the certificates for both the client and server
3. Import the certificates into the opposing truststores
4. Define a security realm, for example CertificateRealm, on the server that uses the server’s keystore and truststore
5. Update the undertow subsystem to use the security realm and require client verification

The first four steps are covered in Setting up Two-way SSL/TLS for the Management Interfaces.

**IMPORTANT**

If the server has not been reloaded since the new security realm has been added, you must reload the server before performing the next step.

**Update the Undertow Subsystem**

Once the keystores, certificates, truststores, and security realms have been created and configured, you need to add an HTTPS listener to the undertow subsystem, use the security realm you created, and require client verification:

```
/subsystem=undertow/server=default-server/https-listener=https:write-attribute(name=security-realm, value=CertificateRealm)
/subsystem=undertow/server=default-server/https-listener=https:write-attribute(name=verify-client, value=REQUIRED)
```

**IMPORTANT**

You must reload the server for these changes to take effect.

**IMPORTANT**

Any client connecting to a JBoss EAP instance with two-way SSL/TLS enabled for applications must have access to a client certificate or keystore, in other words a client keystore whose certificate is included in the server’s truststore. If the client is using a browser to connect to the JBoss EAP instance, you need to import that certificate or keystore into the browser’s certificate manager.

**NOTE**

More details on using certificate-based authentication in applications, in addition to two-way SSL/TLS with applications, can be found in the Configuring a Security Domain to Use Certificate-based Authentication section of the JBoss EAP How to Configure Identity Management Guide.
1.6. ENABLE HTTP AUTHENTICATION FOR APPLICATIONS USING THE CLI SECURITY COMMAND

In JBoss EAP, HTTP authentication, using an elytron HTTP authentication factory, can be enabled for the undertow security domain with the `security enable-http-auth-http-server` command. By default this command associates the application HTTP factory to the specified security domain.

**Example: Enable HTTP Authentication on the Undertow Security Domain**

```bash
security enable-http-auth-http-server --security-domain=SECURITY_DOMAIN
```

Server reloaded.
Command success.
Authentication configured for security domain SECURITY_DOMAIN
http authentication-factory=application-http-authentication
security-domain=SECURITY_DOMAIN

**NOTE**

Once the command is executed, the management CLI will reload the server and reconnect to it.

If an HTTP factory already exists, then the factory is updated to use the mechanism defined by the `--mechanism` argument.

1.6.1. Disabling HTTP authentication for the management interfaces

This procedure describe how to disable HTTP authentication for the management interfaces.

**Procedure**

- To remove the active HTTP authentication factory use the following command.

  ```bash
  security disable-http-auth-http-server --security-domain=SECURITY_DOMAIN
  ```

- Alternatively, you can use the following command to remove specific mechanisms from the active SASL authentication factory.

  ```bash
  security disable-http-auth-http-server --mechanism=MECHANISM --security-domain=SECURITY_DOMAIN
  ```

1.7. SASL AUTHENTICATION MECHANISMS

**Simple Authentication and Security Layer (SASL)** authentication mechanisms are used for defining the mechanisms for authenticating connections to a JBoss EAP server using the elytron subsystem, and for clients connecting to servers. Clients can be other JBoss EAP instances, or Elytron Client. SASL authentication mechanisms in JBoss EAP are also significantly used in Elytron Integration with the Remoting Subsystem.

1.7.1. Choosing SASL Authentication Mechanisms
NOTE

Although JBoss EAP and Elytron Client work with a variety of SASL authentication mechanisms, you must ensure that the mechanisms you use are supported. See this list for the support levels for SASL authentication mechanisms.

The authentication mechanisms you use depends on your environment and desired authentication method. The following list summarizes the use of some of the supported SASL authentication mechanisms:

**ANONYMOUS**
Unauthenticated guest access.

**DIGEST-MD5**
Uses HTTP digest authentication as a SASL mechanism.

**EXTERNAL**
Uses authentication credentials that are implied in the context of the request. For example, IPsec or TLS authentication.

**Mechanisms beginning with GS**
Authentication using Kerberos.

**JBOSS-LOCAL-USER**
Provides authentication by testing that the client has the same file access as the local user that is running the JBoss EAP server. This is useful for other JBoss EAP instances running on the same machine.

**OAUTHBEARER**
Uses authentication provided by OAuth as a SASL mechanism.

**PLAIN**
Plain text username and password authentication.

**Mechanisms beginning with SCRAM**
Salted Challenge Response Authentication Mechanism (SCRAM) that uses a specified hashing function.

**Mechanisms ending with -PLUS**
Indicates a channel binding variant of a particular authentication mechanism. You should use these variants when the underlying connection uses SSL/TLS.

For more information on individual SASL authentication mechanisms, see the IANA SASL mechanism list and individual mechanism memos.

1.7.2. Configuring SASL Authentication Mechanisms on the Server Side

Configuring SASL authentication mechanisms on the server side is done using SASL authentication factories.

There are two levels of configuration required:

- A `sasl-authentication-factory`, where you specify authentication mechanisms.

- A `configurable-sasl-server-factory` that aggregates one or more of `sasl-authentication-factory`, and configures mechanism properties as well as optionally applying filters to enable or disable certain authentication mechanisms.
The following example demonstrates creating a new `configurable-sasl-server-factory`, and a `sasl-authentication-factory` that uses DIGEST-MD5 as a SASL authentication mechanism for application clients.

```
/subsystem=elytron/configurable-sasl-server-factory=mySASLServerFactory:add(sasl-server-factory=elytron)
/subsystem=elytron/sasl-authentication-factory=mySASLAuthFactory:add(sasl-server-factory=mySASLServerFactory,security-domain=ApplicationDomain,mechanism-configurations=[[mechanism-name=DIGEST-MD5,mechanism-realm-configurations=[[realm-name=ApplicationRealm]]]])
```

1.7.3. Specifying SASL Authentication Mechanisms on the Client Side

SASL authentication mechanisms used by a client are specified using a `sasl-mechanism-selector`. You can specify any supported SASL authentication mechanisms that are exposed by the server that the client is connecting to.

A `sasl-mechanism-selector` is defined in Elytron configurations where authentication is configured:

- In the `elytron` subsystem, this is an attribute of an `authentication-configuration`. For example:
  ```
  /subsystem=elytron/authentication-configuration=myAuthConfig:write-attribute(name=sasl-mechanism-selector,value="DIGEST-MD5")
  ```
  An example of using an `authentication-configuration` with a `sasl-mechanism-selector` can be seen in Configuring SSL or TLS with `elytron`.

- For Elytron Client, it is specified under the `configuration` element of `authentication-configurations` in the client configuration file, usually named `wildfly-config.xml`. For example:
  ```
  <configuration>
    <authentication-client xmlns="urn:elytron:client:1.2">
      <authentication-rules>
        <rule use-configuration="default"/>
      </authentication-rules>
      <authentication-configurations>
        <configuration name="default">
          <sasl-mechanism-selector selector="#ALL"/>
          ...
        </configuration>
      </authentication-configurations>
    </authentication-client>
  </configuration>
  ```
  See *How to Configure Identity Management* for more information on configuring client authentication with Elytron Client.

`sasl-mechanism-selector` Grammar

The selector string for `sasl-mechanism-selector` has a specific grammar.

In a simple form, individual mechanisms are specified by listing their names in order, separated by a spaces. For example, to specify DIGEST-MD5, SCRAM-SHA-1, and SCRAM-SHA-256 as allowed authentication mechanisms, use the following string: `DIGEST-MD5 SCRAM-SHA-1 SCRAM-SHA-256`. 
More advanced usage of the grammar can use the following special tokens:

- **#ALL**: All mechanisms.
- **#FAMILY(NAME)**: Mechanisms belonging to the specified mechanism family. For example, the family could be DIGEST, EAP, GS2, SCRAM, or IEC-ISO-9798.
- **#PLUS**: Mechanisms that use channel binding. For example, SCRAM-SHA-XXX-PLUS or GS2-XXX-PLUS.
- **#MUTUAL**: Mechanisms that authenticate the server in some way, for example making the server prove that the server knows the password. **#MUTUAL** includes families such as **#FAMILY(SCRAM)** and **#FAMILY(GS2)**.
- **#HASH(ALGORITHM)**: Mechanisms that use the specified hash algorithm. For example, the algorithm could be MD5, SHA-1, SHA-256, SHA-384, or SHA-512.

The above tokens and names can also be used with the following operations and predicates:

- `-`: Forbids
- `!`: Inverts
- `&&`: And
- `||`: Or
- `==`: Equals
- `?`: If

**#TLS**: Is true when TLS is active, otherwise false.

Below are some examples of **sasl-mechanism-selector** strings and their meaning:

- **#TLS & & !#MUTUAL**: When TLS is active, all mechanisms without mutual authentication.
- **#ALL -ANONYMOUS**: All mechanisms, except for ANONYMOUS.
- **SCRAM-SHA-1 SCRAM-SHA-256**: Adds those two mechanisms in that order.
- **(SCRAM-SHA-1 || SCRAM-SHA-256)**: Adds the two mechanisms in the order that the provider or server presents them.
- **!#HASH(MD5)**: Any mechanism that does not use the MD5 hashing algorithm.
- **#FAMILY(DIGEST)**: Any digest mechanism.

### 1.7.4. Configuring SASL Authentication Mechanism Properties

You can configure authentication mechanism properties on both the server side and on the client side.

- On the server side, you define authentication mechanism properties in the **configurable-sasl-server-factory**. The following example defines the **com.sun.security.sasl.digest.utf8** property with a value of **false**.
On the client side, you define authentication mechanisms properties in the client’s authentication configuration:

- In the `elytron` subsystem, define the authentication mechanism properties in your `authentication-configuration`. The following example defines the `wildfly.sasl.local-user.quiet-auth` property with a value of `true`.

- For Elytron Client, authentication mechanism properties are specified under the `configuration` element of `authentication-configurations` in the client configuration file, usually named `wildfly-config.xml`. For example:

```xml
...<authentication-configurations>
  <configuration name="default">
    <sasl-mechanism-selector selector="#ALL"/>
    <set-mechanism-properties>
      <property key="wildfly.sasl.local-user.quiet-auth" value="true"/>
    </set-mechanism-properties>
  </configuration>
...</authentication-configurations>
```

You can see a list of standard Java SASL authentication mechanism properties in the Java documentation. Other JBoss EAP-specific SASL authentication mechanism properties are listed in the Authentication Mechanisms Reference.

### 1.8. ELYTRON INTEGRATION WITH THE MODCLUSTER SUBSYSTEM

One of the security capabilities exposed by `elytron` subsystem is a client `ssl-context` that can be used to configure the `modcluster` subsystem to communicate with a load balancer using SSL/TLS.

When protecting the communication between the application server and the load balancer, you need to define a client `ssl-context` in order to:

- Define a truststore holding the certificate chain that will be used to validate load balancer’s certificate.
- Define a trust manager to perform validations against the load balancer’s certificate.

#### 1.8.1. Defining a Client SSL Context and Configuring ModCluster Subsystem

The following procedure requires that a truststore and trust manager be configured. For information on creating these see Create an Elytron Truststore and Create an Elytron Trust Manager.

1. Create the client SSL context.
   This SSL context is going to be used by the `modcluster` subsystem when connecting to the load balancer using SSL/TLS:
2. Reference the newly created client SSL context using one of the following options.
   - Configure the modcluster subsystem by setting the ssl-context.
     
     ```
     /subsystem=modcluster/mod-cluster-config=configuration:write-attribute(name=ssl-context, value=modcluster-client-ssl-context)
     ```
   - Configure the undertow subsystem by defining the ssl-context attribute of the modcluster filter.
     
     ```
     /subsystem=undertow/configuration=filter/mod-cluster=modcluster:write-attribute(name=ssl-context,value=modcluster-client-ssl-context)
     ```

3. Reload the server.

   ```
   reload
   ```

For configuring the modcluster subsystem and using two-way authentication, along with the trust manager, the key manager also needs to be configured.

1. Create the keystore.

   ```
   /subsystem=elytron/key-store=twoWayKS:add(path=/path/to/client.keystore.jks, credential-reference={clear-text=secret},type=JKS)
   ```

2. Configure the key manager.

   ```
   /subsystem=elytron/key-manager=twoWayKM:add(key-store=twoWayKS, algorithm="SunX509", credential-reference={clear-text=secret})
   ```

3. Create the client SSL context.

   ```
   /subsystem=elytron/client-ssl-context=modcluster-client-ssl-context:add(trust-manager=default-trust-manager, key-manager=twoWayKM)
   ```

   **NOTE**

   If you already have an existing client SSL context, you can add the key-manager to it as follows:

   ```
   /subsystem=elytron/client-ssl-context=modcluster-client-ssl-context:write-attribute(name=key-manager, value=twoWayKM)
   ```

4. Reload the server.

   ```
   reload
   ```
1.9. ELYTRON INTEGRATION WITH THE JGROUPS SUBSYSTEM

Components in the elytron subsystem may be referenced when defining authorization or encryption protocols in the jgroups subsystem. Full instructions on configuring these protocols are found in the Securing a Cluster section of the Configuration Guide.

1.10. ELYTRON INTEGRATION WITH THE REMOTING SUBSYSTEM

1.10.1. Elytron integration with remoting connectors

A remoting connector is specified by a SASL authentication factory, a socket binding, and an optional SSL context. In particular, the attributes for a connector are as follows:

**sasl-authentication-factory**

A reference to the SASL authentication factory to use for authenticating requests to this connector. For more information on creating this factory, see Create an Elytron Authentication Factory.

**socket-binding**

A reference to the socket binding, detailing the interface and port where the connector should listen for incoming requests.

**ssl-context**

An optional reference to the server-side SSL Context to use for this connector. The SSL Context contains the server key manager and trust manager to be used, and should be defined in instances where SSL is desired.

For example, a connector can be added as follows, where `SASL_FACTORY_NAME` is an already defined authentication factory and `SOCKET_BINDING_NAME` is an existing socket binding.

```
/subsystem=remoting/connector=CONNECTOR_NAME:add(sasl-authentication-factory=SASL_FACTORY_NAME,socket-binding=SOCKET_BINDING_NAME)
```

If SSL is desired, a preconfigured `server-ssl-context` may be referenced using the `ssl-context` attribute, as seen below.

```
/subsystem=remoting/connector=CONNECTOR_NAME:add(sasl-authentication-factory=SASL_FACTORY_NAME,socket-binding=SOCKET_BINDING_NAME,ssl-context=SSL_CONTEXT_NAME)
```

1.10.1.1. Enabling one-way SSL/TLS for remoting connectors using the elytron subsystem

The following SASL mechanisms support channel binding to external secure channels, such as SSL/TLS:

- GS2-KRB5-PLUS
- SCRAM-SHA-1-PLUS
- SCRAM-SHA-256-PLUS
- SCRAM-SHA-384-PLUS
- SCRAM-SHA-512-PLUS
To use any of these mechanisms, you can configure a custom SASL factory, or modify one of the predefined SASL authentication factories. A SASL mechanism selector can be used on the client to specify the appropriate SASL mechanism.

Prerequisites

- A **key-store** is configured.
- A **key-manager** is configured.
- A **server-ssl-context** is configured that references the defined **key-manager**

Procedure

1. Create a **socket-binding** for the connector. The following command defines the **oneWayBinding** binding that listens on port 11199.

   ```
   /socket-binding-group=standard-sockets/socket-binding=oneWayBinding:add(port=11199)
   ```

2. Create a connector that references the SASL authentication factory, the previously created socket binding, and the SSL context.

   ```
   /subsystem=remoting/connector=oneWayConnector:add(sasl-authentication-factory=SASL_FACTORY,socket-binding=oneWayBinding,ssl-context=SSL_CONTEXT)
   ```

   **IMPORTANT**

   In cases where you have both a **security-realm** and **ssl-context** defined, JBoss EAP will use the SSL/TLS configuration provided by **ssl-context**.

3. Configure the client to trust the server certificate. A generic example client is found at Elytron Client Side One Way Example. This example configures an **ssl-context** using the client **trust-store**.

Additional resources

- **key-store**
- **key-manager**
- **server-ssl-context**

1.10.1.2. Enabling two-way SSL/TLS for remoting connectors using the elytron subsystem

The following SASL mechanisms support channel binding to external secure channels, such as SSL/TLS:

- GS2-KRB5-PLUS
- SCRAM-SHA-1-PLUS
- SCRAM-SHA-256-PLUS
- SCRAM-SHA-384-PLUS
To use any of these mechanisms, you can configure a custom SASL factory, or modify one of the predefined SASL authentication factories to offer any of these mechanisms. A SASL mechanism selector can be used on the client to specify the appropriate SASL mechanism.

Prerequisites

- Separate **key-store** components for the client and server certificates are configured.
- A **key-manager** for the server **key-store** is configured.
- A **trust-manager** for the server **trust-store** is configured.
- A **server-ssl-context** that references the defined **key-manager** and **trust-manager** is configured.

Procedure

1. Create a **socket-binding** for the connector. The following command defines the **twoWayBinding** binding that listens on port **11199**.

   ```
   /socket-binding-group=standard-sockets/socket-binding=twoWayBinding:add(port=11199)
   ```

2. Create a connector that references the SASL authentication factory, the previously created socket binding, and the SSL context.

   ```
   /subsystem=remoting/connector=twoWayConnector:add(sasl-authentication-factory=SASL_FACTORY,socket-binding=twoWayBinding,ssl-context=SSL_CONTEXT)
   ```

   **IMPORTANT**

   In cases where you have both a **security-realm** and **ssl-context** defined, JBoss EAP will use the SSL/TLS configuration provided by **ssl-context**.

3. Configure your client to trust the server certificate, and to present its certificate to the server. You need to configure your client to present the trusted client certificate to the server to complete the two-way SSL/TLS authentication. For example, if using a browser, you need to import the trusted certificate into the browser’s truststore. A generic example client is found at Elytron Client Side Two Way Example. This example configures an **ssl-context** using the client **trust-store** and **key-store**.

Two-way SSL/TLS is now enabled on the remoting connector.

Additional resources

- **key-store**
- **key-manager**
- **trust-manager**
- **trust-store**
1.10.2. Elytron integration with remoting HTTP connectors

A remote HTTP connection is specified by referencing a connector in the **undertow** subsystem and a SASL authentication factory defined in the **elytron** subsystem. The HTTP connector provides the configuration for the HTTP upgrade-based remoting connector, and connects to an HTTP listener specified by the **connector-ref** attribute.

The attributes for a **connector** are as follows:

- **connector-ref**
  - A reference to a predefined **undertow** listener.

- **sasl-authentication-factory**
  - A reference to the SASL authentication factory to use for authenticating requests to this connector. For more information on creating this factory, see [Create an Elytron Authentication Factory](#).

For example, a **http-connector** can be added as follows, where **CONNECTOR_NAME** references the **undertow** listener, and **SASL_FACTORY_NAME** is an already defined authentication factory in the **elytron** subsystem.

```
/subsystem=remoting/http-connector=HTTP_CONNECTOR_NAME:add(connector-ref=CONNECTOR_NAME,sasl-authentication-factory=SASL_FACTORY_NAME)
```

1.10.2.1. Enabling one-way SSL on the remoting HTTP connector

The following SASL mechanisms support channel binding to external secure channels, such as SSL/TLS:

- GS2-KRB5-PLUS
- SCRAM-SHA-1-PLUS
- SCRAM-SHA-256-PLUS
- SCRAM-SHA-384-PLUS
- SCRAM-SHA-512-PLUS

To use any of the above mechanisms, a custom SASL factory can be configured, or one of the predefined SASL authentication factories can be modified to offer any of these mechanisms. A **SASL mechanism selector** can be used on the client to specify the appropriate SASL mechanism.

**Prerequisites**

- A **key-store** is configured.
- A **key-manager** is configured.
- A **server-ssl-context** is configured that references the defined **key-manager**.

**Procedure**
1. Check whether the **https-listener** is configured to use a legacy security realm for its SSL configuration.

   ```
   /subsystem=undertow/server=default-server/https-listener=https:read-
   attribute(name=security-realm)
   {
     "outcome" => "success",
     "result" => "ApplicationRealm"
   }
   ```

   The above command shows that the **https-listener** is configured to use the **ApplicationRealm** legacy security realm for its SSL configuration. Undertow cannot reference both a legacy security realm and an **ssl-context** in Elytron at the same time so you must remove the reference to the legacy security realm.

   **NOTE**

   If the result is **undefined**, you do not need to remove the reference to the security realm in the next step.

2. Remove the reference to the legacy security realm, and update the **https-listener** to use the **ssl-context** from Elytron.

   **NOTE**

   **https-listener** must always have either a **security-realm** or **ssl-context** configured. When changing between the two configurations, the commands must be executed as a single batch, as shown below.

   ```
   batch
   /subsystem=undertow/server=default-server/https-listener=https:undefine-
   attribute(name=security-realm)
   /subsystem=undertow/server=default-server/https-listener=https:write-attribute(name=ssl-
   context, value=SERVER_SSL_CONTEXT)
   run-batch
   ```

3. Create an HTTP connector that references the HTTPS listener and the SASL authentication factory.

   ```
   /subsystem=remoting/http-connector=ssl-http-connector:add(connector-ref=https,sasl-
   authentication-factory=SASL_FACTORY)
   ```

4. Reload the server.

   ```
   reload
   ```

5. Configure the client to trust the server certificate. For example, if using a browser, you need to import the trusted certificate into the browser's truststore.

Additional resources

- **key-store**
1.10.2.2. Enabling two-way SSL/TLS on the remoting HTTP connectors

The following SASL mechanisms support channel binding to external secure channels, such as SSL/TLS:

- GS2-KRB5-PLUS
- SCRAM-SHA-1-PLUS
- SCRAM-SHA-256-PLUS
- SCRAM-SHA-384-PLUS
- SCRAM-SHA-512-PLUS

To use any of the above mechanisms, a custom SASL factory can be configured, or one of the predefined SASL authentication factories can be modified to offer any of these mechanisms. A SASL mechanism selector can be used on the client to specify the appropriate SASL mechanism.

Prerequisites

- Separate key-store components for the client and server certificates are configured.
- A key-manager for the server key-store is configured.
- A trust-manager for the server trust-store is configured.
- A server-ssl-context that references the defined key-manager and trust-manager is configured.

Procedure

1. Check whether the https-listener is configured to use a legacy security realm for its SSL configuration.

```
/subsystem=undertow/server=default-server/https-listener=https:read-attribute(name=security-realm)
{
  "outcome" => "success",
  "result" => "ApplicationRealm"
}
```

The above command shows that the https-listener is configured to use the ApplicationRealm legacy security realm for its SSL configuration. Undertow cannot reference both a legacy security realm and an ssl-context in Elytron at the same time so you must remove the reference to the legacy security realm.

**NOTE**

If the result is undefined, you do not need to remove the reference to the security realm in the next step.
2. Remove the reference to the legacy security realm, and update the **https-listener** to use the **ssl-context** from Elytron.

   **NOTE**

   The **https-listener** must always have either a **security-realm** or **ssl-context** configured. When changing between the two configurations, the commands must be executed as a single batch, as shown below.

   ```
   batch
   /subsystem=undertow/server=default-server/https-listener=https:undefine-
   attribute(name=security-realm)
   /subsystem=undertow/server=default-server/https-listener=https:write-attribute(name=ssl-
   context, value=SERVER_SSL_CONTEXT)
   run-batch
   ```

3. Create an HTTP connector that references the HTTPS listener and the SASL authentication factory.

   ```
   /subsystem=remoting/http-connector=ssl-http-connector:add(connector-ref=https,sasl-
   authentication-factory=SASL_FACTORY)
   ```

4. Reload the server.

   ```
   reload
   ```

5. Configure your client to trust the server certificate, and to present its certificate to the server. You need to configure your client to present the trusted client certificate to the server to complete the two-way SSL/TLS authentication. For example, if using a browser, you need to import the trusted certificate into the browser’s truststore.

   Two-way SSL/TLS is now enabled on the remoting HTTP connector.

   **IMPORTANT**

   In cases where you have both a **security-realm** and **ssl-context** defined, JBoss EAP will use the SSL/TLS configuration provided by **ssl-context**.

**Additional resources**

- **key-store**
- **key-manager**
- **trust-manager**
- **trust-store**
- **server-ssl-context**
- custom SASL factory

1.10.3. Elytron integration with remoting outbound connectors
A remote outbound connection is specified by an outbound socket binding and an authentication context. The authentication context provides all of the security information that is needed for the connection. In particular, the attributes for a remote-outbound-connection are as follows:

- **outbound-socket-binding-ref** - The name of the outbound socket binding, which is used to determine the destination address and port for the connection.

- **authentication-context** - A reference to the authentication context, which contains the authentication configuration and the defined SSL context, if one exists, required for the connection. For information on defining an authentication context, see Creating an Authentication Context.

For example, a remote-outbound-connection can be added as follows, where OUTBOUND_SOCKET_BINDING_NAME is an already defined outbound-socket-binding and AUTHENTICATION_CONTEXT_NAME is an authentication-context that has already been defined in the elytron subsystem configuration.

```
/subsystem=remoting/remote-outbound-connection=OUTBOUND_CONNECTION_NAME:add(authentication-context=AUTHENTICATION_CONTEXT_NAME, outbound-socket-binding-ref=OUTBOUND_SOCKET_BINDING_NAME)
```

Additional resources

- Creating an Authentication Context

### 1.11. ADDITIONAL ELYTRON COMPONENTS FOR SSL/TLS

The basic concepts behind configuring one-way SSL/TLS and two-way SSL/TLS are covered in the following:

- Enable One-way SSL/TLS for Applications Using the Elytron Subsystem
- Enable Two-way SSL/TLS for Applications Using the Elytron Subsystem
- Enable One-way SSL/TLS for the Management Interfaces Using the Elytron Subsystem
- Enable Two-way SSL/TLS for the Management Interfaces Using the Elytron Subsystem

Elytron also offers some additional components for configuring SSL/TLS.

### 1.11.1. Using an ldap-key-store

An ldap-key-store allows you to use a keystore stored in an LDAP server. You can use an ldap-key-store in the same way as you use a key-store.

**NOTE**

It is not possible to use a Jakarta Management ObjectName to decrypt the LDAP credentials. Instead, credentials can be secured by using a credential store. For information about credential stores, see Credential store in Elytron.

To create and use an ldap-key-store:
1. Configure a **dir-context**.
   To connect to the LDAP server from JBoss EAP, you need to configure a **dir-context** that provides the URL as well as the principal used to connect to the server.

   **Example: dir-context**
   ```
   /subsystem=elytron/dir-context=exampleDC:add(url="ldap://127.0.0.1:10389",
   principal="uid=admin,ou=system", credential-reference={clear-text="secret"})
   ```

2. Configure an **ldap-key-store**.
   When you configure an **ldap-key-store**, you need to specify both the **dir-context** used to connect to the LDAP server as well as how to locate the keystore stored in the LDAP server. At a minimum, this requires you to specify a **search-path**.

   **Example: ldap-key-store**
   ```
   /subsystem=elytron/ldap-key-store=ldapKS:add(dir-context=exampleDC, search-
   path="ou=Keystores,dc=wildfly,dc=org")
   ```

3. Use the **ldap-key-store**.
   Once you have defined your **ldap-key-store**, you can use it in the same places where a **key-store** could be used. For example, you could use an **ldap-key-store** when configuring One-way SSL/TLS and Two-way SSL/TLS for applications.

   For the full list of attributes for **ldap-key-store** as well as other Elytron components, see Elytron Subsystem Components Reference.

### 1.11.2. Using a filtering-key-store

A **filtering-key-store** allows you to expose a subset of aliases from an existing **key-store**, and use it in the same places you could use a **key-store**. For example, if a keystore contained `alias1`, `alias2`, and `alias3`, but you only wanted to expose `alias1` and `alias3`, a **filtering-key-store** provides you several ways to do that.

To create a **filtering-key-store**:

1. Configure a **key-store**.
   ```
   /subsystem=elytron/key-store=myKS:add(path=keystore.jks, relative-
   to=jboss.server.config.dir, credential-reference={clear-text=secret}, type=JKS)
   ```

2. Configure a **filtering-key-store**.
   When you configure a **filtering-key-store**, you specify which **key-store** you want to filter and the **alias-filter** for filtering aliases from the **key-store**. The filter can be specified in one of the following formats:
   - `alias1,alias3`, which is a comma-delimited list of aliases to expose.
   - `ALL:-alias2`, which exposes all aliases in the keystore except the ones listed.
   - `NONE:+alias1:+alias3`, which exposes no aliases in the keystore except the ones listed.

   This example uses a comma-delimited list to expose `alias1` and `alias3`. 
3. Use the `filtering-key-store`.
   Once you have defined your `filtering-key-store`, you can use it in the same places where a `key-store` could be used. For example, you could use a `filtering-key-store` when configuring One-way SSL/TLS and Two-way SSL/TLS for applications.

For the full list of attributes for `filtering-key-store` as well as other Elytron components, see Elytron Subsystem Components Reference.

1.11.3. Reload a Keystore

You can reload a keystore configured in JBoss EAP from the management CLI. This is useful in cases where you have made changes to certificates referenced by a keystore.

To reload a keystore:

```
/subsystem=elytron/key-store=httpsKS:load
```

1.11.4. Reinitialize a Key Manager

You can reinitialize a `key-manager` configured in JBoss EAP from the management CLI. This is useful in cases where you have made changes in certificates provided by keystore resource and you want to apply this change to new SSL connections without restarting the server.

```
/subsystem=elytron/key-manager=httpsKM:init()
```

To reinitialize a `key-manager`:

```
/subsystem=elytron/key-manager=httpsKM:init()
```

1.11.5. Reinitialize a Trust Manager

You can reinitialize a `trust-manager` configured in JBoss EAP from the management CLI or the management console. This is useful when you have made changes to certificates provided by a keystore resource and want to apply the changes to the new SSL connections without restarting the server.

Reinitializing a Trust Manager from the management CLI
NOTE

If the **key-store** is file based then it must be loaded first.

```
/subsystem=elytron/key-store=httpsKS:load()
```

To reinitialize a **trust-manager**:

```
/subsystem=elytron/trust-manager=httpsTM:init()
```

**Reinitializing a Trust Manager from the management console**

1. Navigate to the management console and click the **Runtime** tab.
2. In the **Monitor** column, click **Security (Elytron)**.
3. In the **Security** column, click **SSL → View**.
4. On the navigation pane, click **Trust Manager**.
5. Click **Initialize** on the top right corner of the screen to reinitialize a **trust-manager**.

### 1.11.6. Keystore Alias

The **alias** denotes the stored secret or credential in the store. If you add a keystore to the **elytron** subsystem using the **key-store** component, you can check the keystore's contents using the **alias** related **key-store** operations.

The different operations for alias manipulation are:

- **read-alias** - Read an alias from a keystore.
- **read-aliases** - Read aliases from a keystore.
- **remove-alias** - Remove an alias from a keystore.

For example, to read an alias:

```
/subsystem=elytron/key-store=httpsKS/:read-alias(alias=localhost)
```

### 1.11.7. Using a client-ssl-context

A **client-ssl-context** is used for providing an SSL context when the JBoss EAP instance creates an SSL connection as a client, such as using SSL in remoting.

To create a **client-ssl-context**:

1. Create **key-store**, **key-manager**, and **trust-manager** components as needed.

   If establishing a two-way SSL/TLS connection, you need to create separate **key-store** components for the client and server certificates, a **key-manager** for the client **key-store**, and a **trust-manager** for the server **key-store**. Alternatively, if you are doing a one-way SSL/TLS connection, you need to create a **key-store** for the server certificate and a **trust-manager** that references it. Examples on creating keystores and truststores are available in the **Enable Two-way SSL/TLS for Applications using the Elytron Subsystem** section.
2. Create a `client-ssl-context` referencing keystores, truststores, as well as any other necessary configuration options.

   **Example: client-ssl-context**

   ```
   /subsystem=elytron/client-ssl-context=exampleCSC:add(key-manager=clientKM, trust-manager=clientTM, protocols=["TLSv1.2"])
   ```

3. Reference the `client-ssl-context`.

   For the full list of attributes for `client-ssl-context` as well as other Elytron components, see Elytron Subsystem Components Reference.

### 1.11.8. Using a server-ssl-context

A `server-ssl-context` is used for providing a server-side SSL context. In addition to the usual configuration for an SSL context, it is possible to configure additional items such as cipher suites and protocols. The SSL context will wrap any additional items that are configured.

1. Create `key-store`, `key-manager`, and `trust-manager` components as needed.
   
   If establishing a two-way SSL/TLS connection, you need to create separate `key-store` components for the client and server certificates, a `key-manager` for the server `key-store`, and a `trust-manager` for the server `trust-store`. Alternatively, if you are doing a one-way SSL/TLS connection, you need to create a `key-store` for the server certificate and a `key-manager` that references it. Examples on creating keystores and truststores are available in the Enable Two-way SSL/TLS for Applications Using the Elytron Subsystem section.

2. Create a `server-ssl-context`.
   
   Create a `server-ssl-context` that references the key manager, trust manager, or any other desired configuration options using one of the options outlined below.

**Add a Server SSL Context Using the Management CLI**

```
/subsystem=elytron/server-ssl-context=newServerSSLContext:add(key-manager=KEY_MANAGER,protocols=["TLSv1.2"])
```

**IMPORTANT**

You need to determine what HTTPS protocols will be supported. The example commands above use TLSv1.2. You can use the `cipher-suite-filter` argument to specify which cipher suites are allowed, and the `use-cipher-suites-order` argument to honor server cipher suite order. The `use-cipher-suites-order` attribute by default is set to `true`. This differs from the legacy `security` subsystem behavior, which defaults to honoring client cipher suite order.

**Add a Server SSL Context Using the Management Console**

1. Access the management console. For more information, see the Management Console section in the JBoss EAP Configuration Guide.

2. Navigate to Configuration → Subsystems → Security (Elytron) → Other Settings and click View.
3. Click on SSL → Server SSL Context and click Add to configure a new server SSL context. For the full list of attributes for server-ssl-context as well as other Elytron components, see Elytron Subsystem Components Reference.

1.11.9. Using a server-ssl-sni-context

A server-ssl-sni-context is used for providing a server-side SNI matching. It provides matching rules to correlate host names to SSL contexts, along with a default in case none of the provided host names are matched. The SSL SNI contexts can be used in place of a standard server SSL context, such as when defining a context in the undertow subsystem.

1. Create key-store, key-manager, trust-manager, and server-ssl-context components as needed. There must be a server SSL context defined to create the server-ssl-sni-context.

2. Create a server-ssl-sni-context that provides matching information for the server-ssl-context elements. A default SSL context must be specified, using the default-ssl-context attribute, which will be used if no matching host names are found. The host-context-map accepts a comma-separated list of host names to match to the various SSL contexts.

```
/subsystem=elytron/server-ssl-sni-context=SERVER_SSL_SNI_CONTEXT:add(default-ssl-context=DEFAULT_SERVER_SSL_CONTEXT,host-context-map={HOSTNAME=SERVER_SSL_CONTEXT,...})
```

The following would be used to define a server-ssl-sni-context that defaults to the serverSSL SSL context, and matches incoming requests for www.example.com to the exampleSSL context.

```
/subsystem=elytron/server-ssl-sni-context=exampleSNIContext:add(default-ssl-context=serverSSL,host-context-map={www\.example\.com=exampleSSL})
```

**NOTE**

The attribute value for host matching works as a regular expression, so be sure to escape any periods (.) used to delimit the domain name.

Configure server-ssl-sni-context Using the Management Console

1. Access the management console. For more information, see the Management Console section in the JBoss EAP Configuration Guide.

2. Navigate to Configuration → Subsystems → Security (Elytron) → Other Settings and click View.

3. Click SSL → Server SSL SNI Context to configure the required ssl-sni-context.

For the complete list of attributes for Elytron components, see Elytron Subsystem Components Reference.

1.11.10. Custom SSL Components

When configuring SSL/TLS in the elytron subsystem, you can provide and use custom implementations of the following components:
- key-store
- key-manager
- trust-manager
- client-ssl-context
- server-ssl-context

**WARNING**

It is not recommended to provide custom implementations of any component outside of the **trust-manager** without an intimate knowledge of the Java Secure Socket Extension (JSSE).

**IMPORTANT**

When using FIPS it is not possible to utilize a custom trust manager or key manager, as FIPS requires these managers be embedded in the JDK for security reasons. Similar behavior can be accomplished by implementing a **SecurityRealm** that validates X509 evidences.

When creating custom implementations of Elytron components, they must present the appropriate capabilities and requirements. For more details on capabilities and requirements, see the Capabilities and Requirements section of the JBoss EAP Security Architecture guide. Implementation details for each component are provided by the JDK vendor.

### 1.11.10.1. Add a Custom Component to Elytron

The following steps describe adding a custom component within Elytron.

1. Add the JAR containing the provider for the custom component as a module into JBoss EAP, declaring any required dependencies, such as **javax.api**:

   ```
   module add --name=MODULE_NAME --resources=FACTORY_JAR --dependencies=javax.api,DEPENDENCY_LIST
   ```
IMPORTANT

Using the module management CLI command to add and remove modules is provided as Technology Preview only. This command is not appropriate for use in a managed domain or when connecting to the management CLI remotely. Modules should be added and removed manually in a production environment. For more information, see the Create a Custom Module Manually and Remove a Custom Module Manually sections of the JBoss EAP Configuration Guide.

Technology Preview features are not supported with Red Hat production service level agreements (SLAs), might not be functionally complete, and Red Hat does not recommend to use them for production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

See Technology Preview Features Support Scope on the Red Hat Customer Portal for information about the support scope for Technology Preview features.

2. When the component is added to the elytron subsystem the java.util.ServiceLoader will be used to discover the provider. Alternatively, a reference to the provider can be provided by defining a provider-loader. There are two methods of creating the loader, and only one should be implemented for each component.

- Reference the provider directly when defining the provider-loader:

  /subsystem=elytron/provider-loader=LOADER_NAME:add(class-names=[CLASS_NAME],module=MODULE_NAME)

- Include a reference to the provider in META-INF/services/java.security.Provider. This reference is automatically created when using the @MetaInfServices annotation in org.kohsuke.metainf-services. When using this method only the module needs to be referenced by the provider-loader, as seen below:

  /subsystem=elytron/provider-loader=LOADER_NAME:add(module=MODULE_NAME)

3. Add the custom component into Elytron’s configuration, using the appropriate element for the type to be added and referencing any defined providers.

  /subsystem=elytron/COMPONENT_NAME=NEW_COMPONENT:add(providers=LOADER_NAME,...)

For instance, to define a trust manager, the trust-manager element would be used, as seen in the following command:

Example: Adding a Custom Trust Manager

  /subsystem=elytron/trust-manager=newTrustManager:add(algorithm=MyX509,providers=customProvider,key-store=sampleKeystore)

4. Once defined, the component can be referenced from other elements.

Additional resources
1.11.10.2. Including Arguments in a Custom Elytron Component

You can include arguments within a custom component if your class implements the `initialize` method, as seen below.

```java
void initialize(final Map<String, String> configuration);
```

This method allows the custom class to receive a set of configuration strings when defined. These are passed in using the `configuration` attribute when defining the component. For instance, the following example defines an attribute named `myAttribute` with a value of `myValue`.

```
/subsystem=elytron/COMPONENT_NAME=NEW_COMPONENT:add(class-name=CLASS_NAME,module=MODULE_NAME,configuration={myAttribute="myValue"})
```

1.11.10.3. Using Custom Trust Managers with Elytron

By implementing a custom trust manager, it is possible to extend the validation of certificates when using HTTPS in Undertow, LDAPS in a `dir-context`, or any place where Elytron is used for SSL connections. This component is responsible for making trust decisions for the server, and it is strongly recommended that these be implemented if a custom trust manager is used.

**IMPORTANT**

When using FIPS it is not possible to utilize a custom trust manager, as FIPS requires this manager be embedded in the JDK for security reasons. Similar behavior can be accomplished by implementing a `SecurityRealm` that validates X509 evidences.

Requirements for Implementing a Custom Trust Manager

When using a custom trust manager, the following must be implemented:

- A trust manager that implements the `X509ExtendedTrustManager` interface.
- A trust manager factory that extends `TrustManagerFactorySpi`.
- The provider of the trust manager factory.

The provider must be included in the JAR file to be added into JBoss EAP. Any implemented classes must be included in JBoss EAP as a module. Classes are not required to be in one module, and can be loaded from module dependencies.

Example Implementations

The following example demonstrates a provider that registers the custom trust manager factory as a service.

Example: Provider

```java
import org.kohsuke.MetaInfServices;
import javax.net.ssl.TrustManagerFactory;
import java.security.Provider;
import java.util.Collections;
import java.util.List;
import java.util.Map;
```

For more information see modules and dependencies.
The following example demonstrates a custom trust manager. This trust manager contains overloaded methods on checking if a client or server is trusted.

**Example: TrustManager**

```java
@MetaInfServices(Provider.class)
public class CustomProvider extends Provider {

    public CustomProvider() {
        super("CustomProvider", 1.0, "Demo provider");

        System.out.println("CustomProvider initialization.");

        final List<String> emptyList = Collections.emptyList();
        final Map<String, String> emptyMap = Collections.emptyMap();

        putService(new Service(this, TrustManagerFactory.class.getSimpleName(),
                CustomAlgorithm, CustomTrustManagerFactorySpi.class.getName(), emptyList, emptyMap));
    }
}
```

The following example demonstrates a custom trust manager. This trust manager contains overloaded methods on checking if a client or server is trusted.

**Example: TrustManager**

```java
import javax.net.ssl.SSLEngine;
import javax.net.ssl.X509ExtendedTrustManager;
import java.net.Socket;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;

public class CustomTrustManager extends X509ExtendedTrustManager {

    public void checkClientTrusted(X509Certificate[] x509Certificates, String s, Socket socket) throws CertificateException {
        // Insert your code here
    }

    public void checkServerTrusted(X509Certificate[] x509Certificates, String s, Socket socket) throws CertificateException {
        // Insert your code here
    }

    public void checkClientTrusted(X509Certificate[] x509Certificates, String s, SSLEngine sslEngine) throws CertificateException {
        // Insert your code here
    }

    public void checkServerTrusted(X509Certificate[] x509Certificates, String s, SSLEngine sslEngine) throws CertificateException {
        // Insert your code here
    }

    public void checkClientTrusted(X509Certificate[] x509Certificates, String s) throws CertificateException {
        // Insert your code here
    }
}
```
The following example is a factory used to return instances of the trust manager.

**Example: TrustManagerFactorySpi**

```java
import javax.net.ssl.ManagerFactoryParameters;
import javax.net.ssl.TrustManager;
import javax.net.ssl.TrustManagerFactorySpi;
import java.security.InvalidAlgorithmParameterException;
import java.security.KeyStore;
import java.security.KeyStoreException;

public class CustomTrustManagerFactorySpi extends TrustManagerFactorySpi {

    protected void engineInit(KeyStore keyStore) throws KeyStoreException {
        // Insert your code here
    }

    protected void engineInit(ManagerFactoryParameters managerFactoryParameters) throws InvalidAlgorithmParameterException {
        // Insert your code here
    }

    protected CustomTrustManager[] engineGetTrustManagers() {
        // Insert your code here
    }
}
```

**Adding the Custom Trust Manager**

Once the provider and trust manager have been created, add them to the *elytron* subsystem by using the steps outlined in *Add a Custom Component to Elytron*.

### 1.11.1. Default SSLContext

Many libraries used within deployments might require SSL configuration for connections they establish. These libraries tend to be configurable by the caller. If no configuration is provided, they use the default `SSLContext` for the process.

The default `SSLContext` is available using the following method call:

```java
javax.net.ssl.SSLContext.getDefault();
```
By default this `SSLContext` is configured using system properties. However, within the `elytron` subsystem, it is possible to specify which one of the configured contexts should be associated and used as the default.

To make use of this feature, configure your `SSLContext` as normal. The following command can then be used to specify which `SSLContext` should be used as the default.

```bash
/subsystem=elytron:write-attribute(name=default-ssl-context, value=client-context)
```

As existing services and deployments could have cached the default `SSLContext` prior to this being set, a reload is required to ensure that the default gets set before the deployments are activated.

```bash
:reload
```

If the `default-ssl-context` attribute is subsequently `undefined`, the standard APIs do not provide any mechanism to revert the default. In this situation, the Java process would need be restarted.

```bash
/subsystem=elytron:undefine-attribute(name=default-ssl-context)
{
  "outcome" => "success",
  "response-headers" => {
    "operation-requires-restart" => true,
    "process-state" => "restart-required"
  }
}
```

### 1.11.12. Using a Certificate Revocation List

If you want to validate a certificate against a certificate revocation list (CRL), you can configure this using the `certificate-revocation-list` attribute for a trust manager in the `elytron` subsystem. For example:

```bash
/subsystem=elytron/trust-manager=TRUST_MANAGER:write-attribute(name=certificate-revocation-list,value={path=/path/to/CRL_FILE.crl.pem})
```

For more information on the available attributes for a trust manager, see the `trust-manager` attributes table.

**NOTE**

Your truststore must contain the certificate chain in order to check the validity of both the certification revocation list and the certificate. The truststore should not contain end-entity certificates, just certificate authority and intermediate certificates.

You can instruct the trust manager to reload the certificate revocation list by using the `reload-certificate-revocation-list` operation.

```bash
/subsystem=elytron/trust-manager=TRUST_MANAGER:reload-certificate-revocation-list
```

### 1.11.13. Using a Certificate Authority to Manage Signed Certificates
You can obtain and manage signed certificates using the JBoss EAP management CLI and the management console. This allows you to create a signed certificate directly from the CLI or the console and then import it into the required keystore.

**NOTE**

Many of the commands in this section have an optional staging parameter that indicates whether the certificate authority’s staging URL should be used. This value defaults to false, and is designed to assist in testing purposes. This parameter should never be enabled in a production environment.

**Configure a Let’s Encrypt Account**

As of JBoss EAP 7.4, Let’s Encrypt is the only supported certificate authority. To manage signed certificates an account must be created with the certificate authority, and the following information provided:

- A keystore to contain the alias of the certificate authority account key.
- The alias of the certificate authority. If the provided alias does not exist in the given keystore, then one will be created and stored as a private key entry.
- An optional list of URLs, such as email addresses, that the certificate authority can contact in the result of any issues.

```
/subsystem=elytron/certificate-authority-account=CERTIFICATE_ACCOUNT:add(key-store=KEYSTORE,alias=ALIAS,contact-urls=[mailto:EMAIL_ADDRESS])
```

**Create an Account with the Certificate Authority**

Once an account has been configured it may be created with the certificate authority by agreeing to their terms of service.

```
/subsystem=elytron/certificate-authority-account=CERTIFICATE_ACCOUNT:create-account(agree-to-terms-of-service=true)
```

**Update an Account with the Certificate Authority**

The certificate authority account options can be updated using the update-account command.

```
/subsystem=elytron/certificate-authority-account=CERTIFICATE_ACCOUNT:update-account(agree-to-terms-of-service=true)
```

**Change the Account Key Associated with the Certificate Authority**

The key associated with the certificate authority account can be changed by using the change-account-key command.

```
/subsystem=elytron/certificate-authority-account=CERTIFICATE_ACCOUNT:change-account-key()
```

**Deactivate the Account with the Certificate Authority**

If the account is no longer desired, then it may be deactivated by using the deactivate-account command.

```
/subsystem=elytron/certificate-authority-account=CERTIFICATE_ACCOUNT:deactivate-account()
```

**Get the Metadata Associated with the Certificate Authority**
The metadata for the account can be queried with the `get-metadata` command. This provides the following information:

- A URL to the terms of service.
- A URL to the certificate authority website.
- A list of the certificate authority accounts.
- Whether or not an external account is required.

```
/subsystem=elytron/certificate-authority-account=CERTIFICATE_ACCOUNT:get-metadata()
```

Configure a Let’s Encrypt Account Using Management Console
To configure a Let’s Encrypt account using the management console:

1. Access the management console.
   For more information, see the Management Console section in the JBoss EAP Configuration Guide.

2. Navigate to Runtime → Host → Security (Elytron) → SSL and click View.

3. Click Certificate Auth... to open the Certificate Authority Account page.

4. You can perform the following configurations for the selected alias by clicking the buttons with the labels:
   - **Create**
     Create an account with a certificate authority.
   - **Deactivate**
     Deactivate the selected certificate authority account.
   - **Update**
     Update the selected account with the certificate authority.
   - **Get Metadata**
     View the following information about the certificate authority account:
     - Associated alias
     - Certificate authority name
     - Contact details
     - Keystore name
     - Certificate authority details
   - **Change Account Key**
     Change the associated key with the certificate authority.

1.11.14. Keystore Manipulation Operations
You can perform various keystore manipulation operations on an Elytron key-store resource using the management CLI and the management console.
Keystore Manipulation Operations Using the Management CLI

Using the management CLI, you can perform the following keystore manipulation operations:

- Generate a key pair.
  The `generate-key-pair` command generates a key pair and wraps the resulting public key in a self-signed X.509 certificate. The generated private key and self-signed certificate will be added to the keystore.

  ```
  /subsystem=elytron/key-store=httpsKS:add(path=/path/to/server.keystore.jks,credential-reference={clear-text=secret},type=JKS)
  /subsystem=elytron/key-store=httpsKS:generate-key-pair(alias=example,algorithm=RSA,key-size=1024,validity=365,credential-reference={clear-text=secret},distinguished-name="CN=www.example.com")
  ```

- Generate a certificate signing request.
  The `generate-certificate-signing-request` command generates a PKCS #10 certificate signing request using a `PrivateKeyEntry` from the keystore. The generated certificate signing request will be written to a file.

  ```
  /subsystem=elytron/key-store=httpsKS:generate-certificate-signing-request(alias=example,path=server.csr,relative-to=jboss.server.config.dir,distinguished-name="CN=www.example.com",extensions=[[critical=false,name=KeyUsage,value=digitalSignature]],credential-reference={clear-text=secret})
  ```

- Import a certificate or certificate chain.
  The `import-certificate` command imports a certificate or certificate chain from a file into an entry in the keystore.

  ```
  /subsystem=elytron/key-store=httpsKS:import-certificate(alias=example,path=/path/to/certificate_or_chain/file,relative-to=jboss.server.config.dir,credential-reference={clear-text=secret},trust-cacerts=true)
  ```

- Export a certificate.
  The `export-certificate` command exports a certificate from an entry in the keystore to a file.

  ```
  /subsystem=elytron/key-store=httpsKS:export-certificate(alias=example,path=serverCert.cer,relative-to=jboss.server.config.dir,pem=true)
  ```

- Change an alias.
  The `change-alias` command moves an existing keystore entry to a new alias.

  ```
  /subsystem=elytron/key-store=httpsKS:change-alias(alias=example,new-alias=newExample,credential-reference={clear-text=secret})
  ```

- Store changes made to keystores.
  The `store` command persists any changes that have been made to the file that backs the keystore.

  ```
  /subsystem=elytron/key-store=httpsKS:store()
  ```
Keystore Manipulation Operations Using the Management Console

To perform the operations using the management console:

1. Access the management console.
   For more information, see the Management Console section in the JBoss EAP Configuration Guide.


3. Click Key Store to open the keystore definitions page.

4. Click the required keystore name.
   You can perform the following operations for the selected keystore by clicking the buttons with the labels:
   
   - **Load**
     Load or reload the keystore.
   
   - **Store**
     Persist changes made to the file backing the keystore.
   
   - **Generate Key Pair**
     Generate a key pair, wrap the public key in a self-signed X.509 certificate, and add the private key and the certificate to the keystore.
   
   - **Import Certificate**
     Import a certificate chain to the keystore from a file.
   
   - **Obtain**
     Obtain a signed certificate from a Certificate Authority and store it in the keystore.


You can perform the following operations on the keystore after you Configure a Let’s Encrypt Account.

**NOTE**

Many of the commands in this section have an optional staging parameter that indicates whether the certificate authority’s staging URL should be used. This value defaults to false, and is designed to assist in testing purposes. This parameter should never be enabled in a production environment.

Keystore Certificate Authority Operations Using the Management CLI

Using the management CLI, you can perform the following keystore certificate authority operations:

- **Obtain a Signed Certificate.**
  Once a certificate authority account has been defined for the keystore, you can use the obtain-certificate command to obtain a signed certificate and store it in the keystore. If an account with the certificate authority does not exist, then it will be automatically created.

  ```
  /subsystem=elytron/key-store=KEYSTORE:obtain-certificate(alias=ALIAS,domain-names=[DOMAIN_NAME],certificate-authority-account=CERTIFICATE_ACCOUNT,agree-to-terms-of-service=true,algorithm=RSA,credential-reference={clear-text=secret})
  ```

- **Revoke a signed certificate.**
The `revoke-certificate` command revokes a certificate that was issued by the certificate authority.

```
/subsystem=elytron/key-store=KEYSTORE:revoke-certificate(alias=ALIAS,certificate-authority-account=CERTIFICATE_ACCOUNT)
```

- Check if a signed certificate is due for renewal.
  The `should-renew-certificate` command determines if a certificate is due for renewal. The command returns `true` if the certificate expires in less than the given number of days, and `false` otherwise.

The following command determines if the certificate expires in the next 7 days.

```
/subsystem=elytron/key-store=KEYSTORE:should-renew-certificate(alias=ALIAS,expiration=7)
```

### Keystore Certificate Authority Operations Using the Management Console

To perform the operations using the management console:

1. Access the management console.
   For more information, see the Management Console section in the JBoss EAP Configuration Guide.
3. Click Key Store to open the keystore definitions page.
4. Click Aliases next to the required keystore name.
5. Click the required alias name.
   You can perform the following operations for the selected alias by clicking on the buttons with the labels:
   - **Change Alias**
     Change the alias for the entry.
   - **Export Certificate**
     Export a certificate from a keystore entry to a file.
   - **Generate CSR**
     Generate a certificate signing request.
   - **Remove Alias**
     Remove the selected alias from the keystore.
   - **Details**
     View the details of the certificate associated with the alias.
   - **Revoke**
     Revoke the certificate associated with the alias.
   - **Verify Renew**
     Determine if the associated certificate is due for renewal.
1.11.15. Configuring Evidence Decoder for X.509 Certificate with Subject Alternative Name Extension

By default, the principal associated with an X.509 certificate in Elytron is the subject name in the certificate and the principal associated with an X.509 certificate chain is the subject name in the first certificate in a certificate chain. You can configure an `X509-subject-alt-name-evidence-decoder` to use subject alternative name extension in an X.509 certificate as the principal.

The subject alternative name extension specification for an X.509 certificate and an X.509 certificate chain is defined in `RFC 5280`.

**Prerequisites**

- You know the expected format of a client certificate, or you have a client certificate available locally.

**Procedure**

1. Identify which subject alternative name extension to use.
   If you have the client certificate locally, the subject alternative name extension can be viewed using the `keytool` command:
   ```
   keytool -printcert -file /path/to/certificate/certificate.cert
   ```
   The subject alternative name extension is listed as:
   ```
   SubjectAlternativeName [  
   DNS:one.example.org  
   IP Address:127.0.0.1  
   ]
   ```

2. Create an `x509-subject-alt-name-evidence-decoder` to use the identified subject alternative name:
   ```
   /subsystem=elytron/x509-subject-alt-name-evidence-decoder=exampleDnsDecoder:add(alt-name-type=__EXTENSION_TO_USE__)
   ```

   - To use the evidence decoder, reference it in a security-domain:
     ```
     /subsystem=elytron/security-domain=__Security_Domain_Name__:write-attribute(name="evidence-decoder",value="exampleDnsDecoder")
     ```

**Additional resources**

- `x509-subject-alternative-name-evidence-decoder Attributes`

1.11.16. Configuring an Aggregate Evidence Decoder

You can configure an aggregate evidence decoder to combine two or more evidence decoders. The evidence decoders are applied in the configured order until an evidence decoder returns a non-null principal or until there are no more evidence decoders left to try.

**Prerequisites**
The evidence decoders to be aggregated are configured. For information about configuring evidence decoder, see Configuring Evidence Decoder for X.509 Certificate with Subject Alternative Name Extension.

**Procedure**

1. Create an aggregate evidence decoder from existing the evidence decoders:

   ```
   /subsystem=elytron/aggregate-evidence-decoder=aggregateDecoder:add(evidence-decoders=[__DECODER_1__, __DECODER_2__, ..., __DECODER_N__])
   ```

   • To use the evidence decoder, reference it in a security domain:

   ```
   /subsystem=elytron/security-domain=__SECURITY_DOMAIN__:write-attribute(name="evidence-decoder", value="aggregateDecoder")
   ```

1.11.17. Configuring X.500 Subject Evidence Decoder

Configure `x500-subject-evidence-decoder` to extract the subject from the first certificate in a certificate chain.

**Procedure**

• Create an x.500 subject evidence decoder:

  ```
  /subsystem=elytron/x500-subject-evidence-decoder=exampleSubjectDecoder:add()
  ```

1.11.18. Using Custom Evidence Decoder Implementation

You can use a custom `org.wildfly.security.auth.server.EvidenceDecoder` implementation in Elytron by adding it as a module to JBoss EAP.

**Procedure**

1. Package the custom implementation class as a Java Archive (JAR).

2. Add a module to JBoss EAP containing the JAR.

   For information about adding modules to JBoss EAP, see Create a Custom Module section in the Configuration Guide.

3. Add the custom evidence decoder to Elytron:

   ```
   /subsystem=elytron/custom-evidence-decoder=myCustomEvidenceDecoder:add(module=__MODULE_NAME__, class-name=__FULLY_QUALIFIED_CLASS_NAME__)
   ```
CHAPTER 2. SECURING A MANAGED DOMAIN

You can secure communication between a managed domain controller and its host controllers.

2.1. CONFIGURING PASSWORD AUTHENTICATION FOR THE DOMAIN CONTROLLER WITH ELYTRON

You need to add a user to the master domain controller so that the slave controller can authenticate as the user. The slave controller attempts to authenticate at the HTTP interface of the master domain controller.

Procedure

1. Add a user on the master domain controller. Use the `add-user` utility to add the user name, password, and other configurations. If the HTTP interface is secured with the `ManagementRealm` Elytron security realm, you must add a user to `ManagementRealm`.

   NOTE
   If you are using the default file based user and group authentication mechanism, run the `EAP_HOME/bin/add-user.sh` script.

   NOTE
   After you use the `add-user` utility to add user information, the server caches the contents of the properties files in memory. However, the server does check the modified time of the properties files on each authentication request and reloads if the time has been updated. This means that all changes made by the `add-user` utility are immediately applied to any running server.

   NOTE
   The default name of the realm for management users is `ManagementRealm`. When the `add-user` utility prompts for the realm name, you must accept the default realm name; that is, unless you have switched to a different realm.

2. Add an `authentication-configuration` to the slave controller. The following example demonstrates adding a new `authentication-configuration` called `slave` with the user `slave` and the `password1!`:

   /host=slave/subsystem=elytron/authentication-configuration=slave:add(authentication-name=slave, credential-reference={clear-text=password1!})

3. Add an `authentication-context` to the slave controller, as demonstrated in the following example:

   /host=slave/subsystem=elytron/authentication-context=slave-context:add(match-rules= [{authentication-configuration=slave}])

4. Specify the domain controller location and `authentication-context` in the slave controller, as demonstrated in the following example:
2.2. CONFIGURING PASSWORD AUTHENTICATION FOR THE DOMAIN CONTROLLER WITH LEGACY CORE-MANAGEMENT AUTHENTICATION

By default, Red Hat JBoss Enterprise Application Platform configures the master domain controller to require authentication from each slave controller that connects to the master domain controller.

Use the procedure to configure slave controllers with the proper credentials.

Procedure

1. Use the add-user script to add a user to the master domain controller.
   a. Check that the user is added to the same realm the master uses to secure its management interface, which by default is ManagementRealm.
   b. Add a slave user, as demonstrated in the following example. Select yes for the Is this new user going to be used for one AS process to connect to another AS process? question.

   ```
   $ EAP_HOME/bin/add-user.sh
   
   What type of user do you wish to add?
   a) Management User (mgmt-users.properties)
   b) Application User (application-users.properties)
   (a): a
   
   Enter the details of the new user to add.
   Using realm 'ManagementRealm' as discovered from the existing property files.
   Username : slave-user
   Password recommendations are listed below. To modify these restrictions edit the add-user.properties configuration file.
   - The password should be different from the username
   - The password should not be one of the following restricted values {root, admin, administrator}
   - The password should contain at least 8 characters, 1 alphabetic character(s), 1 digit(s), 1 non-alphanumeric symbol(s)
   Password :
   Re-enter Password :
   What groups do you want this user to belong to? (Please enter a comma separated list, or leave blank for none)[  ]:
   About to add user 'slave-user' for realm 'ManagementRealm'
   ```
Is this correct yes/no? yes
Added user 'slave-user' to file '/home/user/EAP-7.4.0/standalone/configuration/mgmt-users.properties'
Added user 'slave-user' to file '/home/user/EAP-7.4.0/domain/configuration/mgmt-users.properties'
Added user 'slave-user' with groups to file '/home/user/EAP-7.4.0/standalone/configuration/mgmt-groups.properties'
Added user 'slave-user' with groups to file '/home/user/EAP-7.4.0/domain/configuration/mgmt-groups.properties'
Is this new user going to be used for one AS process to connect to another AS process? e.g. for a slave host controller connecting to the master or for a Remoting connection for server to server EJB calls.
yes/no? yes
To represent the user add the following to the server-identities definition
<secret value="ABCzc3dv11Qx"/>

IMPORTANT
After adding the user, the script outputs a <secret> element. You need to use this element in the next step.

2. Configure the slave controllers to use the credential. After you created the user on the master domain controller, you must update each slave controller to use that credential in the host configuration file. For example, host.xml or host-slave.xml.
The following example shows the addition of the user name to the <remote> element in the domain controller configuration. Further the example shows the addition of the <secret> to the server-identities of the realm that is used to secure the <remote> element.

NOTE
Both the user name and <secret> were obtained by adding the user to the master domain controller in the previous step.

...<security-realm name="ManagementRealm">
  <server-identities>
    <!-- Replace this with either a base64 password of your own, or use a vault with a vault expression -->
    <secret value="ABCzc3dv11Qx"/>
  </server-identities>
...
  <domain-controller>
    <remote security-realm="ManagementRealm" username="slave-user">
      <discovery-options>
        <static-discovery name="primary" protocol="${jboss.domain.master.protocol:remote}" host="${jboss.domain.master.address}" port="${jboss.domain.master.port:9990}"/>
      </discovery-options>
    </remote>
  </domain-controller>

Additional resources
2.3. CONFIGURING SSL OR TLS FOR THE DOMAIN CONTROLLER WITH ELYTRON

You can configure JBoss EAP instances in a managed domain to use Secure Sockets Layer (SSL) or Transport Layer Security (TLS) when communicating with each other, in other words, between the master domain controller and host controllers.

IMPORTANT

When you configure SSL or TLS to be used between JBoss EAP instances in a managed domain, each instance can have a client or server role depending on the interaction. This includes all host controllers and domain controllers. For best results, set up two-way SSL or TLS between endpoints.

Prerequisites

- Generated and configured all necessary certificates and keystores. To enable two-way SSL/TLS for the management interfaces, see either Enabling two-way SSL/TLS using a security command or Enabling two-way SSL/TLS using the Elytron subsystem commands.

NOTE

To set up two-way SSL or TLS between endpoints, you need to generate and configure certificates and keystores for the master domain controller and each host controller.

Additionally, you must import the certificate of the master domain controller into each host controller keystore. Further, import each host controller certificate into the master domain controller keystore.

Procedure

1. Add a user on the master domain controller. If you are re-using the default file based user and group authentication mechanism, run the `EAP_HOME/bin/add-user.sh` script. When prompted, add the username, password and other configurations.

NOTE

The default name of the realm for management users is `ManagementRealm`. When the `add-user` utility prompts for the realm name, you must accept the default realm name; that is, unless you have switched to a different realm.

NOTE

You must add a user on the master domain controller for the slave controller to authenticate the user.
NOTE

The server caches the contents of the properties files in memory. However, the server does check the modified time of the properties files on each authentication request and reloads if the time has been updated. Therefore, any changes by the add-user utility are immediately applied to any running server.

2. Configure the master domain controller to use SSL or TLS. The following example shows commands for configuring domain controller key-store, key-manager, trust-manager, and server-ssl-context for the server keystore and truststore.

```
/host=master/subsystem=elytron/key-store=twoWayKS:add(path=/path/to/server.keystore.jks,credential-reference={clear-text=secret},type=JKS)
```

```
/host=master/subsystem=elytron/key-store=twoWayTS:add(path=/path/to/server.truststore.jks,credential-reference={clear-text=secret},type=JKS)
```

```
/host=master/subsystem=elytron/key-manager=twoWayKM:add(key-store=twoWayKS,credential-reference={clear-text=secret})
```

```
/host=master/subsystem=elytron/trust-manager=twoWayTM:add(key-store=twoWayTS)
```

```
/host=master/subsystem=elytron/server-ssl-context=twoWaySSC:add(key-manager=twoWayKM,protocols=["TLSv1.2"],trust-manager=twoWayTM,want-client-auth=true,need-client-auth=true)
```

```
/host=master/core-service=management/management-interface=http-interface:write-attribute(name=ssl-context, value=twoWaySSC)
```

IMPORTANT

Red Hat did not specify the algorithm attribute in the previous command, because the Elytron subsystem uses KeyManagerFactory.getDefaultAlgorithm() and TrustManagerFactory.getDefaultAlgorithm() to determine an algorithm by default. However, you can specify the algorithm attribute. To specify the algorithm attribute, you need to know what key manager algorithms are provided by the JDK you are using. For example, a JDK that uses SunJSSE provides the PKIX and SunX509 algorithms.

In the previous command you can specify SunX509 as the key manager algorithm attribute and PKIX as the trust manager algorithm attribute.

Further, you need to determine what HTTPS protocols you want to support. The examples in this procedure use TLSv1.2.

You can use the cipher-suite-filter to specify cipher suites, and the use-cipher-suites-order argument to honor server cipher suite order. The use-cipher-suites-order attribute by default is set to true. This differs from the legacy security subsystem behavior, which defaults to honoring client cipher suite order.
3. Configure an authentication context and domain controller location on each slave host controller. The following example configuration shows a domain controller that exists on localhost.

**NOTE**

You must specify the correct management user, password, and domain controller location for your environment.

```
/host=slave1/subsystem=elytron/authentication-context=slaveHostSSLContext:add()
/host=slave1/subsystem=elytron/authentication-configuration=slaveHostSSLConfiguration:add()
/host=slave1/subsystem=elytron/authentication-configuration=slaveHostSSLConfiguration:write-attribute(name=sasl-mechanism-selector,value=DIGEST-MD5)
/host=slave1/subsystem=elytron/authentication-configuration=slaveHostSSLConfiguration:write-attribute(name=authentication-name,value=slave)
/host=slave1/subsystem=elytron/authentication-configuration=slaveHostSSLConfiguration:write-attribute(name=realm,value=ManagementRealm)
/host=slave1/subsystem=elytron/authentication-configuration=slaveHostSSLConfiguration:write-attribute(name=credential-reference,value=\{clear-text=password1!\})
/host=slave1/subsystem=elytron/authentication-context=slaveHostSSLContext:write-attribute(name=match-rules,value=\[{match-host=localhost,authentication-configuration=slaveHostSSLConfiguration}\]}
/host=slave1:write-remote-domain-controller(host=localhost,port=9990,protocol=remote,authentication-context=slaveHostSSLContext)
```

4. Configure each slave host controller to use SSL or TLS. The following commands configure the slave host controller `key-store`, `key-manager`, `trust-manager`, `client-ssl-context`, and `authentication-context` for the server keystore and truststore. Further, the example shows a domain controller that exists on localhost.

**NOTE**

You must specify the correct domain controller location for your environment.

```
/host=slave1/subsystem=elytron/key-store=twoWayKS:add(path=/path/to/client.keystore.jks,credential-reference=\{clear-text=secret\},type=JKS)
/host=slave1/subsystem=elytron/key-store=twoWayTS:add(path=/path/to/client.truststore.jks,credential-reference=\{clear-text=secret\},type=JKS)
```
/host=slave1/subsystem=elytron/key-manager=twoWayKM:add(key-store=twoWayKS,credential-reference={clear-text=secret})

/host=slave1/subsystem=elytron/trust-manager=twoWayTM:add(key-store=twoWayTS)

/host=slave1/subsystem=elytron/client-ssl-context=twoWayCSC:add(key-manager=twoWayKM,protocols=["TLSv1.2"],trust-manager=twoWayTM)

/host=slave1/subsystem=elytron/authentication-context=slaveHostSSLContext:write-attribute(name=match-rules,value=[[match-host=localhost,authentication-configuration=slaveHostSSLConfiguration,ssl-context=twoWayCSC]])

IMPORTANT

Red Hat did not specify the algorithm attribute in the previous command, because the Elytron subsystem uses `KeyManagerFactory.getDefaultAlgorithm()` and `TrustManagerFactory.getDefaultAlgorithm()` to determine an algorithm by default. However, you can specify the algorithm attribute. To specify the algorithm attribute, you need to know what key manager algorithms are provided by the JDK you are using. For example, a JDK that uses SunJSSE provides the PKIX and SunX509 algorithms.

In the previous command you can specify SunX509 as the key manager algorithm attribute and PKIX as the trust manager algorithm attribute.

Further, you need to determine what HTTPS protocols you want to support. The examples in this procedure use TLSv1.2.

You can use the `cipher-suite-filter` to specify cipher suites, and the `use-cipher-suites-order` argument to honor server cipher suite order. The `use-cipher-suites-order` attribute by default is set to true. This differs from the legacy `security` subsystem behavior, which defaults to honoring client cipher suite order.

5. Reload all the JBoss EAP hosts in your managed domain.

Additional resources

- For information about concepts and general configuration for the managed domain operating mode, see the Domain Management section of the JBoss EAP Configuration Guide.

- For information about managing users, see the Management Users section of the JBoss EAP Configuration Guide.

2.4. CONFIGURING SSL IN DOMAIN MODE BY USING ELYTRON

In JBoss EAP 7.1 or later versions, you can use Elytron to configure SSL in domain mode.

Prerequisites

- JBoss EAP 7.1 or later.

- Elytron
Procedure

1. Create a self-signed certificate to enable SSL:

   keytool -genkey -alias jboss -keysize 2048 -validity 365 -keyalg RSA -sigalg SHA256withRSA -keystore jboss.jks -storepass jboss@123 -keypass jboss@123 -dname "CN=example.com, OU=JavaEE, O=Red Hat, C=IN"

2. Create a keystore, a key-manager and an ssl-context using the management CLI.

   #Configure a keystore
   /profile=<profile-name>/subsystem=elytron/key-store=httpsKS:add(path="${jboss.home.dir}/ssl/jboss.jks", credential-reference={clear-text=jboss@123}, type=JKS)

   #Create a new key-manager
   /profile=<profile-name>/subsystem=elytron/key-manager=httpsKM:add(key-store=httpsKS,algorithm="SunX509",credential-reference={clear-text=jboss@123})

   #Configure new server-ssl-context reference with protocol and ciphers
   /profile=<profile-name>/subsystem=elytron/server-ssl-context=httpsSSC:add(key-manager=httpsKM,protocols=["TLSv1.2"])

3. Configure the undertow subsystem to map the Elytron ssl-context:

   batch
   /profile=<profile-name>/subsystem=undertow/server=default-server/https-listener=https:undefine-attribute(name=security-realm)
   /profile=<profile-name>/subsystem=undertow/server=default-server/https-listener=https:write-attribute(name=ssl-context,value=httpsSSC)
   run-batch

4. Optional: Secure the management-interface to use the same ssl-context.
   The host-*.xml files define the configuration for the domain controllers and host controllers, which hold the management interfaces. To ensure that SSL is configured successfully, you must define the ssl-context again on the host.

   #Configure a keystore on the master DC host
   /host=<host-name>/subsystem=elytron/key-store=httpsKS:add(path="${jboss.home.dir}/ssl/jboss.jks", credential-reference={clear-text=jboss@123}, type=JKS)

   #Create a new key-manager on the master DC host
   /host=<host-name>/subsystem=elytron/key-manager=httpsKM:add(key-store=httpsKS,algorithm="SunX509",credential-reference={clear-text=jboss@123})

   #Configure new server-ssl-context reference with protocol and ciphers on the master DC host
   /host=<host-name>/subsystem=elytron/server-ssl-context=httpsSSC:add(key-manager=httpsKM,protocols=["TLSv1.2"])

   #Configure the secure-port and ssl-context for management-http interface on the master DC host
   /host=<host-name>/core-service=management/management-interface=http-interface:write-attribute(name=ssl-context,value=httpsSSC)
5. Ensure that a truststore is properly configured to enable the remote host controllers to connect to the domain controller through SSL. For more information, see Configuring SSL/TLS Between Domain and Host Controllers Using Elytron.

6. Reload the servers to ensure that the changes are effective:

   ```
   reload --host=<host-name>
   ```

**Verification**

- Verify the TLS connection by using a browser or openssl on the Red Hat Enterprise Linux command line:

  ```
  openssl s_client -connect host:8443
  ```

  The output displays information about the certificate and the TLS version used.

**2.5. CONFIGURING SSL OR TLS FOR THE LEGACY CORE MANAGEMENT AUTHENTICATION MECHANISM**

You can configure JBoss EAP instances in a managed domain to use Secure Sockets Layer (SSL) or Transport Layer Security (TLS) when communicating with each other, in other words, between the master domain controller and host controllers.

**IMPORTANT**

When you configure SSL or TLS to be used between JBoss EAP instances in a managed domain, each instance can have a client or server role depending on the interaction. This includes all host controllers and domain controllers. For best results, set up two-way SSL or TLS between endpoints.

**Prerequisites**

- Generated and configured all necessary certificates and keystores. To enable two-way SSL/TLS for the management interfaces, see Setting up two-way SSL/TLS for the management interfaces with legacy core management authentication.

**NOTE**

To set up two-way SSL or TLS between endpoints, you need to generate and configure certificates and keystores for the master domain controller and each host controller.

Additionally, you must import the certificate of the master domain controller into each host controller keystore. Further, import each host controller certificate into the master domain controller keystore.
Procedure

1. Configure the master domain controller to use SSL or TLS as demonstrated in the following example. When you have configured all the certificates and the keystores, you need to configure a security realm to use two-way SSL/TLS. You can achieve this by configuring a security realm to use SSL/TLS. The configured security realm secures the management interface used for connecting between host controllers and the master domain controller.

   **NOTE**

   Run the following commands in either batch mode or on the server. You must reload the server after adding the ssl server identity. The example in this procedure is run in batch mode.

   ```batch
   /host=master/core-service=management/security-realm=CertificateRealm:add()
   /host=master/core-service=management/security-realm=CertificateRealm/authentication=truststore:add(keystore-relative-to=jboss.domain.config.dir,keystore-path=domaincontroller.jks,keystore-password=secret)
   /host=master/core-service=management/security-realm=CertificateRealm/authentication=local:add(default-user=$local)
   /host=master/core-service=management/security-realm=CertificateRealm/authentication=properties:add(relative-to=jboss.domain.config.dir,path=mgmt-users.properties)
   /host=master/core-service=management/management-interface=http-interface:write-attribute(name=security-realm,value=CertificateRealm)
   run-batch
   ```

2. Configure all host controllers to use SSL or TLS, as demonstrated in the following example:

   ```batch
   /host=instance1/core-service=management/security-realm=CertificateRealm:add()
   /host=instance1/core-service=management/security-realm=CertificateRealm/server-identity=ssl:add(alias=instance1,keystore-relative-to=jboss.domain.config.dir,keystore-path=instance1.jks,keystore-password=secret)
   /host=instance1/core-service=management/security-realm=CertificateRealm/authentication=truststore:add(keystore-relative-to=jboss.domain.config.dir,keystore-path=instance1.jks,keystore-password=secret)
   /host=instance1/core-service=management/security-realm=CertificateRealm/authentication=local:add(default-user="$local")
   ```
3. Update the security realm that is used when connecting the master domain controller. You must make this update to the host controller configuration file whilst the server is not running. For example, *host.xml* or *host-slave.xml*.

```xml
<domain-controller>
  <remote security-realm="CertificateRealm" username="slave-user">
    <discovery-options>
      <static-discovery name="primary" protocol="${jboss.domain.master.protocol:remote}" host="${jboss.domain.master.address}" port="${jboss.domain.master.port:9990}"/>
    </discovery-options>
  </remote>
</domain-controller>
```

Additional resources

- For information about concepts and general configuration for the managed domain operating mode, see the Domain Management section of the JBoss EAP Configuration Guide.

- For information about the process of setting up a two-way SSL or TLS, see Setting up Two-way SSL/TLS for the Management Interfaces with Legacy Core Management Authentication in the How to Configure Server Security guide.
CHAPTER 3. SECURING USERS OF THE SERVER AND ITS MANAGEMENT INTERFACES

3.1. USER AUTHENTICATION WITH ELYTRON

3.1.1. Default Configuration

By default, the JBoss EAP management interfaces are secured by the legacy core management authentication.

Example: Default Configuration

```plaintext
/core-service=management/management-interface=http-interface:read-resource()
{
  "outcome" => "success",
  "result" => {
    "allowed-origins" => undefined,
    "console-enabled" => true,
    "http-authentication-factory" => undefined,
    "http-upgrade" => {"enabled" => true},
    "http-upgrade-enabled" => true,
    "sasl-protocol" => "remote",
    "secure-socket-binding" => undefined,
    "security-realm" => "ManagementRealm",
    "server-name" => undefined,
    "socket-binding" => "management-http",
    "ssl-context" => undefined
  }
}
```

JBoss EAP does provide management-http-authentication and management-sasl-authentication in the elytron subsystem for securing the management interfaces as well.

To update JBoss EAP to use the default Elytron components:

1. Set http-authentication-factory to use management-http-authentication:

   ```plaintext
   ```

2. Set sasl-authentication-factory to use management-sasl-authentication:

   ```plaintext
   /core-service=management/management-interface=http-interface:write-attribute(name=http-upgrade.sasl-authentication-factory, value=management-sasl-authentication)
   ```

3. Undefine security-realm:

   ```plaintext
   /core-service=management/management-interface=http-interface:undefined-attribute(name=security-realm)
   ```

4. Reload JBoss EAP for the changes to take affect:

   ```plaintext
   reload
   ```
The management interfaces are now secured using the default components provided by the **elytron** subsystem.

### 3.1.1.1. Default Elytron HTTP Authentication Configuration

When you access the management interface over http, for example when using the web-based management console, JBoss EAP will use the **management-http-authentication** http-authentication-factory.

```yaml
/subsystem=elytron/http-authentication-factory=management-http-authentication:read-resource()
{
  "outcome" => "success",
  "result" => {
    "http-server-mechanism-factory" => "global",
    "mechanism-configurations" => [{
      "mechanism-name" => "DIGEST",
      "mechanism-realm-configurations" => [{"realm-name" => "ManagementRealm"}]
    }],
    "security-domain" => "ManagementDomain"
  }
}
```

The **management-http-authentication** http-authentication-factory, is configured to use the **ManagementDomain** security domain.

```yaml
/subsystem=elytron/security-domain=ManagementDomain:read-resource()
{
  "outcome" => "success",
  "result" => {
    "default-realm" => "ManagementRealm",
    "permission-mapper" => "default-permission-mapper",
    "post-realm-principal-transformer" => undefined,
    "pre-realm-principal-transformer" => undefined,
    "principal-decoder" => undefined,
    "realm-mapper" => undefined,
    "realms" => [
      {
        "realm" => "ManagementRealm",
        "role-decoder" => "groups-to-roles"
      },
      {
        "realm" => "local",
        "role-mapper" => "super-user-mapper"
      }
    ],
    "role-mapper" => undefined,
    "trusted-security-domains" => undefined
  }
}
```

The **ManagementDomain** security domain is backed by the **ManagementRealm** Elytron security realm, which is a properties-based realm.
IMPORTANT

A properties-based realm is only read when the server starts. Any users added after server start, either manually or by using an `add-user` script, will require a server reload. This reload is accomplished by running the `reload` command from the management CLI.

```
reload
```

```
/subsystem=elytron/properties-realm=ManagementRealm:read-resource()
{
  "outcome" => "success",
  "result" => {
    "groups-attribute" => "groups",
    "groups-properties" => {
      "path" => "mgmt-groups.properties",
      "relative-to" => "jboss.server.config.dir"
    },
    "plain-text" => false,
    "users-properties" => {
      "path" => "mgmt-users.properties",
      "relative-to" => "jboss.server.config.dir"
    }
  }
}
```

### 3.1.1.2. Default Elytron Management CLI Authentication

By default, the management CLI (`jboss-cli.sh`) is configured to connect over `remote+http`.

**Example: Default `jboss-cli.xml`**

```
<jboss xmlns="urn:jboss:cli:3.1">
  <default-protocol use-legacy-override="true">remote+http</default-protocol>

  <!-- The default controller to connect to when 'connect' command is executed w/o arguments -->
  <default-controller>
    <protocol>remote+http</protocol>
    <host>localhost</host>
    <port>9990</port>
  </default-controller>
</jboss>
```

This will establish a connection over HTTP and use HTTP upgrade to change the communication protocol to **Remoting**. The HTTP upgrade connection is secured in the `http-upgrade` section of the `http-interface` using a `sasl-authentication-factory`.

**Example: Configuration with Default Components**

```
/core-service=management/management-interface=http-interface:read-resource()
{
  "outcome" => "success",
  "result" => {
    "allowed-origins" => undefined,
  }
}
```
The default sasl-authentication-factory is management-sasl-authentication.

The management-sasl-authentication sasl-authentication-factory specifies JBOSS-LOCAL-USER and DIGEST-MD5 mechanisms.

The ManagementRealm Elytron security realm, used in DIGEST-MD5, is the same realm used in the management-http-authentication http-authentication-factory.

Example: JBOSS-LOCAL-USER Realm
The local Elytron security realm is for handling silent authentication for local users.

### 3.1.2. Secure the Management Interfaces with a New Identity Store

1. Create a security domain and any supporting security realms, decoders, or mappers for your identity store.
   
   This process is covered in the Elytron Subsystem section of JBoss EAP How to Configure Identity Management Guide. For example, if you wanted to secure the management interfaces using a filesystem-based identity store, you would follow the steps in Configure Authentication with a Filesystem-based Identity Store.

2. Create an http-authentication-factory or sasl-authentication-factory.

   **Example: http-authentication-factory**
   
   ```java
   /subsystem=elytron/http-authentication-factory=example-http-auth:add(http-server-mechanism-factory=global, security-domain=exampleSD, mechanism-configurations=[{mechanism-name=DIGEST, mechanism-realm-configurations=[{realm-name=exampleManagementRealm}]}])
   ```

   **Example: sasl-authentication-factory**
   
   ```java
   /subsystem=elytron/sasl-authentication-factory=example-sasl-auth:add(sasl-server-factory=configured, security-domain=exampleSD, mechanism-configurations=[{mechanism-name=DIGEST-MD5, mechanism-realm-configurations=[{realm-name=exampleManagementRealm}]}])
   ```

3. Add pattern-filter to the configured configurable-sasl-server-factory.

   **Example: Add GSSAPI to the Configured configurable-sasl-server-factory**
   
   ```java
   /subsystem=elytron/configurable-sasl-server-factory=configured:list-add(name=filters, value={pattern-filter=GSSAPI})
   ```

   This is an optional step. When a client attempts to connect to the HTTP management interfaces, JBoss EAP sends back an HTTP response with a status code of **401 Unauthorized**, and a set of headers that list the supported authentication mechanisms, for example, Digest, GSSAPI, and so on. For more information, see the Local and Remote Client Authentication with the HTTP Interface section in the JBoss EAP Security Architecture guide.

4. Update the management interfaces to use your http-authentication-factory or sasl-authentication-factory.

   **Example: Update http-authentication-factory**
   
   ```java
   /core-service=management/management-interface=http-interface:write-attribute(name=http-authentication-factory, value=example-http-auth)
   ```

   **Example: Update sasl-authentication-factory**
   
   ```java
   /core-service=management/management-interface=http-interface:write-attribute(name=http-
When using legacy core management authentication, you can only secure the http management interface with a single legacy security realm. This forces the HTTP and SASL configuration to appear in a single legacy security realm. When using the elytron subsystem, you can configure the http-authentication-factory and sasl-authentication-factory separately, allowing you to use distinct security domains for securing the HTTP and SASL mechanisms of the http management interface.

NOTE

If two different attributes with similar implementation in legacy security and Elytron, respectively, are configured in the management interface, only the Elytron related configurations are used. For example, if security-realm for legacy security and http-authentication-factory for Elytron are configured, then authentication is handled by http-authentication-factory configuration.

NOTE

When the management interface includes both http-authentication-factory, or sasl-authentication-factory for the HTTP interface, as well as the security-realm, and the ssl-context attribute is not used, the authentication is handled by Elytron and the SSL is handled by the legacy security realm.

When the management interface includes both the security-realm and the ssl-context, and the http-authentication-factory or sasl-authentication-factory for the HTTP interface is not used, then authentication is handled by the legacy security realm and SSL is handled by Elytron.

3.1.3. Adding Silent Authentication

By default, JBoss EAP provides an authentication mechanism for local users, also know as silent authentication, through the local security realm. You can find more details see Silent authentication section.

Silent authentication must be added to a sasl-authentication-factory.

To add silent authentication to an existing sasl-authentication-factory:

```
/subsystem=elytron/sasl-authentication-factory=example-sasl-auth:list-add(name=mechanism-configurations, value={mechanism-name=JBOSS-LOCAL-USER, realm-mapper=local})
reload
```

To create a new sasl-server-factory with silent authentication:

```
/subsystem=elytron/sasl-authentication-factory=example-sasl-auth:add(sasl-server-factory=configured,security-domain=ManagementDomain,mechanism-configurations=[[mechanism-name=DIGEST-MD5,mechanism-realm-configurations=[[realm-name=exampleManagementRealm]]]],
```
NOTE

The above example uses the existing ManagementDomain security domain, but you can also create and use other security domains. You can find more examples of creating security domains in the Elytron Subsystem section of the JBoss EAP How to Configure Identity Management Guide.

IMPORTANT

If the Elytron security is used and an authentication attempt comes in using the JBOSS-LOCAL-USER SASL mechanism with an authentication name that does not correspond to a real identity, authentication fails.

Choosing a custom user name for JBOSS-LOCAL-USER is possible with legacy security subsystem. There the authentication proceeds by mapping the user name to a special identity.

3.1.4. Mapping Identity for Authenticated Management Users

When using the elytron subsystem to secure the management interfaces, you can provide a security domain to the management interfaces for identity mapping of authenticated users. This allows authenticated users to appear with the appropriate identity when logged into the management interfaces.

The application server exposes more than one kind of management interface. Each type of interface can be associated with an independent authentication-factory to handle the authentication requirements of that interface.

To make the authorization decision, the current security identity is obtained from the security domain. The returned security identity has the role mapping and permission assignment, based on the rules defined within that security domain.

NOTE

In most cases, a common security domain is used for all management; for authentication of the management interfaces as well as for obtaining the security identity used for the authorization decisions. In these cases, the security domain is associated with the authentication factory of the management interface and no special access=identity needs to be defined.

In some cases, a different security domain is used to obtain the identity for the authorization decisions. Here, the access=identity resource is defined. It contains a reference to a security domain to obtain the identity for authorization.

The below example assumes you have secured the management interfaces with the exampleSD Elytron security domain and have it exposed as exampleManagementRealm.

To define the identity mapping, add the identity resource to the management interfaces.

Example: Add the identity Resource
Once you have added the identity resource, the identity of an authenticated user will appear when accessing the management interfaces. When the identity resource is not added, then the identity of the security domain used for authentication is used.

For example, if you logged into the management CLI as user1, your identity will properly appear.

Example: Display the Identity of an Authenticated User from the Management CLI

```
:whoami
{
  "outcome" => "success",
  "result" => {"identity" => {"username" => "user1"}}
}
```

**IMPORTANT**

If the identity resource is added and legacy security realms are used to secure the management interfaces, authenticated users will always have the anonymous identity. Once the identity resource is removed, users authenticated from the legacy security realms will appear with the appropriate identity.

Authorization for management operation always uses the security domain, which is the domain specified on access=identity. If not specified, it is the domain used for authentication. Any role mapping is always in the context of the security domain.

The identity resource for the current request will return a set of roles as mapped using the Elytron configuration. When an RBAC based role mapping definition is in use, the roles from the identity resource will be taken as groups and fed into the management RoleMapping to obtain the management roles for the current request.

**Table 3.1. Identity to be Used for Different Scenarios**

<table>
<thead>
<tr>
<th>Scenario</th>
<th>No access=identity definition</th>
<th>access=identity referencing an Elytron security-domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP management interface using legacy security-realm</td>
<td>Identity from connection.</td>
<td>Unsupported or anonymous identity.</td>
</tr>
<tr>
<td>HTTP management interface using elytron HTTP authentication factory backed by security-domain</td>
<td>Identity from connection.</td>
<td>Identity from referenced security-domain if it was successfully inflowed.</td>
</tr>
<tr>
<td>Native management, including over HTTP Upgrade, interface using legacy security-realm</td>
<td>Identity from connection.</td>
<td>Unsupported or anonymous identity.</td>
</tr>
</tbody>
</table>
Native management, including over HTTP Upgrade, interface using elytron SASL authentication factory backed by security-domain

<table>
<thead>
<tr>
<th>Scenario</th>
<th>No access=identity definition</th>
<th>access=identity referencing an Elytron security-domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native management, including over HTTP Upgrade, interface using elytron SASL authentication factory backed by security-domain</td>
<td>Identity from connection.</td>
<td>Identity from referenced security-domain if it was successfully inflowed.</td>
</tr>
</tbody>
</table>

**NOTE**

If security domain used in the identity resource does not trust the security domain from authentication, anonymous identity is used.

The security domain used in the identity resource does not need to trust the security domain from authentication, when both are using an identical security realm.

The trusted security domains is not transitive.

Where no access=identity resource is defined, then the identity established during authentication against the management interface will be used. Identities established using connections, through the remoting subsystem or using applications, will not be usable in this case.

Where an access=identity resource is defined but the security domain used by the management interfaces is different and not listed in the list of domains to inflow from, no identity will be established. An inflow will be attempted using the identity established during authentication. Identities established using connections through the remoting subsystem or using applications will not be inflowed in this way.

**IMPORTANT**

Where the management interfaces are secured using the legacy security realms, the identity will not be sharable across different security domains. In that case no access=identity resource should be defined. So the identity established during authentication can be used directly. Thus, applications secured using PicketBox are not supported for the identity resource.

### 3.1.5. Using Elytron Client with the Management CLI

You can configure the management CLI to use Elytron Client for providing security information when connecting to JBoss EAP.

1. Secure the management interfaces with Elytron.
   In order to use Elytron Client with the management CLI, you must secure the management interfaces with Elytron. You can find more details on securing the management interfaces with Elytron in [User Authentication with Elytron](#).

2. Create an Elytron Client configuration file.
   You need to create an Elytron Client configuration file that houses your authentication configuration as well as rules for using that configuration. You can find more details on creating an authentication configuration in the [The Configuration File Approach](#) section of the JBoss EAP How to Configure Identity Management Guide.

Example: custom-config.xml
3. Use the Elytron Client configuration file with management CLI script.

```bash
$ ./jboss-cli.sh -c -Dwildfly.config.url=/path/to/custom-config.xml
```

### 3.2. Identity Propagation and Forwarding with Elytron

#### 3.2.1. Propagating Security Identities for Remote Calls

JBoss EAP 7.1 introduced the ability to easily configure the server and your applications to propagate a security identity from a client to the server for remoting calls. You can also configure server components to run within the security identity of a given user.

The example in this section demonstrates how to forward security identity credentials. It propagates the security identity of a client and an Jakarta Enterprise Beans to a remote Jakarta Enterprise Beans. It returns a string containing the name of the Principal that called the remote Jakarta Enterprise Beans along with the user’s authorized role information. The example consists of the following components.

- A secured Jakarta Enterprise Beans that contains a single method, accessible by all users, that returns authorization information about the caller.
- An intermediate Jakarta Enterprise Beans that contains a single method. It makes use of a remote connection and invokes the method on the secured Jakarta Enterprise Beans.
- A remote standalone client application that invokes the intermediate Jakarta Enterprise Beans.

- A `META-INF/wildfly-config.xml` file that contains the identity information used for authentication.

You must first enable security identity propagation by configuring the server. Next review the example application code that uses the `WildFlyInitialContextFactory` to look up and invoke the remote Jakarta Enterprise Beans.
Configure the Server for Security Propagation

1. Configure the **ejb3** subsystem to use the Elytron **ApplicationDomain**.

   ```
   /subsystem=ejb3/application-security-domain=quickstart-domain:add(security-domain=ApplicationDomain)
   ```

   This adds the following **application-security-domain** configuration to the **ejb3** subsystem.

   ```
   <subsystem xmlns="urn:jboss:domain:ejb3:5.0">
      ....
      <application-security-domains>
         <application-security-domain name="quickstart-domain" security-domain="ApplicationDomain"/>
      </application-security-domains>
   </subsystem>
   ```

2. Add the **PLAIN** authentication configuration to send plain text user names and passwords, and the authentication context that is to be used for outbound connections. See [Mechanisms That Support Security Identity Propagation](#) for the list of mechanisms that support identity propagation.

   ```
   /subsystem=elytron/authentication-configuration=ejb-outbound-configuration:add(security-domain=ApplicationDomain,sasl-mechanism-selector="PLAIN")
   /subsystem=elytron/authentication-context=ejb-outbound-context:add(match-rules=[{authentication-configuration=ejb-outbound-configuration}])
   ```

   This adds the following **authentication-client** configuration to the **elytron** subsystem.

   ```
   <subsystem xmlns="urn:wildfly:elytron:4.0" final-providers="combined-providers" disallowed-providers="OracleUcrypto">
      <authentication-client>
         <authentication-configuration name="ejb-outbound-configuration" security-domain="ApplicationDomain" sasl-mechanism-selector="PLAIN"/>
         <authentication-context name="ejb-outbound-context">
            <match-rule authentication-configuration="ejb-outbound-configuration"/>
         </authentication-context>
      </authentication-client>
      ....
   </subsystem>
   ```

3. Add the remote destination outbound socket binding to the **standard-sockets** socket binding group.

   ```
   /socket-binding-group=standard-sockets/remote-destination-outbound-socket-binding=ejb-outbound:add(host=localhost,port=8080)
   ```

   This adds the following **ejb-outbound** outbound socket binding to the **standard-sockets** socket binding group.

   ```
   <socket-binding-group name="standard-sockets" default-interface="public" port-offset="$[jboss.socket.binding.port-offset:0]">
      ....
      <outbound-socket-binding name="ejb-outbound">
      ....
   </socket-binding-group>
   ```
4. Add the remote outbound connection and set the SASL authentication factory in the HTTP connector.

```xml
/subsystem=remoting/remote-outbound-connection=ejb-outbound-connection:add(outbound-socket-binding-ref=ejb-outbound, authentication-context=ejb-outbound-context)
/subsystem=remoting/http-connector=http-remoting-connector:write-attribute(name=sasl-authentication-factory,value=application-sasl-authentication)
```

This adds the following `http-remoting-connector` and `ejb-outbound-connection` configuration to the `remoting` subsystem.

5. Configure the Elytron SASL authentication to use the `PLAIN` mechanism.

```xml
/subsystem=elytron/sasl-authentication-factory=application-sasl-authentication:write-attribute(name=mechanism-configurations,value=[{mechanism-name=PLAIN},{mechanism-name=JBOSS-LOCAL-USER,realm-mapper=local},{mechanism-name=DIGEST-MD5,mechanism-realm-configurations=[{realm-name=ApplicationRealm}]}])
```

This adds the following `application-sasl-authentication` configuration to the `elytron` subsystem.
The server is now configured to enable security propagation for the following example application.

**Review the Example Application Code That Propagates a Security Identity**

Once security identity propagation is enabled in the server configuration, the Jakarta Enterprise Beans client application can use the `WildFlyInitialContextFactory` to look up and invoke the Jakarta Enterprise Beans proxy. The Jakarta Enterprise Beans is invoked as the user that authenticated in the client example shown below. The following abbreviated code examples are taken from the `ejb-security-context-propagation` quickstart that ships with JBoss EAP 7.4. See that quickstart for a complete working example of security identity propagation.

To invoke the Jakarta Enterprise Beans as a different user, you can set the `Context.SECURITY_PRINCIPAL` and `Context.SECURITY_CREDENTIALS` in the context properties.

**Example: Remote Client**

```java
public class RemoteClient {

    public static void main(String[] args) throws Exception {
        // invoke the intermediate bean using the identity configured in wildfly-config.xml
        invokeIntermediateBean();

        // now lets programmatically setup an authentication context to switch users before invoking the intermediate bean
        AuthenticationConfiguration superUser = AuthenticationConfiguration.empty().setSaslMechanismSelector(SaslMechanismSelector.NONE.addMechanism("PLAIN"));
        superUser.useName("superUser").usePassword("superPwd1!");
        final AuthenticationContext authCtx = AuthenticationContext.empty().with(MatchRule.ALL, superUser);

        AuthenticationContext.getContextManager().setThreadDefault(authCtx);
        invokeIntermediateBean();
    }

    private static void invokeIntermediateBean() throws Exception {
        final Hashtable<String, String> jndiProperties = new Hashtable<>();
        jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY, "org.wildfly.naming.client.WildFlyInitialContextFactory");
        jndiProperties.put(Context.PROVIDER_URL, "remote+http://localhost:8080");
        final Context context = new InitialContext(jndiProperties);
        IntermediateEJBRemote intermediate = (IntermediateEJBRemote) context.lookup("ejb:/ejb-security-context-propagation/IntermediateEJB!" + IntermediateEJBRemote.class.getName());
        // Call the intermediate EJB
        System.out.println(intermediate.makeRemoteCalls());
    }
}
```

**Example: Intermediate Jakarta Enterprise Beans**

```java
@Stateless
@Remote(IntermediateEJBRemote.class)
@SecurityDomain("quickstart-domain")
@PermitAll
public class IntermediateEJB implements IntermediateEJBRemote {
```
Example: Secured Jakarta Enterprise Beans

```java
@Stateless
@Remote(SecuredEJBRemote.class)
@SecurityDomain("quickstart-domain")
public class SecuredEJB implements SecuredEJBRemote {

    @Resource
    private SessionContext context;

    @PermitAll
    public String getSecurityInformation() {
        StringBuilder sb = new StringBuilder("[");
        sb.append("Principal=[").append(context.getCallerPrincipal().getName()).append("]");
        userInRole("guest", sb).append(",");
        userInRole("user", sb).append(",");
        userInRole("admin", sb).append("]");
        return sb.toString();
    }
}
```

Example: wildfly-config.xml File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
```
3.2.2. Utilizing Authorization Forwarding Mode

In addition to credential forwarding, Elytron supports the trusted use of identities between peers. This can be useful in the following cases.

- Requirements are such that you cannot send passwords over the wire.
- The authentication type is one that does not support credential forwarding.
- The environment requires a need to limit which systems are allowed to receive the propagated requests.

To utilize authorization forwarding, you first configure an authentication client on the forwarding server and then configure the receiving server to accept and handle the authorization.

**Configure the Authentication Client on the Forwarding Server**

To enable authorization forwarding, you must configure an authentication client configuration in the forwarding server configuration.

The following management CLI commands create a default authentication client configuration to enable authentication forwarding. You can configure a more advanced rule based selection if you need one.

**Example: Management CLI Command to Create the Authentication Client Configuration**

```
$subsystem=elytron/authentication-configuration=forwardit:add(authentication-name=theserver1,security-domain=ApplicationDomain,realm=ApplicationRealm,forwarding-mode=authorization,credential-reference={clear-text=thereallysecretpassword})
$subsystem=elytron/authentication-context=forwardctx:add(match-rules=[[authentication-configuration=forwardit,match-no-user=true]])
```

These commands add the following `authentication-configuration` and `authentication-context` configuration to the `elytron` subsystem.

**Example: Authentication Client Configuration**

```
<authentication-client xmlns="urn:elytron:client:1.2">
  <authentication-rules>
    <rule use-configuration="default"/>
  </authentication-rules>
  <authentication-configurations>
    <configuration name="default">
      <set-user-name name="quickstartUser"/>
      <credentials>
        <clear-password password="quickstartPwd1!"/>
      </credentials>
      <sasl-mechanism-selector selector="PLAIN"/>
      <providers>
        <use-service-loader/>
      </providers>
    </configuration>
  </authentication-configurations>
</authentication-client>
```
When the forwarding server contacts the receiving server, instead of using the default authentication-based user name and credentials, it uses the predefined server login name `theserver1` to establish the trust relationship.

Configure the Authorization Forwarding on the Receiving Server

For the forwarding to complete successfully, the receiving server configuration needs to be configured with the identity matching the one passed by the forwarding server. In this case, you must configure a user named `theserver1` on the receiving server with the correct credentials.

You must also configure a "RunAs" permission mapping in the `elytron` subsystem to allow the identity switch for the `theserver1` identity that is passed from the forwarding server. For more information about permission mapping, see Create an Elytron Permission Mapper in How to Configure Server Security for JBoss EAP.

The command below adds a `simple-permission-mapper` named `auth-forwarding-permission-mapper` that includes the following configurations.

- A permission mapping for the user `anonymous`. This user has no permissions, which prevents an anonymous user from being able to log in.

- A permission mapping for the user `theserver1`. This user is assigned the `RunAsPrincipalPermission` permission of `*`, which gives this user global permissions to run as any identity. You can restrict the permission to a specific identity if you prefer.

- A permission mapping for all other users.

Example: Management CLI Command to the Create Simple Permission Mapper

```
<authentication-client>
  <authentication-configuration name="forwardit" authentication-name="theserver1" security-domain="ApplicationDomain" forwarding-mode="authorization" realm="ApplicationRealm">
    <credential-reference clear-text="thereallysecretpassword"/>
  </authentication-configuration>
  <authentication-context name="forwardctx">
    <match-rule match-no-user="true" authentication-configuration="forwardit"/>
  </authentication-context>
</authentication-client>
```

Example: Simple Permission Mapper Configuration

```
<mappers>
  <simple-permission-mapper name="auth-forwarding-permission-mapper">
    <permission-mapping>
      <principal name="anonymous"/>
    </permission-mapping>
  </simple-permission-mapper>
</mappers>
```
The login-permission and default-permissions permission sets are already present in the default configuration.

In cases where principal transformers are used after forwarding authorization, then those transformers are applied on both the authentication and the authorization principals.

3.2.3. Creating a case-principal-transformer to change the case characters of your principal username

The elytron subsystem includes the case-principal-transformer principal transformer. You can use this principal transformer to change a principal’s username to either uppercase or lowercase characters.

The case-principal-transformer principal transformer includes the upper-case attribute that is set as true by default.

To demonstrate a use case for case-principal-transformer, consider that you are using an authentication mechanism to map a principal to a security realm. A realm mapper manipulates the mapped principal to identify a security realm and load one of its identities. The authentication mechanism passes the identity to a post-realm mapping stage and to a final principal transformation
stage. Subsequently, the authentication mechanism verifies the identity for authentication purposes. You can use a **case-principal-transformer** principal transformer to convert the character case format of your mapped principal.

The example in the procedure uses the **case-principal-transformer** in the context of a security domain. You can also use the principal transformer inline with the following authentication policies:

- **http-authentication-factory**
- **sasl-authentication-factory**
- **ssl-context**
- **aggregate-realm**

**Procedure**

1. Add the **case-principal-transformer** to the **elytron** subsystem, and choose the character case for the username.
   
   a. To change the username of a transformer to uppercase characters, do not change the default **upper-case** attribute value.

   **Example showing <transformer_name> added to the elytron subsystem with the default upper-case attribute setting defined:**
   
   ```
   /subsystem=elytron/case-principal-transformer=<transformer_name>:add(upper-case="true")
   ```

   Alternatively, you can truncate the command syntax to use the default **upper-case** attribute value:

   ```
   /subsystem=elytron/case-principal-transformer=<transformer_name>:add()
   ```

   b. To change the username of a transformer to lowercase characters, set the **upper-case** attribute to **false**.

   **Example showing <transformer_name> added to the elytron subsystem with the upper-case attribute set to false:**

   ```
   /subsystem=elytron/case-principal-transformer=<transformer_name>:add(upper-case="false")
   ```

2. **Optional:** Use the **elytron** subsystem to configure your principal transformer. The following example configured a principal transformer to the default **ApplicationDomain** configuration that was provided by the **elytron** subsystem. Elytron applies the default **ApplicationDomain** configuration to a **pre-realm-principal-transformer**:

   ```
   /subsystem=elytron/security-domain=ApplicationDomain:write-attribute(name=pre-realm-principal-transformer,value=<transformer_name>)
   ```
NOTE

You can configure a post-realm-principal-transformer to use the ApplicationDomain configuration after a security realm has been identified by a realm mapper.

Additional resources

- For information about the upper-case attribute, see Table A.26 case-principal-transformer attributes.

3.2.4. Retrieving Security Identity Credentials

There might be situations where you need to retrieve identity credentials for use in outgoing calls, for example, by an HTTP client. The following example demonstrates how to retrieve security credentials programmatically.

```java
import org.wildfly.security.auth.server.IdentityCredentials;
import org.wildfly.security.auth.server.SecurityDomain;
import org.wildfly.security.auth.server.SecurityIdentity;
import org.wildfly.security.credential.PasswordCredential;
import org.wildfly.security.password.interfaces.ClearPassword;

SecurityIdentity securityIdentity = null;
ClearPassword password = null;

// Obtain the SecurityDomain for the current deployment.
// The calling code requires the
// org.wildfly.security.permission.ElytronPermission("getSecurityDomain") permission
// if running with a security manager.
SecurityDomain securityDomain = SecurityDomain.getCurrent();
if (securityDomain != null) {
    // Obtain the current security identity from the security domain.
    // This always returns an identity, but it could be the representation
    // of the anonymous identity if no authenticated identity is available.
    securityIdentity = securityDomain.getCurrentSecurityIdentity();
    // The private credentials can be accessed to obtain any credentials delegated to the identity.
    // The calling code requires the
    // org.wildfly.security.permission.ElytronPermission("getPrivateCredentials")
    // permission if running with a security manager.
    IdentityCredentials credentials = securityIdentity.getPrivateCredentials();
    if (credentials.contains(PasswordCredential.class)) {
        password = credentials.getCredential(PasswordCredential.class).getPassword(ClearPassword.class);
    }
}
```

3.2.5. Mechanisms That Support Security Identity Propagation

The following SASL mechanisms support propagation of security identities:

- PLAIN
- OAUTHBEARER
The following HTTP mechanisms support propagation of security identities:

- FORM
- BASIC
- BEARER_TOKEN
- SPNEGO

FORM authentication is not automatically handled by the web browser. For this reason, you cannot use identity propagation with web applications that use FORM authentication when running in an HA cluster. Other mechanisms, such as BASIC and SPNEGO, support identity propagation in an HA cluster environment.

### 3.3. IDENTITY SWITCHING WITH ELYTRON

#### 3.3.1. Switching Identities in Server-to-server Jakarta Enterprise Beans Calls

By default, when you make a remote call to a Jakarta Enterprise Beans deployed to an application server, the identity used for authentication on the remote server is the same one that was used on the source server. In some cases, you might want to run the remote secured Jakarta Enterprise Beans within the security context of a different identity.

You can use the Elytron API to switch identities in server-to-server Jakarta Enterprise Beans calls. When you do that, the request received over the connection is executed as a new request, using the identity specified programmatically in the API call.

The following code example demonstrates how to switch the identity that is used for authentication on a remote Jakarta Enterprise Beans. The remoteUsername and remotePassword arguments passed in the `securityDomain.authenticate()` method are the identity credentials that are to be used for authentication on the target server.

**Example: Switching Identities in Server-to-server Jakarta Enterprise Beans Calls**

```java
SecurityDomain securityDomain = SecurityDomain.getCurrent();
Callable<T> forwardIdentityCallable = () -> {
    return AuthenticationContext.empty()
        .with(MatchRule.ALL,
            AuthenticationConfiguration.empty()
                .setSaslMechanismSelector(SaslMechanismSelector.ALL)
                .useForwardedIdentity(securityDomain))
        .runCallable(callable);
};

securityDomain.authenticate(remoteUsername, new PasswordGuessEvidence(remotePassword.toCharArray())).runAs(forwardIdentityCallable);
```
3.4. USER AUTHENTICATION WITH LEGACY CORE MANAGEMENT AUTHENTICATION

3.4.1. Default User Configuration

All management interfaces in JBoss EAP are secured by default and users can access them in two different ways: local interfaces and remote interfaces. The basics of both of these authentication mechanisms are covered in the Default Security and JBoss EAP Out of the Box sections of the JBoss EAP Security Architecture guide. By default, access to these interfaces is configured in the Management Realm security realm. Initially, the local interface is enabled and requires access to the host machine running the JBoss EAP instance. Remote access is also enabled and is configured to use a file-based identity store. By default it uses mgmt-users.properties file to store user names and passwords, and mgmt-groups.properties to store user group information.

User information is added to these files by using the included adduser script located in the EAP_HOME/bin/ directory.

To add a user via the adduser script:

1. Run the add-user.sh or add-user.bat command.
2. Choose whether to add a management user or application user.
3. Choose the realm the user will be added to. By default, the only available realms are ManagementRealm and ApplicationRealm. If a custom realm has been added, its name can be manually entered instead.
4. Type the desired user name, password, and optional roles when prompted. The changes are written to each of the properties files for the security realm.

3.4.2. Adding Authentication via LDAP

JBoss EAP also supports using LDAP authentication for securing the management interfaces. The basics of LDAP and how it works with JBoss EAP are covered in the LDAP, Using LDAP with the Management Interfaces, and Using LDAP with the ManagementRealm sections of the Red Hat JBoss Enterprise Application Platform 7 Security Architecture guide. For more specifics on how to secure the management interfaces using LDAP authentication, see the Securing the Management Interfaces with LDAP section of the JBoss EAP How to Configure Identity Management Guide.

3.4.3. Using JAAS for Securing the Management Interfaces

JAAS is a declarative security API used by JBoss EAP to manage security. For more details and background regarding JAAS and declarative security, see the Declarative Security and JAAS section of the Red Hat JBoss Enterprise Application Platform Security Architecture guide.

NOTE

When JBoss EAP instances are configured to run in ADMIN_ONLY mode, using JAAS to secure the management interfaces is not supported. For more information on ADMIN_ONLY mode, see the Running JBoss EAP in ADMIN_ONLY Mode section of the JBoss EAP Configuration Guide.

To use JAAS to authenticate to the management interfaces, the following steps must be performed:
1. Create a security domain.
   In this example, a security domain is created with the **UserRoles** login module, but other login modules may be used as well:

   ```
   /subsystem=security/security-domain=UsersLMDomain:add(cache-type=default)
   /subsystem=security/security-domain=UsersLMDomain/authentication=classic:add
   /subsystem=security/security-domain=UsersLMDomain/authentication=classic/login-module=UsersRoles:add(code=UsersRoles, flag=required,module-options=[("usersProperties">"users.properties"),("rolesProperties">"roles.properties")])
   ```

2. Create a security realm with JAAS authentication.

   ```
   /core-service=management/security-realm=SecurityDomainAuthnRealm:add
   /core-service=management/security-realm=SecurityDomainAuthnRealm/authentication=jaas:add(name=UsersLMDomain)
   ```

3. Update the **http-interface** management interface to use new security realm.

   ```
   /core-service=management/management-interface=http-interface/:write-attribute(name=security-realm,value=SecurityDomainAuthnRealm)
   ```

4. **Optional:** Assign group membership.
   The attribute **assign-groups** determines whether loaded user membership information from the security domain is used for group assignment in the security realm. When set to **true**, this group assignment is used for Role-Based Access Control (RBAC).

   ```
   /core-service=management/security-realm=SecurityDomainAuthnRealm/authentication=jaas:write-attribute(name=assign-groups,value=true)
   ```

### 3.5. ROLE-BASED ACCESS CONTROL

The basics of Role-Based Access Control are covered in the **Role-Based Access Control** and **Adding RBAC to the Management Interfaces** sections of the JBoss EAP Security Architecture guide.

#### 3.5.1. Enabling Role-Based Access Control

By default the Role-Based Access Control (RBAC) system is disabled. It is enabled by changing the **provider** attribute from **simple** to **rbac**. **provider** is an attribute of the **access-control** element of the **management** element. This can be done using the management CLI or by editing the server configuration XML file if the server is offline. When RBAC is disabled or enabled on a running server, the server configuration must be reloaded before it takes effect.
WARNING

Before changing the provider to rbac, be sure your configuration has a user who will be mapped to one of the RBAC roles, preferably with at least one in the Administrator or SuperUser role. Otherwise your installation will not be manageable except by shutting it down and editing the XML configuration. If you have started with one of the standard XML configurations shipped with JBoss EAP, the $local user will be mapped to the SuperUser role and the local authentication scheme will be enabled. This will allow a user, running the CLI on the same system as the JBoss EAP process, to have full administrative permissions. Remote CLI users and web-based management console users will have no permissions.

It is recommended to map at least one user, besides $local, before switching the provider to rbac. You can do all of the configuration associated with the rbac provider even when the provider is set to simple.

Once enabled it can only be disabled by a user of the Administrator or SuperUser roles. By default the management CLI runs as the SuperUser role if it is run on the same machine as the server.

CLI to Enable RBAC

To enable RBAC with the management CLI, use the write-attribute operation of the access authorization resource to set the provider attribute to rbac.

```
/core-service=management/access=authorization:write-attribute(name=provider, value=rbac)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

reload

In a managed domain, the access control configuration is part of the domain wide configuration, so the resource address is the same as above, but the management CLI is connected to the master domain controller.

```
/core-service=management/access=authorization:write-attribute(name=provider, value=rbac)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    },
    "result" => undefined,
    "server-groups" => {"main-server-group" => {"host" => {"master" => {"server-one" => {"response" => {
            "outcome" => "success",
            "response-headers" => {
                "operation-requires-reload" => true,
            }
        }
    }
```
"process-state" => "reload-required"
}
"server-two" => {
"response" => {
"outcome" => "success",
"response-headers" => {
"operation-requires-reload" => true,
"process-state" => "reload-required"
}
}
}}}}
}
reload --host=master

NOTE

As with a standalone server, a reload or restart is required for the change to take effect. In a managed domain, all hosts and servers in the domain will need to be reloaded or restarted, starting with the master domain controller.

Management CLI Command to Disable RBAC

To disable RBAC with the management CLI, use the write-attribute operation of the access authorization resource to set the provider attribute to simple.

/core-service=management/access=authorization:write-attribute(name=provider, value=simple)

XML Configuration to Enable or Disable RBAC

If the server is offline the XML configuration can be edited to enable or disable RBAC. To do this, edit the provider attribute of the access-control element of the management element. Set the value to rbac to enable, and simple to disable.

Example: XML Configuration to Enable or Disable RBAC

```xml
<management>
    <access-control provider="rbac">
        <role-mapping>
            <role name="SuperUser">
                <include>
                    <user name="$local"/>
                </include>
            </role>
        </role-mapping>
    </access-control>
</management>
```

3.5.2. Changing the Permission Combination Policy

The Permission Combination Policy determines how permissions are determined if a user is assigned more than one role. This can be set to permissive or rejecting. The default is permissive.

When set to permissive, if any role is assigned to the user that permits an action, then the action is allowed.
When set to **rejecting**, if multiple roles are assigned to a user, then no action is allowed. This means that when the policy is set to rejecting each user should only be assigned one role. Users with multiple roles will not be able to use the management console or the management CLI when the policy is set to rejecting.

The Permission Combination Policy is configured by setting the `permission-combination-policy` attribute to either **permissive** or **rejecting**. This can be done using the management CLI or by editing the server configuration XML file if the server is offline. The `permission-combination-policy` attribute is part of the `access-control` element and the `access-control` element can be found in the `management` element.

**Setting the Permission Combination Policy**

Use the write-attribute operation of the access authorization resource to set the permission-combination-policy attribute to the required policy name.

```xml
/core-service=management/access=authorization:write-attribute(name=permission-combination-policy, value=POLICYNAME)
```

The valid policy names are **rejecting** and **permissive**.

**Example: Management CLI Command for Rejecting Permission Combination Policy**

```xml
/core-service=management/access=authorization:write-attribute(name=permission-combination-policy, value=rejecting)
```

If the server is offline the XML configuration can be edited to change the permission combination policy value. To do this, edit the `permission-combination-policy` attribute of the `access-control` element.

**Example: XML Configuration for Rejecting Permission Combination Policy**

```xml
<access-control provider="rbac" permission-combination-policy="rejecting">
  <role-mapping>
    <role name="SuperUser">
      <include>
        <user name="$local"/>
      </include>
    </role>
  </role-mapping>
</access-control>
```

### 3.5.3. Managing Roles

When Role-Based Access Control (RBAC) is enabled, what a management user is permitted to do is determined by the roles to which the user is assigned. JBoss EAP 7 uses a system of includes and excludes based on both the user and group membership to determine to which role a user belongs.

A user is considered to be assigned to a role if the user is:

- listed as a user to be included in the role, or
- a member of a group that is listed to be included in the role.

A user is also considered to be assigned to a role if the user is not:

- listed as a user to exclude from the role, or
• a member of a group that is listed to be excluded from the role.

Exclusions take priority over inclusions.

Role include and exclude settings for users and groups can be configured using both the management console and the management CLI.

Only users of the SuperUser or Administrator roles can perform this configuration.

3.5.3.1. Configure User Role Assignment Using the Management CLI

The configuration of mapping users and groups to roles is located at: /core-service=management/access=authorization as role-mapping elements.

Only users of the SuperUser or Administrator roles can perform this configuration.

Viewing Role Assignment Configuration

Use the :read-children-names operation to get a complete list of the configured roles:

```
/core-service=management/access=authorization:read-children-names(child-type=role-mapping)
{
    "outcome" => "success",
    "result" => [ 
        "Administrator",
        "Deployer",
        "Maintainer",
        "Monitor",
        "Operator",
        "SuperUser"
    ]
}
```

Use the read-resource operation of a specified role-mapping to get the full details of a specific role:

```
/core-service=management/access=authorization/role-mapping=ROLENAME:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => { 
        "include-all" => false,
        "exclude" => undefined,
        "include" => { 
            "user-theboss" => { 
                "name" => "theboss",
                "realm" => undefined,
                "type" => "USER"
            },
            "user-harold" => { 
                "name" => "harold",
                "realm" => undefined,
                "type" => "USER"
            },
            "group-SysOps" => { 
                "name" => "SysOps",
                "realm" => undefined,
```
Add a New Role
This procedure shows how to add a role-mapping entry for a role. This must be done before the role can be configured.

Use the `add` operation to add a new role configuration.

```
$core-service=management/access=authorization/role-mapping=$ROLENAME:add
```

$ROLENAME is the name of the role that the new mapping is for, such as Auditor.

**Example: Management CLI Command for New Role Configuration**

```
$core-service=management/access=authorization/role-mapping=Auditor:add
```

Add a User as Included in a Role
This procedure shows how to add a user to the included list of a role.

If no configuration for a role has been done, then a role-mapping entry for it must be done first.

Use the `add` operation to add a user entry to the includes list of the role.

```
$core-service=management/access=authorization/role-mapping=$ROLENAME/include=$ALIAS:add(name=$USERNAME, type=USER)
```

- $ROLENAME is the name of the role being configured, such as Auditor.

- $ALIAS is a unique name for this mapping. Red Hat recommends the use of a naming convention for aliases, such as `user-$USERNAME` (for example, `user-max`).

- $USERNAME is the name of the user being added to the include list, such as max.

**Example: Management CLI Command for User Included in a Role**

```
$core-service=management/access=authorization/role-mapping=Auditor/include=user-max:add(name=max, type=USER)
```

Add a User as Excluded in a Role
This procedure shows how to add a user to the excluded list of a role.

If no configuration for a role has been done, then a role-mapping entry for it must be done first.

Use the `add` operation to add a user entry to the excludes list of the role.

```
$core-service=management/access=authorization/role-mapping=$ROLENAME/exclude=$ALIAS:add(name=$USERNAME, type=USER)
```

- $ROLENAME is the name of the role being configured, for example Auditor.
- **USERNAME** is the name of the user being added to the exclude list, for example `max`.
- **ALIAS** is a unique name for this mapping. Red Hat recommends that the use of a naming convention for aliases, such as `user-USERNAME` (for example, `user-max`).

**Example: Management CLI Command User Excluded in a Role**

```
/core-service=management/access=authorization/role-mapping=Auditor/exclude=user-max:add(name=max, type=USER)
```

**Remove User Role Include Configuration**

This procedure shows how to remove a user include entry from a role mapping.

Use the `remove` operation to remove the entry.

```
/core-service=management/access=authorization/role-mapping=ROLENAME/include=ALIAS:remove
```

- **ROLENAME** is the name of the role being configured, such as `Auditor`.
- **ALIAS** is a unique name for this mapping. Red Hat recommends that the use of a naming convention for aliases, such as `user-USERNAME` (for example, `user-max`).

**Example: Management CLI Command for Removing User Role Include Configuration**

```
/core-service=management/access=authorization/role-mapping=Auditor/include=user-max:remove
```

**NOTE**

Removing the user from the list of includes does not remove the user from the system, nor does it guarantee that the role will not be assigned to the user. The role might still be assigned based on group membership.

**Remove User Role Exclude Configuration**

This procedure shows how to remove an user exclude entry from a role mapping.

Use the remove operation to remove the entry.

```
/core-service=management/access=authorization/role-mapping=ROLENAME/exclude=ALIAS:remove
```

- **ROLENAME** is the name of the role being configured, such as `Auditor`.
- **ALIAS** is a unique name for this mapping. Red Hat recommends that the use of a naming convention for aliases, such as `user-USERNAME` (for example, `user-max`).

```
/core-service=management/access=authorization/role-mapping=Auditor/exclude=user-max:remove
```

**NOTE**

Removing the user from the list of excludes does not remove the user from the system, nor does it guarantee the role will be assigned to the user. Roles might still be excluded based on group membership.
3.5.4. Configure User Role Assignment with the Elytron Subsystem

In addition to adding role mappings for users directly, as covered in Managing Roles section, you can also configure RBAC roles to be directly taken from the identity provided by the elytron subsystem.

To configure the RBAC system to use roles provided by the elytron subsystem:

```
/core-service=management/access=authorization:write-attribute(name=use-identity-roles,value=true)
```

**IMPORTANT**

RBAC must be enabled to use this functionality, and the principal must have RBAC roles.

3.5.5. Roles and User Groups

A user group is an arbitrary label that can be assigned to one or more users. When authenticating using the management interfaces, users are assigned groups from either the elytron subsystem or core management authentication, depending on how the management interfaces are secured. The RBAC system can be configured to automatically assign roles to users depending on what user groups they are members of. It can also exclude users from roles based on group membership.

3.5.6. Configure Group Role Assignment Using the Management CLI

Groups to be included or excluded from a role can be configured in the management console and the management CLI. This topic only shows using the management CLI.

The configuration of mapping users and groups to roles is located in the management API at: `/core-service=management/access=authorization` as `role-mapping` elements.

Only users in the **SuperUser** or **Administrator** roles can perform this configuration.

**Viewing Group Role Assignment Configuration**

Use the `read-children-names` operation to get a complete list of the configured roles:

```
/core-service=management/access=authorization:read-children-names(child-type=role-mapping)
{
    "outcome" => "success",
    "result" => [
        "Administrator",
        "Deployer",
        "Maintainer",
        "Monitor",
        "Operator",
        "SuperUser"
    ]
}
```

Use the `read-resource` operation of a specified role-mapping to get the full details of a specific role:

```
/core-service=management/access=authorization/role-mapping=ROLENAME:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
```
Add a New Role
This procedure shows how to add a role-mapping entry for a role. This must be done before the role can be configured.

Use the add operation to add a new role configuration.

```
/core-service=management/access=authorization/role-mapping=ROLENAME:add
```

Add a Group as Included in a Role
This procedure shows how to add a group to the included list of a role.

If no configuration for a role has been done, then a role-mapping entry for it must be done first.

Use the add operation to add a group entry to the includes list of the role.

```
/core-service=management/access=authorization/role-mapping=ROLENAME/include=ALIAS:add(name=GROUPNAME, type=GROUP)
```

- **ROLENAME** is the name of the role being configured, such as Auditor.
- **GROUPNAME** is the name of the group being added to the include list, such as investigators.
- **ALIAS** is a unique name for this mapping. Red Hat recommends that you use a naming convention for your aliases, such as group-GROUPNAME (for example, group-investigators).

Example: Management CLI Command for Adding a Group as Included in a Role

```
/core-service=management/access=authorization/role-mapping=Auditor/include=group-investigators:add(name=investigators, type=GROUP)
```

Add a Group as Excluded in a Role
This procedure shows how to add a group to the excluded list of a role.
If no configuration for a role has been done, then a role-mapping entry for it must be created first.

Use the `add` operation to add a group entry to the excludes list of the role.

```
/core-service=management/access=authorization/role-mapping=ROLENAME/exclude=ALIAS:add(name=GROUPNAME, type=GROUP)
```

- `ROLENAME` is the name of the role being configured, such as `Auditor`.
- `GROUPNAME` is the name of the group being added to the include list, such as `supervisors`.
- `ALIAS` is a unique name for this mapping. Red Hat recommends that you use a naming convention for your aliases, such as `group-GROUPNAME` (for example, `group-supervisors`).

**Example: Management CLI Command for Adding a Group as Excluded in a Role**

```
/core-service=management/access=authorization/role-mapping=Auditor/exclude=group-supervisors:add(name=supervisors, type=GROUP)
```

Remove Group Role Include Configuration

This procedure shows how to remove a group include entry from a role mapping.

Use the `remove` operation to remove the entry.

```
/core-service=management/access=authorization/role-mapping=ROLENAME/include=ALIAS:remove
```

- `ROLENAME` is the name of the role being configured, such as `Auditor`.
- `ALIAS` is a unique name for this mapping. Red Hat recommends that you use a naming convention for your aliases, such as `group-GROUPNAME` (for example, `group-investigators`).

**Example: Management CLI Command for Removing Group Role Include Configuration**

```
/core-service=management/access=authorization/role-mapping=Auditor/include=group-investigators:remove
```

**NOTE**

Removing the group from the list of includes does not remove the group from the system, nor does it guarantee that the role will not be assigned to users in this group. The role might still be assigned to users in the group individually.

Remove a User Group Exclude Entry

This procedure shows how to remove a group exclude entry from a role mapping.

Use the `remove` operation to remove the entry.

```
/core-service=management/access=authorization/role-mapping=ROLENAME/exclude=ALIAS:remove
```

- `ROLENAME` is the name of the role being configured, such as `Auditor`.
ALIAS is a unique name for this mapping. Red Hat recommends that you use a naming convention for your aliases, such as **group-GROUPNAME** (for example, **group-supervisors**).

```text
/core-service=management/access=authorization/role-mapping=Auditor/exclude=group-supervisors:remove
```

**NOTE**

Removing the group from the list of excludes does not remove the group from the system. It also does not guarantee the role will be assigned to members of the group. Roles might still be excluded based on group membership.

### 3.5.7. Using RBAC with LDAP

The basics of using RBAC with LDAP as well as how to configure JBoss EAP to use RBAC with LDAP are covered in the LDAP and RBAC section of the JBoss EAP *How to Configure Identity Management Guide*.

### 3.5.8. Scoped Roles

Scoped roles are user-defined roles that grant the permissions of one of the standard roles but only for one or more specified server groups or hosts in an JBoss EAP managed domain. Scoped roles allow for management users to be granted permissions that are limited to only those server groups or hosts that are required.

**IMPORTANT**

Scoped roles can be created by users assigned the **Administrator** or **SuperUser** roles.

They are defined by five characteristics:

- A unique name.
- The standard roles which it is based on.
- If it applies to server groups or hosts.
- The list of server groups or hosts that it is restricted to.
- If all users are automatically included. This defaults to **false**.

Once created a scoped role can be assigned to users and groups the same way that the standard roles are.

Creating a scoped role does not allow for defining new permissions. Scoped roles can only be used to apply the permissions of an existing role in a limited scope. For example, a scoped role could be created based on the **Deployer** role which is restricted to a single server group.

There are only two scopes that roles can be limited to:

**Host-scoped roles**

A role that is host-scoped restricts the permissions of that role to one or more hosts. This means access is provided to the relevant `/host=*` resource trees but resources that are specific to other hosts are hidden.
Server-group-scoped roles

A role that is server-group-scoped restricts the permissions of that role to one or more server groups. Additionally the role permissions will also apply to the profile, socket binding group, server configuration, and server resources that are associated with the specified server-groups. Any sub-resources within any of those that are not logically related to the server-group will not be visible to the user.

IMPORTANT

Some resources are non-addressable to server-group and host scoped roles in order to provide a simplified view of the management model to improve usability. This is distinct from resources that are non-addressable to protect sensitive data.

For host scoped roles this means that resources in the `/host=*` portion of the management model will not be visible if they are not related to the server groups specified for the role.

For server-group scoped roles, this means that resources in the profile, socket-binding-group, deployment, deployment-overlay, server-group, server-config and server portions of the management model will not be visible if they are not related to the server groups specified for the role.

3.5.8.1. Configuring Scoped Roles from the Management CLI

IMPORTANT

Only users in the SuperUser or Administrator roles can perform this configuration.

Add a New Scoped Role

To add a new scoped role, the following operations must be done:

```
/core-service=management/access=authorization/role-mapping=NEW-SCOPED-ROLE:add
```

```
/core-service=management/access=authorization/server-group-scoped-role=NEW-SCOPED-ROLE:add(base-role=BASE-ROLE, server-groups=[SERVER-GROUP-NAME])
```

Replace NEW-SCOPED-ROLE, BASE-ROLE, and SERVER-GROUP-NAME with the proper information.

Viewing and Editing a Scoped Role Mapping

A scoped role’s details, including members, can be viewed by using the following command:

```
/core-service=management/access=authorization/role-mapping=NEW-SCOPED-ROLE:read-resource(recursive=true)
```

Replace NEW-SCOPED-ROLE with the proper information.

To edit a scoped role’s details, the write-attribute command may be used. For example:

```
/core-service=management/access=authorization/role-mapping=NEW-SCOPED-ROLE:write-attribute(name=include-all, value=true)
```

Replace NEW-SCOPED-ROLE with the proper information.
Delete a Scoped Role

```
/core-service=management/access=authorization/role-mapping=NEW-SCOPED-ROLE:remove
```

```
/core-service=management/access=authorization/server-group-scoped-role=NEW-SCOPED-ROLE:remove
```

Replace *NEW-SCOPED-ROLE* with the proper information.

**IMPORTANT**

A scoped role cannot be deleted if users or groups are assigned to it. Remove the role assignments first, and then delete it.

Adding and Removing Users

Adding and removing users to and from scoped roles follows the same process as adding and removing standard roles.

3.5.8.2. Configuring Scoped Roles from the Management Console

**IMPORTANT**

Only users in the **SuperUser** or **Administrator** roles can perform this configuration.

Scoped role configuration in the management console can be found by following these steps:

1. Log in to the management console.
2. Click on the **Access Control** tab.
3. Click on **Roles** to view all roles, including scoped roles.

The following procedures show how to perform configuration tasks for scoped roles.

Add a New Scoped Role

1. Log in to the management console.
2. Click on the **Access Control** tab.
3. Select **Roles** and click the Add (+) button.
4. Choose **Host Scoped Role** or **Server Group Scoped Role**.
5. Specify the following details:
   - **Name**: The unique name for the new scoped role.
   - **Base Role**: The role which this role will base its permissions on.
   - **Hosts** or **Server Groups**: The list of hosts or server groups that the role is restricted to, depending on the type of scoped role being added. Multiple entries can be selected.
   - **Include All**: Whether this role should automatically include all users. Defaults to **OFF**.
6. Click **Add** to create the new role.

**Edit a Scoped Role**

1. Log in to the management console.
2. Click on the **Access Control** tab.
3. Click on the **Roles** menu on the left.
4. Click on the desired scoped role to edit and click **Edit**.
5. Update the desired details to change and click the **Save** button.

**View Scoped Role Members**

1. Log in to the management console.
2. Click on the **Access Control** tab.
3. Click on the **Roles** menu on the left.
4. Click on the desired scoped role to view the included and excluded members.

**Delete a Scoped Role**

1. Log in to the management console.
2. Click on the **Access Control** tab.
3. Click on the **Roles** menu on the left.
4. Click on the desired scoped role and click **Remove** from the drop down.
5. Click **Yes** to remove the role and all of its assignments.

**Adding and Removing Users**

Adding and removing users to and from scoped roles follows the same process as adding and removing standard roles. To update a user’s scoped roles:

1. Log in to the management console.
2. Click on the **Access Control** tab.
3. Click on the **Roles** menu on the left and click on the desired scoped role.
4. Select the Plus (+) button to include a member or the Minus (-) button to exclude a member.

**3.5.9. Configuring Constraints**

**3.5.9.1. Configure Sensitivity Constraints**

Each sensitivity constraint defines a set of resources that are considered **sensitive**. A **sensitive** resource is generally one that either should be secret, like passwords, or one that will have serious impact on the server, like networking, JVM configuration, or system properties. The access control system itself is also considered sensitive. Resource sensitivity limits which roles are able to read, write or address a specific resource.
Sensitivity constraint configuration is at `/core-service=management/access=authorization/constraint=sensitivity-classification`.

Within the management model each sensitivity constraint is identified as a classification. The classifications are then grouped into types. Each classification has an `applies-to` element which is a list of path patterns to which the classifications configuration applies.

To configure a sensitivity constraint, use the `write-attribute` operation to set the `configured-requires-read`, `configured-requires-write`, or `configured-requires-addressable` attribute. To make that type of operation sensitive set the value of the attribute to `true`, otherwise to make it nonsensitive set it to `false`. By default these attributes are not set and the values of `default-requires-read`, `default-requires-write`, and `default-requires-addressable` are used. Once the configured attribute is set it is that value that is used instead of the default. The default values cannot be changed.

**Example: Make Reading System Properties a Sensitive Operation**

```
/core-service=management/access=authorization/constraint=sensitivity-classification/type=core/classification=system-property:write-attribute(name=configured-requires-read,value=true)
```

**Example: Result**

```
/core-service=management/access=authorization/constraint=sensitivity-classification/type=core/classification=system-property:read-resource
```

```
{    "outcome" => "success",    "result" => {        "configured-requires-addressable" => undefined,        "configured-requires-read" => true,        "configured-requires-write" => undefined,        "default-requires-addressable" => false,        "default-requires-read" => false,        "default-requires-write" => true,        "applies-to" => {            "/core-service=platform-mbean/type=runtime" => undefined,            "/system-property=" => undefined,            "*" => undefined        }    } }
```

The roles, and the respective operations that they are able to perform, depend on the configuration of the attributes. This is summarized in the following table:

**Table 3.2. Sensitivity Constraint Configuration Outcomes**

<table>
<thead>
<tr>
<th>Value</th>
<th>requires-read</th>
<th>requires-write</th>
<th>requires-addressable</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>Read is sensitive. Only Auditor, Administrator, SuperUser can read.</td>
<td>Write is sensitive. Only Administrator and SuperUser can write.</td>
<td>Addressing is sensitive. Only Auditor, Administrator, SuperUser can address.</td>
</tr>
</tbody>
</table>
### 3.5.9.2. List Sensitivity Constraints

You can see a list of the available sensitivity constraints directly from the JBoss EAP management model using the following management CLI command:

```
/core-service=management/access=authorization/constraint=sensitivity-classification:read-resource(include-runtime=true,recursive=true)
```

### 3.5.9.3. Configure Application Resource Constraints

Each application resource constraint defines a set of resources, attributes and operations that are usually associated with the deployment of applications and services. When an application resource constraint is enabled management users of the **Deployer** role are granted access to the resources that it applies to.

Application constraint configuration is at `/core-service=management/access=authorization/constraint=application-classification/`.

Each application resource constraint is identified as a classification. The classifications are then grouped into types. Each classification has an `applies-to` element which is a list of path patterns to which the classifications configuration applies.

By default the only application resource classification that is enabled is core. Core includes deployments, deployment overlays, and the deployment operations.

To enable an application resource, use the `write-attribute` operation to set the `configured-application` attribute of the classification to `true`. To disable an application resource, set this attribute to `false`. By default these attributes are not set and the value of `default-application` attribute is used. The default value cannot be changed.

**Example: Enabling the logger-profile Application Resource Classification**

```
/core-service=management/access=authorization/constraint=application-classification/type=logging/classification=logging-profile:write-attribute(name=configured-application,value=true)
```
IMPORTANT

Application resource constraints apply to all resources that match its configuration. For example, it is not possible to grant a **Deployer** user access to one datasource resource but not another. If this level of separation is required then it is recommended to configure the resources in different server groups and create different scoped **Deployer** roles for each group.

### 3.5.9.4. List Application Resource Constraints

You can see a list of the available application resource constraints directly from the JBoss EAP management model using the following management CLI command:

```
/core-service=management/access=authorization/constraint=application-classification:read-resource(include-runtime=true,recursive=true)
```

### 3.5.9.5. Configure the Vault Expression Constraint

By default, reading and writing vault expressions are sensitive operations. Configuring the vault expression constraint allows either or both of those operations to be set to nonsensitive. Changing this constraint allows a greater number of roles to read and write vault expressions.

The vault expression constraint is found at `/core-service=management/access=authorization/constraint=vault-expression`.

To configure the vault expression constraint, use the `write-attribute` operation to set the attributes of `configured-requires-write` and `configured-requires-read` to `true` or `false`. By default these are not set and the values of `default-requires-read` and `default-requires-write` are used. The default values cannot be changed.

**Example: Making Writing to Vault Expressions a Nonsensitive Operation**

```
/core-service=management/access=authorization/constraint=vault-expression:write-attribute(name=configured-requires-write,value=false)
```

**Example: Result**

```
/core-service=management/access=authorization/constraint=vault-expression:read-resource
```
The roles, and the respective vault expressions that they will be able to read and write, depend on the configuration of the attributes. This is summarized in the following table:

Table 3.3. Vault Expression Constraint Configuration Outcomes

<table>
<thead>
<tr>
<th>Value</th>
<th>requires-read</th>
<th>requires-write</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>Read operation is sensitive. Only Auditor, Administrator, and SuperUser can read.</td>
<td>Write operation is sensitive. Only Administrator and SuperUser can write.</td>
</tr>
<tr>
<td>false</td>
<td>Read operation is not sensitive. All management users can read.</td>
<td>Write operation is not sensitive. Monitor, Administrator, and SuperUser can write. Deployer can also write if the vault expression is in an application resource.</td>
</tr>
</tbody>
</table>
CHAPTER 4. SECURE STORAGE FOR CREDENTIALS

JBoss EAP allows the encryption of sensitive strings outside of configuration files. These strings can be stored in a keystore, and subsequently decrypted for applications and verifications systems. Sensitive strings can be stored in either of the following:

- **Credential Store** - Introduced in JBoss EAP 7.1, a credential store can safely secure sensitive and plain text strings by encrypting them in a storage file. Each JBoss EAP server can contain multiple credential stores.

- **Password Vault** - Primarily used in legacy configurations, a password vault uses a Java Keystore to store sensitive strings outside of the configuration files. Each JBoss EAP server can only contain a single password vault.

All of the configuration files in `EAP_HOME/standalone/configuration/` and `EAP_HOME/domain/configuration/` are world readable by default. It is strongly recommended to not store plaintext passwords in the configuration files, and instead place these credentials in either a credential Store or password vault.

If you decide to place plaintext passwords in the configuration files, then these files should only be accessible by limited users. At a minimum, the user account under which JBoss EAP 7 is running requires read-write access.

4.1. CREDENTIAL STORES IN ELYTRON

4.1.1. Credential stores provided by Elytron

Elytron provides two default credential store types you can use to save your credentials: KeyStoreCredentialStore and PropertiesCredentialStore. You can manage credential stores with the JBoss EAP management CLI, or you can use the WildFly Elytron tool to manage them offline. In addition to the two default store types, you can also create, use, and manage your own custom credential stores.

4.1.1.1. KeyStoreCredentialStore/credential-store

You can store all the Elytron credential types in a KeyStoreCredentialStore. The resource name for KeyStoreCredentialStore in the `elytron` subsystem is `credential-store`. The KeyStoreCredentialStore protects your credentials using the mechanisms provided by the KeyStore implementations in the Java Development Kit (JDK).

Access a KeyStoreCredentialStore in the management CLI as follows:

```
/subsystem=elytron/credential-store
```

**Additional resources**

- Credential types in Elytron
- Credential store operations using the JBoss EAP management CLI
- Credential store operations using the WildFly Elytron tool
- KeyStore Javadoc
- `credential-store` Attributes
4.1.1.2. PropertiesCredentialStore/secret-key-credential-store

To start properly, JBoss EAP requires an initial key to unlock certain secure resources. Use the secret-key-credential-store to provide this master secret key to unlock these necessary server resources. You can also use the PropertiesCredentialStore to store SecretKeyCredential, which supports storing Advanced Encryption Standard (AES) secret keys. Use file system permissions to restrict access to the credential store. Ideally, you should give access only to your application server to restrict access to this credential store. The resource name in the elytron subsystem for PropertiesCredentialStore is secret-key-credential-store, and you can access it in the management CLI as follows:

```
/subsystem=elytron/secret-key-credential-store
```

For information on creating and providing the initial key, see Providing an initial key to JBoss EAP to unlock secured resources. Alternately, you can get the master key or password from an external source. For information about obtaining the password from an external source, see Obtain the password for the credential store from an external source.

Additional resources
- Credential types in Elytron
- Credential store operations using the JBoss EAP management CLI
- Credential store operations using the WildFly Elytron tool
- secret-key-credential-store Attributes

4.1.2. Credential types in Elytron

Elytron provides the following three credential types to suit your various security needs, and you can store these credentials in one of Elytron's credential stores.

**PasswordCredential**

With this credential type, you can securely store plain text, or unencrypted, passwords. For the JBoss EAP resources that require a password, use a reference to the PasswordCredential instead of the plain text password to maintain the secrecy of the password.

**Example of connecting to a database**

```
data-source add ... --user-name=db_user --password=StrongPassword
```

In this example database connection command, you can see the password: **StrongPassword**. This means that others can also see it in the server configuration file.

**Example of connecting to a database using a PasswordCredential**

```
data-source add ... --user-name=db_user --credential-reference= {store=exampleKeyStoreCredentialStore, alias=passwordCredentialAlias}
```

When you use a credential reference instead of a password to connect to a database, others can only see the credential reference in the configuration file, not your password.

**KeyPairCredential**
You can use both Secure Shell (SSH) and Public-Key Cryptography Standards (PKCS) key pairs as KeyPairCredential. A key pair includes both a shared public key and a private key that only a given user knows.

You can manage KeyPairCredential using only the WildFly Elytron tool.

**SecretKeyCredential**

A SecretKeyCredential is an Advanced Encryption Standard (AES) key that you can use to create encrypted expressions in Elytron. For information about encrypted expressions, see [Encrypted expressions in Elytron](#).

**Additional resources**

- [Credential stores provided by Elytron](#)
- [Credential types supported by credential stores](#)
- [Encrypted expressions in Elytron](#)

### 4.1.3. Credential types supported by Elytron credential stores

The following table illustrates which credential type is supported by which credential store:

<table>
<thead>
<tr>
<th>Credential type</th>
<th>KeyStoreCredentialStore/credential-store</th>
<th>PropertiesCredentialStore/secret-key-credential-store</th>
</tr>
</thead>
<tbody>
<tr>
<td>PasswordCredential</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>KeyPairCredential</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SecretKeyCredential</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Additional resources**

- [Credential types in Elytron](#)
- [Credential stores provided by Elytron](#)

### 4.1.4. Credential store operations using the JBoss EAP management CLI

To manage JBoss EAP credentials in a running JBoss EAP server, use the provided management CLI operations. You can manage **PasswordCredential** and **SecretKeyCredential** using the JBoss EAP management CLI.

**NOTE**

You can do these operation only on modifiable credential stores. All credential store types are modifiable by default.

#### 4.1.4.1. Creating a KeyStoreCredentialStore/credential-store for a standalone server
Create a KeyStoreCredentialStore for a JBoss EAP running as a standalone server in any directory on the file system. For security, the directory containing the store should be accessible to only limited users.

Prerequisites

- You have provided at least read/write access to the directory containing the KeyStoreCredentialStore for the user account under which JBoss EAP is running.

NOTE

You cannot have the same name for a credential-store and a secret-key-credential-store because they implement the same Elytron capability: org.wildfly.security.credential-store.

Procedure

- Create a KeyStoreCredentialStore using the following management CLI command:

  Syntax

  ```
  /subsystem=elytron/credential-store=<name_of_credential_store>:add(path="<path_to_store_file>", relative-to=<base_path_to_store_file>, credential-reference={clear-text=<store_password>}, create=true)
  ```

  Example

  ```
  /subsystem=elytron/credential-store=exampleKeyStoreCredentialStore:add(path="exampleKeyStoreCredentialStore.jceks", relative-to=jboss.server.data.dir, credential-reference={clear-text=password}, create=true)
  ```

  

  ```
  {"outcome" => "success"}
  ```

Additional resources

- Credential store operations using the JBoss EAP management CLI
- credential-store Attributes

4.1.4.2. Creating a KeyStoreCredentialStore/credential-store for a managed domain

You can create a KeyStoreCredentialStore in a managed domain, but you must first use the WildFly Elytron tool to prepare your KeyStoreCredentialStore. If you have multiple host controllers in a single managed domain, choose one of the following options:

- Create a KeyStoreCredentialStore in each host controller and add credentials to each KeyStoreCredentialStore.

- Copy a populated KeyStoreCredentialStore from one host controller to all the other host controllers.

- Save your KeyStoreCredentialStore file in your Network File System (NFS), then use that file for all the KeyStoreCredentialStore resources you create.
Alternatively, you can create a KeyStoreCredentialStore file with credentials on a host controller without using the WildFly Elytron tool.

NOTE

You don’t have to define a KeyStoreCredentialStore resource on every server, because every server on the same profile contains your KeyStoreCredentialStore file. You can find the KeyStoreCredentialStore file in the server data directory, relative-to=jboss.server.data.dir.

IMPORTANT

You cannot have the same name for a credential-store and a secret-key-credential-store because they implement the same Elytron capability: org.wildfly.security.credential-store.

The following procedure describes how to use the NFS to provide the KeyStoreCredentialStore file to all host controllers.

Procedure

1. Use the WildFly Elytron tool to create a KeyStoreCredentialStore storage file. For more information on this, see Credential store operations using the WildFly Elytron tool.

2. Distribute the storage file. For example, allocate it to each host controller by using the scp command, or store it in your NFS and use it for all of your KeyStoreCredentialStore resources.

   NOTE

   To maintain consistency, for a KeyStoreCredentialStore file that multiple resources and host controllers use and which you stored in your NFS, you must use the KeyStoreCredentialStore in read-only mode. Additionally, make sure you provide an absolute path for your KeyStoreCredentialStore file.

Syntax

```
/profile=<profile_name>/subsystem=elytron/credential-store=<name_of_credential_store>:add(path=<absolute_path_to_keystore>,credential-reference={clear-text="<store_password>"},create=false,modifiable=false)
```

Example

```
/profile=full-ha/subsystem=elytron/credential-store=exampleCredentialStoreDomain:add(path=/usr/local/etc/example-cred-store.cs,credential-reference={clear-text="password"},create=false,modifiable=false)
```

3. Optional: If you need to define the credential-store resource in a profile, use the storage file to create the resource.

   Syntax
/profile=<profile_name>/subsystem=elytron/credential-store=<name_of_credential_store>:add(path=<path_to_store_file>,credential-reference={clear-text="<store_password>"})

Example

/profile=full-ha/subsystem=elytron/credential-store=exampleCredentialStoreHA:add(path=/usr/local/etc/example-cred-store-ha.cs, credential-reference={clear-text="password"})

4. Optional: Create the KeyStoreCredentialStore resource for a host controller.

Syntax

/host=<host_controller_name>/subsystem=elytron/credential-store=<name_of_credential_store>:add(path=<path_to_store_file>,credential-reference={clear-text="<store_password>"})

Example

/host=master/subsystem=elytron/credential-store=exampleCredentialStoreHost:add(path=/usr/local/etc/example-cred-store-host.cs, credential-reference={clear-text="password"})

Additional resources

- KeyStoreCredentialStore/credential-store
- Credential store operations using the WildFly Elytron tool
- credential-store Attributes

4.1.4.3. Creating a PropertiesCredentialStore/secret-key-credential-store for a standalone server

Create a PropertiesCredentialStore using the management CLI. When you create a PropertiesCredentialStore, JBoss EAP generates a secret key by default. The name of the generated key is key and its size is 256-bit.

Prerequisites

- You have provided at least read/write access to the directory containing the PropertiesCredentialStore for the user account under which JBoss EAP is running.

Procedure

- Use the following command to create a PropertiesCredentialStore using the management CLI:

  Syntax
/subsystem=elytron/secret-key-credential-
store=<name_of_credential_store>:add(path="<path_to_the_credential_store>", relative-
to="<path_to_store_file>")

Example

/subsystem=elytron/secret-key-credential-
store=examplePropertiesCredentialStore:add(path=examplePropertiesCredentialStore.cs,
relative-to=jboss.server.config.dir)
{"outcome" => "success"}

Additional resources
- PropertiesCredentialStore/secret-key-credential-store
- Credential store operations using the JBoss EAP management CLI
- secret-key-credential-store Attributes

4.1.4.4. Adding a PasswordCredential to a KeyStoreCredentialStore/credential-store

Add a plain text password for those resources that require one as a PasswordCredential to the
KeyStoreCredentialStore to hide that password in the configuration file. You can then reference this
stored credential to access those resources, without ever exposing your password.

Prerequisites
- You have created a KeyStoreCredentialStore.
  For information about creating a KeyStoreCredentialStore, see Creating a
  KeyStoreCredentialStore/credential-store for a standalone server.

Procedure
- Add a new PasswordCredential to a KeyStoreCredentialStore:

Syntax

/subsystem=elytron/credential-store=<name_of_credential_store>:add-alias(alias=<alias>,
secret-value=<secret-value>)

Example

/subsystem=elytron/credential-store=exampleKeyStoreCredentialStore:add-
alias(alias=passwordCredentialAlias, secret-value=StrongPassword)
{"outcome" => "success"}

Verification
- Issue the following command to verify that the PasswordCredential was added to the
  KeyStoreCredentialStore:

  Syntax
  -
4.1.4.5. Generating a SecretKeyCredential in a KeyStoreCredentialStore/credential-store

Generate a SecretKeyCredential in a KeyStoreCredentialStore. By default, Elytron creates a 256-bit key. If you want a different size, you can specify either a 128-bit or 192-bit key in the `key-size` attribute.

Prerequisites

- You have created a KeyStoreCredentialStore.
  For information about creating a KeyStoreCredentialStore, see Creating a KeyStoreCredentialStore/credential-store for a standalone server.

Procedure

1. Generate a SecretKeyCredential in a KeyStoreCredentialStore using the following management CLI command:

   **Syntax**

   ```
   /subsystem=elytron/credential-store=<name_of_credential_store>:generate-secret-key(alias=<alias>, key-size=<128_or_192>)
   ```

   **Example**

   ```
   /subsystem=elytron/credential-store=exampleKeyStoreCredentialStore:generate-secret-key(alias=secretKeyCredentialAlias)
   ```

   **Verification**

   - Issue the following command to verify that Elytron stored your SecretKeyCredential in the KeyStoreCredentialStore:

     **Syntax**

     ```
     /subsystem=elytron/credential-store=<credential_store>:read-aliases()
     ```
Example

```json
/subsystem=elytron/credential-store=exampleKeyStoreCredentialStore:read-aliases()
{
    "outcome" => "success",
    "result" => [
        "secretkeycredentialalias"
    ]
}
```

Additional resources

- KeyStoreCredentialStore/credential-store
- Creating an encrypted expression in Elytron
- credential-store Attributes

4.1.4.6. Generating a SecretKeyCredential in a PropertiesCredentialStore/secret-key-credential-store

Generate a SecretKeyCredential in a PropertiesCredentialStore. By default, Elytron creates a 256-bit key. If you want a different size, you can specify either a 128-bit or 192-bit key in the `key-size` attribute.

When you generate a SecretKeyCredential, Elytron generates a new random secret key and stores it as the SecretKeyCredential. You can view the contents of the credential by using the export operation on the PropertiesCredentialStore.

**IMPORTANT**

Make sure that you create a backup of either PropertiesCredentialStore, SecretKeyCredential, or both, because JBoss EAP cannot decrypt or retrieve lost Elytron credentials.

You can use the `export` operation on the PropertiesCredentialStore to get the value of the SecretKeyCredential. You can then save this value as a backup. For information, see Exporting a SecretKeyCredential from a PropertiesCredentialStore/secret-key-credential-store.

Prerequisites

- You have created a PropertiesCredentialStore.
  For information about creating a PropertiesCredentialStore, see Creating a PropertiesCredentialStore/secret-key-credential-store for a standalone server.

Procedure

- Generate a SecretKeyCredential in a PropertiesCredentialStore using the following management CLI command:

  **Syntax**
/subsystem=elytron/secret-key-credential-store=<name_of_the_properties_credential_store>:generate-secret-key(alias=<alias>, key-size=<128_or_192>)

Example

/subsystem=elytron/secret-key-credential-store=examplePropertiesCredentialStore:generate-secret-key(alias=secretKeyCredentialAlias)
{"outcome" => "success"}

Verification

- Issue the following command to verify that Elytron created a SecretKeyCredential:

Syntax

/subsystem=elytron/secret-key-credential-store=<name_of_the_properties_credential_store>:read-aliases()

Example

/subsystem=elytron/secret-key-credential-store=examplePropertiesCredentialStore:read-aliases()
{
  "outcome" => "success",
  "result" => [
    "secretkeycredentialalias",
    "key"
  ]
}

Additional resources

- PropertiesCredentialStore/secret-key-credential-store
- Creating an encrypted expression in Elytron
- secret-key-credential-store Attributes

4.1.4.7. Importing a SecretKeyCredential to PropertiesCredentialStore/secret-key-credential-store

You can import a SecretKeyCredential created outside of the PropertiesCredentialStore into an Elytron PropertiesCredentialStore. Suppose you exported a SecretKeyCredential from another credential store – a KeyStoreCredentialStore, for example – you can import it to the PropertiesCredentialStore.

Prerequisites

- You have created a PropertiesCredentialStore.
  For information about creating a PropertiesCredentialStore, see Creating a PropertiesCredentialStore/secret-key-credential-store for a standalone server.
- You have exported a SecretKeyCredential.
For information about exporting a SecretKeyCredential, see Exporting a SecretKeyCredential from a PropertiesCredentialStore/secret-key-credential-store.

**Procedure**

1. Disable caching of commands in the management CLI using the following command:

   ```
   IMPORTANT
   If you do not disable caching, the secret key is visible to anyone who can access the management CLI history file.
   ```

   ```
   history --disable
   ```

2. Import the secret key using the following management CLI command:

   **Syntax**

   ```
   /subsystem=elytron/secret-key-credential-store=<name_of_credential_store>:import-secret-key(alias=<alias>, key="<secret_key>")
   ```

   **Example**

   ```
   /subsystem=elytron/secret-key-credential-store=examplePropertiesCredentialStore:import-secret-key(alias=imported, key="RUxZAUs+Y1CzEPw0g2AHHOZ+oTKhT9osSabWQtoxR+O+42o11g==")
   ```

3. Re-enable the caching of commands using the following management CLI command:

   ```
   history --enable
   ```

**Additional resources**

- PropertiesCredentialStore/secret-key-credential-store
- secret-key-credential-store Attributes

**4.1.4.8. Listing credentials in the KeyStoreCredentialStore/credential-store**

To view all the credentials stored in the KeyStoreCredentialStore, you can list them using the management CLI.

**Procedure**

- List the credentials stored in a KeyStoreCredentialStore using the following management CLI command:

  **Syntax**

  ```
  /subsystem=elytron/credential-store=<name_of_credential_store>:read-aliases()
  ```

  **Example**
4.1.4.9. Listing credentials in the PropertiesCredentialStore/secret-key-credential-store

To view all the credentials stored in the PropertiesCredentialStore, you can list them using the management CLI.

Procedure

- List the credentials stored in a PropertiesCredentialStore using the following management CLI command:

  **Syntax**

  ```
  /subsystem=elytron/secret-key-credential-store=<name_of_credential_store>:read-aliases()
  ```

  **Example**

  ```
  /subsystem=elytron/secret-key-credential-store=examplePropertiesCredentialStore:read-aliases()
  ```

  ```
  "outcome" => "success",
  "result" => [
    "secretkeycredentialalias",
    "key"
  ]
  ```

Additional resources

- PropertiesCredentialStore/secret-key-credential-store
- secret-key-credential-store Attributes

4.1.4.10. Exporting a SecretKeyCredential from a KeyStoreCredentialStore/credential-store

You can export an existing SecretKeyCredential from a KeyStoreCredentialStore to use the SecretKeyCredential or to create a backup of the SecretKeyCredential.
Prerequisites

- You have generated a SecretKeyCredential the KeyStoreCredentialStore.
  For information about generating a SecretKeyCredential in a KeyStoreCredentialStore, see Generating a SecretKeyCredential in a KeyStoreCredentialStore/credential-store.

Procedure

- Export a SecretKeyCredential from the KeyStoreCredentialStore using the following management CLI command:

  Syntax

  `/subsystem=elytron/credential-store=<name_of_credential_store>:export-secret-key(alias=<alias>)`

  Example

  `/subsystem=elytron/credential-store=exampleKeyStoreCredentialStore:export-secret-key(alias=secretKeyCredentialAlias)
  
  "outcome" => "success",
  "result" => {"key" =>
  "RUxZAUui+8JkoDCE6mFyA3cClbSAZaXq5wgYeij1scYgdDqWiw=="}`

Additional resources

- KeyStoreCredentialStore/credential-store
- credential-store Attributes

4.1.4.11. Exporting a SecretKeyCredential from a PropertiesCredentialStore/secret-key-credential-store

You can export an existing SecretKeyCredential from a PropertiesCredentialStore to use the SecretKeyCredential or to create a backup of the SecretKeyCredential.

Prerequisites

- You have either generated a SecretKeyCredential in the PropertiesCredentialStore or imported one to it.
  For information on generating a SecretKeyCredential in a PropertiesCredentialStore, see Generating a SecretKeyCredential in a PropertiesCredentialStore/secret-key-credential-store.

  For information on importing a SecretKeyCredential to a PropertiesCredentialStore, see Importing a SecretKeyCredential to PropertiesCredentialStore/secret-key-credential-store.

Procedure

- Export a SecretKeyCredential from the PropertiesCredentialStore using the following management CLI command:

  Syntax
Example

```
/subsystem=elytron/secret-key-credential-store=examplePropertiesCredentialStore:export-secret-key(alias=secretkeycredentialalias)
{
    "outcome" => "success",
    "result" => {
        "key" => "RUxZAUtxXcYvz0aukZu+odOynlr0ByLhC72iwzJsi+ZPmONgA=="
    }
}
```

Additional resources
- PropertiesCredentialStore/secret-key-credential-store
- secret-key-credential-store Attributes

4.1.4.12. Removing a credential from KeyStoreCredentialStore/credential-store

You can store every credential type in the KeyStoreCredentialStore but, by default, when you remove a credential, Elytron assumes it’s a PasswordCredential. If you want to remove a different credential type, specify it in the `entry-type` attribute.

Procedure

- Remove a credential from the KeyStoreCredentialStore using the following management CLI command:

**Syntax**

```
/subsystem=elytron/credential-store=exampleKeyStoreCredentialStore:remove-alias(alias=passwordCredentialAlias)
```

**Example removing a PasswordCredential**

```
/subsystem=elytron/credential-store=exampleKeyStoreCredentialStore:remove-alias(alias=passwordCredentialAlias)
{
    "outcome" => "success",
    "response-headers" => {
        "warnings" => {
            "warning" => "Update dependent resources as alias ‘passwordCredentialAlias’ does not exist anymore",
            "level" => "WARNING",
            "operation" => {
                "address" => [
                    ({"subsystem" => "elytron"},
                    {"credential-store" => "exampleKeyStoreCredentialStore"}),
                    "remove-alias"
                ]
            }
        }
    }
}
```
Example removing a SecretKeyCredential

```
/subsystem=elytron/credential-store=exampleKeyStoreCredentialStore:remove-alias(alias=secretKeyCredentialAlias, entry-type=SecretKeyCredential)
{
    "outcome" => "success",
    "response-headers" => {"warnings" => [{
        "warning" => "Update dependent resources as alias 'secretKeyCredentialAlias' does not exist anymore",
        "level" => "WARNING",
        "operation" => {"address" => ["subsystem" => "elytron"], "credential-store" => "exampleKeyStoreCredentialStore"}, "operation" => "remove-alias"}
    }]
}
```

Verification

- Issue the following command to verify that Elytron removed the credential:

  **Syntax**

  ```
  /subsystem=elytron/credential-store=<name_of_credential_store>:read-aliases()
  ```

  **Example**

  ```
  /subsystem=elytron/credential-store=exampleKeyStoreCredentialStore:read-aliases()
  {
    "outcome" => "success",
    "result" => []
  }
  ```

  The credential you removed is not listed.

Additional resources

- `KeyStoreCredentialStore/credential-store`
- `credential-store Attributes`

4.1.4.13. Removing a credential from the PropertiesCredentialStore/secret-key-credential-store

You can store only the SecretKeyCredential type in a PropertiesCredentialStore. This means that, when you remove a credential from a PropertiesCredentialStore, you don’t have to specify an `entry-type`.

Procedure
Remove a SecretKeyCredential from the PropertiesCredentialStore using the following command:

**Syntax**

```
/subsystem=elytron/secret-key-credential-store=<name_of_credential_store>:remove-alias(alias=<alias>)
```

**Example**

```
/subsystem=elytron/secret-key-credential-store=examplePropertiesCredentialStore:remove-alias(alias=secretKeyCredentialAlias)
{
    "outcome" => "success",
    "response-headers" => {
        "warnings" => [{
            "warning" => "Update dependent resources as alias 'secretKeyCredentialAlias' does not exist anymore",
            "level" => "WARNING",
            "operation" => {
                "address" => [
                    {"subsystem" => "elytron"},
                    {"secret-key-credential-store" => "examplePropertiesCredentialStore"}
                ],
                "operation" => "remove-alias"
            }
        }]
    }
}
```

**Verification**

Issue the following command to verify that Elytron removed the credential:

**Syntax**

```
/subsystem=elytron/secret-key-credential-store=<name_of_credential_store>:read-aliases()
```

**Example**

```
/subsystem=elytron/secret-key-credential-store=examplePropertiesCredentialStore:read-aliases()
{
    "outcome" => "success",
    "result" => []
}
```

The credential you removed is not listed.

**Additional resources**

- PropertiesCredentialStore/secret-key-credential-store
- secret-key-credential-store Attributes
4.1.5. Credential store operations using the WildFly Elytron tool

4.1.5.1. Creating a KeyStoreCredentialStore/credential-store using the WildFly Elytron tool

In Elytron, you can create a KeyStoreCredentialStore offline where you can save all the credential types.

Procedure

- Create a KeyStoreCredentialStore using the WildFly Elytron tool with the following command:

  **Syntax**

  ```
  $ EAP_HOME/bin/elytron-tool.sh credential-store --create --location "<path_to_store_file>" --password <store_password>
  ```

  **Example**

  ```
  $ EAP_HOME/bin/elytron-tool.sh credential-store --create --location ".../cred_stores/example-credential-store.jceks" --password storePassword
  
  Credential Store has been successfully created
  ```

  If you don’t want to include your store password in the command, omit that argument and then enter the password manually at the prompt. You can also use a masked password generated by the WildFly Elytron tool. For information about generating masked passwords, see Generating masked encrypted strings using the WildFly Elytron tool.

Additional resources

- KeyStoreCredentialStore/credential-store
- Generating masked encrypted strings using the WildFly Elytron tool

4.1.5.2. Creating a KeyStoreCredentialStore/credential-store using the Bouncy Castle provider

Create a KeyStoreCredentialStore using the Bouncy Castle provider.

Prerequisites

- Make sure that your environment is configured to use Bouncy Castle. For more information, see Configure Your Environment to use the Bouncy Castle Provider.

**NOTE**

You cannot have the same name for a **credential-store** and a **secret-key-credential-store** because they implement the same Elytron capability: `org.wildfly.security.credential-store`.

Procedure

1. Define a Bouncy Castle FIPS Keystore (BCFKS) keystore. FIPS stands for Federal Information Processing Standards. If you already have one, move on to the next step.
$ keytool -genkeypair -alias <key_pair_alias> -keyalg <key_algorithm> -keysize <key_size> -storepass <key_pair_and_keystore_password> -keystore <path_to_keystore> -storetype BCFKS -keypass <key_pair_and_keystore_password>

**IMPORTANT**

Make sure that the keystore `keypass` and `storepass` attributes are identical. If they aren’t, the BCFKS keystore in the elytron subsystem can’t define them.

2. Generate a secret key for the KeyStoreCredentialStore.

$ keytool -genseckey -alias <key_alias> -keyalg <key_algorithm> -keysize <key_size> -keystore <path_to_keystore> -storetype BCFKS -storepass <key_and_keystore_password> -keypass <key_and_keystore_password>

3. Define the KeyStoreCredentialStore using the WildFly Elytron tool with the following command:

$ EAP_HOME/bin/elytron-tool.sh credential-store -c -a <alias> -x <alias_password> -p <key_and_keystore_password> -l <path_to_keystore> -u "keyStoreType=BCFKS;external=true;keyAlias=<key_alias>;externalPath=<path_to_credential_store>"

**Additional resources**

- KeyStoreCredentialStore/credential-store
- WildFly Elytron tool KeyStoreCredentialStore/credential-store operations

4.1.5.3. Creating a PropertiesCredentialStore/secret-key-credential-store using WildFly Elytron tool

In Elytron, you can create a PropertiesCredentialStore offline where you can save SecretKeyCredential instances.

**Procedure**

- Create a PropertiesCredentialStore using the WildFly Elytron tool with the following command:

**Syntax**

$$EAP_HOME/bin/elytron-tool.sh credential-store --create --location "<path_to_store_file>" --type PropertiesCredentialStore$$

**Example**

$$bin/elytron-tool.sh credential-store --create --location=standalone/configuration/properties-credential-store.cs --type PropertiesCredentialStore$$

Credential Store has been successfully created

**Additional resources**

- PropertiesCredentialStore/secret-key-credential-store
4.1.5.4. WildFly Elytron tool KeyStoreCredentialStore/credential-store operations

You can do various KeyStoreCredentialStore tasks using the WildFly Elytron tool, including the following:

Add a PasswordCredential

You can add a PasswordCredential to a KeyStoreCredentialStore using the following WildFly Elytron tool command:

**Syntax**

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "<path_to_store_file>" --password <store_password> --add <alias> --secret <sensitive_string>
```

**Example**

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "../cred_stores/example-credential-store.jceks" --password storePassword --add examplePasswordCredential --secret speci@l_db_p@$$_01
Alias "examplePasswordCredential" has been successfully stored
```

If you don’t want to put your secret in the command, omit that argument, then enter the secret manually when prompted.

Generate a SecretKeyCredential

You can add a SecretKeyCredential to a KeyStoreCredentialStore using the following WildFly Elytron tool command:

**Syntax**

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --generate-secret-key=example --location=<path_to_the_credential_store> --password <store_password>
```

**Example**

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --generate-secret-key=example --location="../cred_stores/example-credential-store.jceks" --password storePassword
Alias "example" has been successfully stored
```

If you don’t want to put your secret in the command, omit that argument, then enter the secret manually when prompted.

By default, when you create a SecretKeyCredential in JBoss EAP, you create a 256-bit secret key. If you want to change the size, you can specify `--size=128` or `--size=192` to create 128-bit or 192-bit keys respectively.

Import a SecretKeyCredential

You can import a SecretKeyCredential using the following WildFly Elytron tool command:

**Syntax**

```
```

```
Example

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --import-secret-key=imported --
location=<path_to_credential_store> --password=<store_password>
```

Enter the secret key you want to import.

List all the credentials

You can list the credentials in the KeyStoreCredentialStore using the following WildFly Elytron tool command:

Syntax

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "<path_to_store_file>" --password
<store_password> --aliases
```

Example:

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --location ".../cred_stores/example-credential-
store.jceks" --password storePassword --aliases
Credential store contains following aliases: examplepasswordcredential example
```

Check if an alias exists

Use the following command to check whether an alias exists in a credential store:

Syntax

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "<path_to_store_file>" --password
<store_password> --exists <alias>
```

Example:

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "../cred_stores/example-credential-
store.jceks" --password storePassword --exists examplepasswordcredential
Alias "examplepasswordcredential" exists
```

Export a SecretKeyCredential

You can export a SecretKeyCredential from a KeyStoreCredentialStore using the following command:

Syntax

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --export-secret-key=<alias> --
location=<path_to_credential_store> --password=storePassword
```

Example

```
Remove a credential

You can remove a credential from a credential store using the following command:

**Syntax**

```bash
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "<path_to_store_file>" --password <store_password> --remove <alias>
```

**Example**

```bash
$ EAP_HOME/bin/elytron-tool.sh credential-store --location ../cred_stores/example-credential-store.jceks --password storePassword --remove examplepasswordcredential
```

Alias "examplepasswordcredential" has been successfully removed

**Additional resources**

- KeyStoreCredentialStore/credential-store
- Credential types in Elytron

4.1.5.5. WildFly Elytron tool PropertiesCredentialStore/secret-key-credential-store operations

You can do the following PropertiesCredentialStore operations for SecretKeyCredential using the WildFly Elytron tool:

**Generate a SecretKeyCredential**

You can generate a SecretKeyCredential in a PropertiesCredentialStore using the following WildFly Elytron tool command:

**Syntax**

```bash
$ EAP_HOME/bin/elytron-tool.sh credential-store --generate-secret-key=example --location "<path_to_the_credential_store>" --type PropertiesCredentialStore
```

**Example**

```bash
$ EAP_HOME/bin/elytron-tool.sh credential-store --generate-secret-key=example --location "standalone/configuration/properties-credential-store.cs" --type PropertiesCredentialStore
```

Alias "example" has been successfully stored

**Import a SecretKeyCredential**

You can import a SecretKeyCredential using the following WildFly Elytron tool command:

**Syntax**

```bash
$ EAP_HOME/bin/elytron-tool.sh credential-store --export-secret-key=example --location=../cred_stores/example-credential-store.jceks --password=storePassword
```

Exported SecretKey for alias example=RUxZAUtBiAnoLP1CA+i6DtcbkZHflybBJxPeS9mlVOmEYwjjmEA==
List all the credentials
You can list the credentials in the PropertiesCredentialStore using the following WildFly Elytron tool command:

**Syntax**

```shell
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "<path_to_store_file>" --aliases --type PropertiesCredentialStore
```

**Example**

```shell
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "standalone/configuration/properties-credential-store.cs" --aliases --type PropertiesCredentialStore
Credential store contains following aliases: example
```

Export a SecretKeyCredential
You can export a SecretKeyCredential from a PropertiesCredentialStore using the following command:

**Syntax**

```shell
$ EAP_HOME/bin/elytron-tool.sh credential-store --export-secret-key=<alias> --location "<path_to_credential_store>" --type PropertiesCredentialStore
```

**Example**

```shell
$ EAP_HOME/bin/elytron-tool.sh credential-store --export-secret-key=example --location "standalone/configuration/properties-credential-store.cs" --type PropertiesCredentialStore
Exported SecretKey for alias example=RUxZAUt1EZM7PsYRgMGypkGirSel+5Eix4aSgwop6jfxGYUQaQ==
```

Remove a credential
You can remove a credential from a credential store using the following command:

**Syntax**

```shell
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "<path_to_store_file>" --remove <alias> --type PropertiesCredentialStore
```

**Example**

```shell
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "<path_to_store_file>" --remove <alias> --type PropertiesCredentialStore
```
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "standalone/configuration/properties-credential-store.cs" --remove example --type PropertiesCredentialStore
Alias "example" has been successfully removed

Additional resources

- PropertiesCredentialStore/secret-key-credential-store
- Credential types in Elytron

4.1.5.6. Adding a credential store created with the WildFly Elytron tool to a JBoss EAP Server

After you have created a credential store with the WildFly Elytron tool, you can add it to your running JBoss EAP server.

Prerequisites

- You have created a credential store with the WildFly Elytron tool.
  For more information, see Creating a KeyStoreCredentialStore/credential-store using the WildFly Elytron tool.

Procedure

- Add the credential store to your running JBoss EAP server with the following management CLI command:

  /subsystem=elytron/credential-store=<store_name>:add(location="<path_to_store_file>",credential-reference={clear-text="<store_password>"})

  For example:

  /subsystem=elytron/credential-store=my_store:add(location="../cred_stores/example-credential-store.jceks",credential-reference={clear-text=storePassword})

After adding the credential store to the JBoss EAP configuration, you can then refer to a password or sensitive string stored in the credential store using the credential-reference attribute.

For more information, use the EAP_HOME/bin/elytron-tool.sh credential-store --help command for a detailed listing of available options.

Additional resources

- Using a PasswordCredential in your JBoss EAP configuration
- credential-store attributes

4.1.5.7. WildFly Elytron tool key pair management operations

You can use the following arguments to operate the elytron-tool.sh to manipulate a credential store, such as generating a new key pair that you can store under an alias in a credential store.
Generate a key pair

Use the `generate-key-pair` command to create a key pair. You can then store the key pair under an alias in the credential store. The following example shows the creation of an RSA key pair, which has an allocated size of 3072 bits that is stored in the location specified for the credential store. The alias given to the key pair is `example`.

```bash
$ EAP_HOME/bin/elytron-tool.sh credential-store --location=<path_to_store_file> --generate-key-pair example --algorithm RSA --size 3072
```

Import a key pair

Use the `import-key-pair` command to import an existing SSH key pair into a credential store with a specified alias. The following example imports a key pair with the alias of `example` from the `/home/user/.ssh/id_rsa` file containing the private key in the OpenSSH format:

```bash
$ EAP_HOME/bin/elytron-tool.sh credential-store --import-key-pair example --private-key-location /home/user/.ssh/id_rsa --location=<path_to_store_file>
```

Export a key pair

Use the `export-key-pair-public-key` command to display the public key of a key pair. The public key has a specified alias in the OpenSSH format. The following example displays the public key for the alias `example`:

```bash
$ EAP_HOME/bin/elytron-tool.sh credential-store --location=<path_to_store_file> --export-key-pair-public-key example
```

Credential store password:
Confirm credential store password:
ecdsa-sha2-nistp256
AAAAE2VjZHNhLXNoYTItbmlzdHAYNTYAAAAAlbmlzdHAYNTYAAAAABBBMfncZuHmR7uglb0M96ieArRFtp42xPn9+ugukbY8dyjOXoi
cZRyy9+X68fyIEWBMzyg+nhjWkxJJu2M2LAGY=

**NOTE**

After issuing the `export-key-pair-public-key` command, you are prompted to enter the credential store passphrase. If no passphrase exists, leave the prompt blank.

4.1.5.8. Example use of stored key pair in the Elytron configuration files

A key pair consists of two separate, but matching, cryptographic keys: a public key and a private key. You need to store a key pair in a credential store before you can reference the key pair in an `elytron` configuration file. You can then provide Git with access to manage your standalone server configuration data.

The following example references a credential store and its properties in the `<credential-stores>` element of an `elytron` configuration file. The `<credential>` element references the credential store and the alias, which stores the key pair.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.6">
```

188
After you configure the elytron configuration file, the key pair can be used for SSH authentication.

Additional resources

- WildFly Elytron tool key pair management operations

4.1.5.9. Generating masked encrypted strings using the WildFly Elytron tool

You can use the WildFly Elytron tool to generate PicketBox-compatible MASK- encrypted strings to use instead of a plain text password for a credential store.

Procedure

- To generate a masked string, use the following command and provide values for the salt and the iteration count:

  ```
  $ EAP_HOME/bin/elytron-tool.sh mask --salt <salt> --iteration <iteration_count> --secret <password>
  ```

  For example:

  ```
  $ EAP_HOME/bin/elytron-tool.sh mask --salt 12345678 --iteration 123 --secret supersecretstorepassword
  MASK-8VzWsSNwBaR676g8ujlDdFKwSjOBHCHgnKf17nun3v;12345678;123
  ```

If you do not want to provide the secret in the command, you can omit that argument and you will be prompted to enter the secret manually using standard input.
For more information, use the `EAP_HOME/bin/elytron-tool.sh mask --help` command for a detailed listing of available options.

### 4.1.6. Encrypted expressions in Elytron

To maintain the secrecy of your sensitive strings, you can use encrypted expressions instead of the sensitive strings in the server configuration file.

An encrypted expression is one that results from encrypting a string with a SecretKeyCredential, then combining it with its encoding prefix and resolver name. The encoding prefix tells Elytron that the expression is an encrypted expression. The resolver maps the encrypted expression to its corresponding SecretKeyCredential in a credential store.

The `expression=encryption` resource in Elytron uses an encrypted expression to decode the encrypted string inside it at run time. By using an encrypted expression instead of the sensitive string itself in the configuration file, you protect the secrecy of the string. An encrypted expression takes the following format:

**Syntax when using a specific resolver**

```
${ENC::RESOLVER_NAME:ENCRYPTED_STRING}
```

- **ENC** is the prefix that denotes an encrypted expression.
- **RESOLVER_NAME** is the resolver Elytron uses to decrypt the encrypted string.

**Example**

```
${ENC::initialresolver:RUxZAUMQE+L5zx9LmCRLyh5jdf1WM7hftKjeoEU+x+RMi6s=}
```

If you create an encrypted expression with a default resolver, it looks like this:

**Syntax when using the default resolver**

```
${ENC::ENCRYPTED_STRING}
```

**Example**

```
${ENC::RUxZAUMQE+L5zx9LmCRLyh5jdf1WM7hftKjeoEU+x+RMi6s=}
```

In this case, Elytron uses the default resolver you defined in the `expression=encryption` resource to decrypt an expression. You can use an encrypted expression on any resource attribute that supports it. To find out whether an attribute supports encrypted expression, use the `read-resource-description` operation, for example:

**Example read-resource-description on mail/mail-session**

```
/subsystem@mail/mail-session=":/read-resource-description(recursive=true,access-control=none)
{
  "outcome":"success",
  "result":{...
  "from":{
```
In this example, the attribute `from` supports encrypted expressions. This means that you can hide your email address in the `from` field by encrypting it and then using the encrypted expression instead.

**Additional resources**

- Creating an encrypted expression in Elytron
- `expression=encryption` Attributes

### 4.1.7. Creating an encrypted expression in Elytron

Create an encrypted expression from a sensitive string and a SecretKeyCredential. Use this encrypted expression instead of the sensitive string in the management model - the server configuration file, to maintain the secrecy of the sensitive string.

**Prerequisites**

- You have generated a secret key in some credential store.
  
  For information on creating a secret key in a **KeyStoreCredentialStore**, see *Generating a SecretKeyCredential in a KeyStoreCredentialStore/credential-store*.

  
  For information on creating a secret key in a **PropertiesCredentialStore**, see *Generating a SecretKeyCredential in a PropertiesCredentialStore/secret-key-credential-store*.

**Procedure**

1. Create a resolver that references the alias of an existing SecretKeyCredential in a credential store using the following management CLI command:

   **Syntax**

   ```
   /subsystem=elytron/expression=encryption:add(resolvers=[{name=<name_of_the_resolver>, credential-store=<name_of_credential_store>, secret-key=<secret_key_alias>}])
   ```

   **Example**

   ```
   /subsystem=elytron/expression=encryption:add(resolvers=[{name=exampleResolver, credential-store=examplePropertiesCredentialStore, secret-key=key}])
   ```

   If an error message about a duplicate resource displays, use the `list-add` operation instead of `add`, as follows:

   **Syntax**

   ```
   /subsystem=elytron/expression=encryption:list-add(name=resolvers, value=
   {name=<name_of_the_resolver>, credential-store=<name_of_credential_store>, secret-key=<secret_key_alias>})
   ```
Example

```
/subsystem=elytron/expression=encryption:list-add(name=resolvers,value=
{name=exampleResolver, credential-store=examplePropertiesCredentialStore, secret-
key=key})
{
  "outcome" => "success",
  "response-headers" => {
    "operation.requires-reload" => true,
    "process-state" => "reload-required"
  }
}
```

2. Reload the server using the following management CLI command:

```
reload
```

3. Disable caching of commands in the management CLI:

```
IMPORTANT

If you do not disable caching, the secret key is visible to anyone who can access
the management CLI history file.
```

```
history --disable
```

4. Create an encrypted expression using the following management CLI command:

**Syntax**

```
/subsystem=elytron/expression=encryption:create-expression(resolver=<existing Resolver>,
clear-text=<sensitive_string_to_protect>)
```

**Example**

```
/subsystem=elytron/expression=encryption:create-expression(resolver=exampleResolver,
clear-text=TestPassword)
{
  "outcome" => "success",
  "result" => {
    "expression" => "${ENC::exampleResolver:RUxZAUMQgtpG7oFIHR2j1Gkn3GKIHff+HR8GcMX1QXHvx2uGurI=}"}
}
```

`${ENC::exampleResolver:RUxZAUMQgtpG7oFIHR2j1Gkn3GKIHff+HR8GcMX1QXHvx2uGurI=}` is the encrypted expression you use instead of **TestPassword** in the management model.

If you use the same plain text in different locations, repeat this command each time before you use the encrypted expression instead of the plain text in that location. When you repeat the same command for the same plain text, you get a different result for the same key because Elytron uses a unique initialization vector for each call.
By using different encrypted expressions you make sure that, if one encrypted expression on a string is somehow compromised, users cannot discover that any other encrypted expressions might also contain the same string.

5. Re-enable the command caching using the following management CLI command:

```bash
history --enable
```

Additional resources

- Using an encrypted expression to secure a KeyStoreCredentialStore/credential-store
- `expression=encryption` Attributes

### 4.1.8. Using a PasswordCredential in your JBoss EAP configuration

To refer to a password or sensitive string stored in a credential store, use the `credential-reference` attribute in your JBoss EAP configuration. You can use `credential-reference` as an alternative to providing a password or other sensitive string in most places throughout the JBoss EAP configuration.

**Prerequisites**

- You have added a PasswordCredential to a KeyStoreCredentialStore. For information on adding PasswordCredential to a KeyStoreCredentialStore, see Adding a PasswordCredential to a KeyStoreCredentialStore.

**Procedure**

- Reference the existing KeyStoreCredentialStore and the alias to the PasswordCredential in the `credential-reference` attribute:

  **Syntax**

  ```
  credential-reference={store=<store_name>, alias=<alias>}
  ```

  **Example**

  ```
  data-source add --name=example_data_source --jndi-name=java:/example_data_source --driver-name=h2 --connection-url=jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE --user-name=db_user --credential-reference={store=exampleKeyStoreCredentialStore, alias=passwordCredentialAlias}
  ```

  In this example, an existing PasswordCredential with the alias `passwordCredentialAlias` in a KeyStoreCredentialStore `exampleKeyStoreCredentialStore` is used instead of the plain text password for the database, protecting the database password.

**Additional resources**

- Obtain the password for the credential store from an external source.
- Credential types in Elytron
4.1.9. Using an encrypted expression to secure a KeyStoreCredentialStore/credential-store

You can use an encrypted expression to secure a KeyStoreCredentialStore.

Prerequisites

- You have created an encrypted expression. For information about creating an encrypted expression, see Creating an encrypted expression in Elytron.

Procedure

- Create a KeyStoreCredentialStore that uses an encrypted expression as the clear-text:

  Syntax

  ```
  /subsystem=elytron/credential-store=<name_of_credential_store>:add(path=<path_to_the_credential_store>, create=true, modifiable=true, credential-reference={clear-text=<encrypted_expression>})
  ```

  Example

  ```
  /subsystem=elytron/credential-store=secureKeyStoreCredentialStore:add(path=secureKeyStoreCredentialStore.jceks, relative-to=jboss.server.data.dir, create=true, modifiable=true, credential-reference={clear-text=${ENC::exampleResolver:RUxZAUMQgtpG7oFlHR2j1Gkn3GKIHff+HR8GcMX1QXHvx2uGurl}})
  ```

  "outcome" => "success"

Additional resources

- expression-encryption Attributes
- credential-store Attributes

4.1.10. Automatic update of credentials in credential store

If you have a credential store, you are not required to add credentials or update existing credentials before you can reference them from a credential reference. Elytron automates this process. When configuring a credential reference, specify both the store and clear-text attributes. Elytron automatically adds or updates a credential in the credential store specified by the store attribute. Optionally, you can specify the alias attribute.

Elytron updates the credential store as follows:

- If you specify an alias:
  - If an entry for the alias exists, the existing credential is replaced with the specified clear text password.
  - If an entry for the alias does not exist, a new entry is added to the credential store with the specified alias and the clear text password.
• If you do not specify an alias, Elytron generates an alias and adds a new entry to the credential store with the generated alias and the specified clear text password.

The clear-text attribute is removed from the management model when the credential store is updated.

The following example illustrates how to create a credential reference that specifies the store, clear-text, and alias attributes:

```
/subsystem=elytron/key-store=exampleKS:add(relative-to=jboss.server.config.dir,
  path=example.keystore, type=JCEKS, credential-reference=
  {store=exampleKeyStoreCredentialStore, alias=myNewAlias, clear-text=myNewPassword})
{
  "outcome" => "success",
  "result" => {"credential-store-update" => {
    "status" => "new-entry-added",
    "new-alias" => "myNewAlias"
  }},
}
```

You can update the credential for the myNewAlias entry that was added to the previously defined credential store with the following command:

```
/subsystem=elytron/key-store=exampleKS:write-attribute(name=credential-reference.clear-text,value=myUpdatedPassword)
{
  "outcome" => "success",
  "result" => {"credential-store-update" => {"status" => "existing-entry-updated"},
    "response-headers" => {
      "operation-requires-reload" => true,
      "process-state" => "reload-required"
    }
  }
}
```

**NOTE**

If an operation that includes a credential-reference parameter fails, no automatic credential store update occurs.

The credential store that was specified by the credential-reference attribute does not change.

4.1.11. Defining FIPS 140-2 compliant credential stores

You can define a Federal Information Processing Standards (FIPS) 140-2-compliant credential store using a Network Security Services (NSS) database, or with a Bouncy Castle provider.

4.1.11.1. Defining a FIPS 140-2 compliant credential store using an NSS database

To get a Federal Information Processing Standards (FIPS)-compliant keystore, use a Sun PKCS#11 (PKCS stands for Public Key Cryptography Standards) provider accessing a Network Security Services (NSS) database. For instructions on defining the database, see Configuring the NSS Database.

Procedure
1. Create a secret key to be used in the credential store.

NOTE
For the keytool command to work, in the nss_pkcsll_fips.cfg file, you must assign the nssDbMode attribute as readWrite.

```bash
$ keytool -keystore NONE -storetype PKCS11 -storepass <keystore_password> -genseckey -alias <key_alias> -keyalg <key_algorithm> -keysize <key_size>
```

2. Create an external credential store. An external credential store holds a secret key in a PKCS#11 keystore and accesses this keystore using the alias defined in the previous step. This keystore is then used to decrypt the credentials in a Java Cryptography Extension Keystore (JCEKS) keystore. In addition to the credential-store attributes, Elytron uses the credential-store KeyStoreCredentialStore implementation properties to configure external credential stores.

```bash
/subsystem=elytron/credential-store=<store_name>:add(modifiable=true, implementation-properties={"keyStoreType":"PKCS11", "external":"true", "keyAlias":"<key_alias>", externalPath="<path_to_JCEKS_file>", credential-reference={clear-text="<keystore_password>", create=true})
```

3. Once created, the credential store can be used to store aliases as normal.

```bash
/subsystem=elytron/credential-store=<store_name>:add-alias(alias="<alias>", secret-value="<sensitive_string>")
```

4. Confirm that the alias has been added successfully by reading from the credential store.

```bash
/subsystem=elytron/credential-store=<store_name>:read-aliases()
```

Additional resources

- Configuring the NSS Database
- credential-store Attributes
- credential-store KeyStoreCredentialStore implementation properties

4.1.11.2. Defining a FIPS 140-2 compliant credential store using Bouncy Castle providers

Define a Federal Information Processing Standards (FIPS) 140–2 compliant credential store using Bouncy Castle providers.

**Prerequisites**

- Ensure that your environment is configured to use the BouncyCastle provider. For more information, see Configure Your Environment to use the BouncyCastle Provider .

**Procedure**

1. Create a secret key to be used in the credential store.
$ keytool -genseckey -alias <key_alias> -keyalg <key_algorithm> -keysize <key_size> -keystore <path_to_keystore> -storetype BCFKS -storepass <key_and_keystore_password> -keypass <key_and_keystore_password>

**IMPORTANT**

The `keypass` and `storepass` for the keystore must be identical for FIPS credential stores to be defined in the `elytron` subsystem.

2. Create an external credential store. An external credential store holds a secret key in a BCFKS keystore, and accesses this keystore using the alias defined in the previous step. This keystore is then used to decrypt the credentials in a JCEKS keystore. The `credential-store KeyStoreCredentialStore` implementation properties are used to configure external credential stores.

```
/subsystem=elytron/credential-store=<BCFKS_credential_store>:add(relative-to=jboss.server.config.dir,credential-reference={clear-text=<key_and_keystore_password>},implementation-properties={keyAlias=<key_alias>,external=true,externalPath=<path_to_credential_store>,keyStoreType=BCFKS},create=true,location=<path_to_keystore>,modifiable=true)
```

3. Once created, the credential store can be used to store aliases as normal.

```
/subsystem=elytron/credential-store=<BCFKS_credential_store>:add-alias(alias="<alias>",secret-value="<sensitive_string>")
```

4. Confirm that the alias has been added successfully by reading from the credential store.

```
/subsystem=elytron/credential-store=<BCFKS_credential_store>:read-aliases()
```

**Additional resources**

- `credential-store Attributes`
- `credential-store KeyStoreCredentialStore` implementation properties

### 4.1.12. Using a custom implementation of the credential store

Use a custom implementation of the credential store.

**Procedure**

1. Create a class that extends the Service Provider Interface (SPI) `CredentialStoreSpi` abstract class.

2. Create a class that implements the Java Security `Provider`. The provider must add the custom credential store class as a service.

3. Create a module containing your credential store and provider classes, and add it to JBoss EAP with a dependency on `org.wildfly.security.elytron`. For example:
module add --name=org.jboss.customcredstore --
resources=/path/to/customcredstoreprovider.jar --dependencies=org.wildfly.security.elytron --
slot=main

4. Create a provider loader for your provider. For example:

/subsystem=elytron/provider-loader=myCustomLoader:add(class-names=
[org.wildfly.security.mycustomcredstore.CustomElytronProvider],module=org.jboss.customcredstore)

5. Create a credential store using the custom implementation.

NOTE

Ensure that you specify the correct providers and type values. The value of type is what is used in your provider class where it adds your custom credential store class as a service.

For example:

/subsystem=elytron/credential-store=my_store:add(providers=myCustomLoader,type=CustomKeyStorePasswordStore,location="cred_stores/my_store.jceks",relative-to=jboss.server.data.dir,credential-reference={clear-text=supersecretstorepassword},create=true)

Alternatively, if you have created multiple providers, you can specify the additional providers using another provider loader with other-providers. This allows you to have other additional implementations for new types of credentials. These specified other providers are automatically accessible in the custom credential store’s initialize method as the Provider[] argument. For example:

/subsystem=elytron/credential-store=my_store:add(providers=myCustomLoader,other-providers=myCustomLoader2,type=CustomKeyStorePasswordStore,location="cred_stores/my_store.jceks",relative-to=jboss.server.data.dir,credential-reference={clear-text=supersecretstorepassword},create=true)

4.1.13. Obtain the password for the credential store from an external source

Instead of providing your credential store’s password in clear text format, you can choose to provide the password by using a pseudo credential store.

You have the following options for providing a password:

EXT

External command using java.lang.Runtime#exec(java.lang.String). You can supply parameters to the command with a space-separated list of strings. An external command refers to any executable file from the operating system, for example, a shell script or an executable binary file. Elytron reads the password from the standard output of the command that you run.

Example
credential-reference={clear-text="[EXT]/usr/bin/getThePasswordScript.sh par1 par2", type="COMMAND"}

CMD

External command using java.lang.ProcessBuilder. You can supply parameters to the command with a comma-separated list of strings. An external command refers to any executable file from the operating system, for example, a shell script or an executable binary file. Elytron reads the password from the standard output of the command that you run.

Example

credential-reference={clear-text="[CMD]/usr/bin/getThePasswordScript.sh par1,par2", type="COMMAND"}

MASK

Masked password using PBE, or Password-Based Encryption. It must be in the following format, which includes the SALT and ITERATION values:

credential-reference={clear-text="MASK-MASKED_VALUE;SALT;ITERATION"}

Example

credential-reference={clear-text="MASK-NqMznhSbL3lwRpDmyuqLBW==;12345678;123"}

IMPORTANT

EXT, CMD, and MASK provide backward compatibility with the legacy security vault style of supplying an external password. For MASK you must use the above format that includes the SALT and ITERATION values.

You can also use a password located in another credential store as the password for a new credential store.

Example Credential Store Created with a Password from Another Credential Store

/subsystem=elytron/credential-store=exampleCS:add(location="cred_stores/exampleCS.jceks", relative-to=jboss.server.data.dir, create=true, credential-reference={store=cred-store, alias=pwd})

Additional resources

- Providing an initial key to JBoss EAP to unlock secured resources

4.1.14. Providing an initial key to JBoss EAP to unlock secured resources

For security, some JBoss EAP components are protected by a PasswordCredential in KeyStoreCredentialStore. This KeyStoreCredentialStore is in turn protected by a secret key stored external to JBoss EAP. This is referred to as a master key. JBoss EAP uses this master key during startup to unlock the KeyStoreCredentialStore to obtain the PasswordCredential stored in the KeyStoreCredentialStore.
You can use a PropertiesCredentialStore in Elytron to provide the master key. Alternately, you can obtain the master key or password from an external source. For information about obtaining the password from an external source, see Obtain the password for the credential store from an external source.

4.1.14.1. Creating a PropertiesCredentialStore/secret-key-credential-store for a standalone server

Create a PropertiesCredentialStore using the management CLI. When you create a PropertiesCredentialStore, JBoss EAP generates a secret key by default. The name of the generated key is key and its size is 256-bit.

Prerequisites

- You have provided at least read/write access to the directory containing the PropertiesCredentialStore for the user account under which JBoss EAP is running.

Procedure

- Use the following command to create a PropertiesCredentialStore using the management CLI:

  **Syntax**

  ```
  /subsystem=elytron/secret-key-credential-store=<name_of_credential_store>:add(path="<path_to_the_credential_store>", relative-to="<path_to_store_file>")
  ```

  **Example**

  ```
  /subsystem=elytron/secret-key-credential-store=examplePropertiesCredentialStore:add(path=examplePropertiesCredentialStore.cs, relative-to=jboss.server.config.dir)
  ```

  ```
  {"outcome" => "success"}
  ```

Additional resources

- PropertiesCredentialStore/secret-key-credential-store
- Credential store operations using the JBoss EAP management CLI
- secret-key-credential-store Attributes

4.1.14.2. Creating an encrypted expression in Elytron

Create an encrypted expression from a sensitive string and a SecretKeyCredential. Use this encrypted expression instead of the sensitive string in the management model - the server configuration file, to maintain the secrecy of the sensitive string.

Prerequisites

- You have generated a secret key in some credential store.
  For information on creating a secret key in a KeyStoreCredentialStore, see Generating a SecretKeyCredential in a KeyStoreCredentialStore/credential-store
For information on creating a secret key in a PropertiesCredentialStore, see Generating a SecretKeyCredential in a PropertiesCredentialStore/secret-key-credential-store

Procedure

1. Create a resolver that references the alias of an existing SecretKeyCredential in a credential store using the following management CLI command:

   **Syntax**
   ```bash
   /subsystem=elytron/expression=encryption:add(resolvers=[{name=<name_of_the_resolver>, credential-store=<name_of_credential_store>, secret-key=<secret_key_alias>}])
   ```

   **Example**
   ```bash
   /subsystem=elytron/expression=encryption:add(resolvers=[{name=exampleResolver, credential-store=examplePropertiesCredentialStore, secret-key=key}])
   ```

   If an error message about a duplicate resource displays, use the **list-add** operation instead of **add**, as follows:

   **Syntax**
   ```bash
   /subsystem=elytron/expression=encryption:list-add(name=resolvers, value={name=<name_of_the_resolver>, credential-store=<name_of_credential_store>, secret-key=<secret_key_alias>})
   ```

   **Example**
   ```bash
   /subsystem=elytron/expression=encryption:list-add(name=resolvers,value={name=exampleResolver, credential-store=examplePropertiesCredentialStore, secret-key=key})
   ```

   ```json
   {   "outcome" => "success",   "response-headers" => {   "operation-requires-reload" => true,   "process-state" => "reload-required"   } }
   ```

2. Reload the server using the following management CLI command:

   ```bash
   reload
   ```

3. Disable caching of commands in the management CLI:

   **IMPORTANT**
   If you do not disable caching, the secret key is visible to anyone who can access the management CLI history file.

   ```bash
   history --disable
   ```
4. Create an encrypted expression using the following management CLI command:

Syntax

/subsystem=elytron/expression=encryption:create-expression(resolver=<existing_resolver>,
clear-text=<sensitive_string_to_protect>)

Example

/subsystem=elytron/expression=encryption:create-expression(resolver=exampleResolver,
clear-text=TestPassword)
{
  "outcome" => "success",
  "result" => {"expression" => "${ENC::exampleResolver:RUxZAUMQgtpG7oFIHR2j1Gkn3GKH7ff+HR8GcMX1QXHvx2uGur1=}"}
}

${ENC::exampleResolver:RUxZAUMQgtpG7oFIHR2j1Gkn3GKH7ff+HR8GcMX1QXHvx2uGu rl1=} is the encrypted expression you use instead of TestPassword in the management model.

If you use the same plain text in different locations, repeat this command each time before you use the encrypted expression instead of the plain text in that location. When you repeat the same command for the same plain text, you get a different result for the same key because Elytron uses a unique initialization vector for each call.

By using different encrypted expressions you make sure that, if one encrypted expression on a string is somehow compromised, users cannot discover that any other encrypted expressions might also contain the same string.

5. Re-enable the command caching using the following management CLI command:

 history --enable

Additional resources

- Using an encrypted expression to secure a KeyStoreCredentialStore/credential-store
- expression=encryption Attributes

4.14.3. Using an encrypted expression to secure a KeyStoreCredentialStore/credential-store

You can use an encrypted expression to secure a KeyStoreCredentialStore.

Prerequisites

- You have created an encrypted expression.
  For information about creating an encrypted expression, see Creating an encrypted expression in Elytron.

Procedure

- Create a KeyStoreCredentialStore that uses an encrypted expression as the clear-text:
CHAPTER 4. SECURE STORAGE FOR CREDENTIALS

Syntax
/subsystem=elytron/credentialstore=<name_of_credential_store>:add(path=<path_to_the_credential_store>, create=true,
modifiable=true, credential-reference={clear-text=<encrypted_expression>})

Example
/subsystem=elytron/credentialstore=secureKeyStoreCredentialStore:add(path="secureKeyStoreCredentialStore.jceks",
relative-to=jboss.server.data.dir, create=true, modifiable=true, credential-reference={cleartext=${ENC::exampleResolver:RUxZAUMQgtpG7oFlHR2j1Gkn3GKIHff+HR8GcMX1QXHvx2u
GurI=}})
{"outcome" => "success"}
Additional resources
expression-encryption Attributes
credential-store Attributes
After you have secured a KeyStoreCredentialStore with an encrypted expression, you can generate a
SecretKeyCredential in the KeyStoreCredentialStore and use the secret key to create another
encrypted expression. You can then use this new encrypted expression instead of a sensitive string in the
management model - the server configuration file. You can create an entire chain of credential stores
for security. Such a chain makes it harder to guess the sensitive string because the string is protected as
follows:
The first encrypted expression secures a KeyStoreCredentialStore.
Another encrypted expression secures a sensitive string.
To decode the sensitive string, you would need to decrypt both the encrypted expressions.
As the chain of encrypted expressions becomes longer, it gets harder to decrypt the sensitive string.

4.1.15. Converting password vaults to credential stores
You can use the WildFly Elytron tool to convert a password vault to a credential store. To convert a
password vault to a credential store, you need the vault’s values used when initializing the vault .

NOTE
When converting a password vault, aliases in the new credential store are named in the
following format based on their equivalent password vault block and attribute name:
VAULT_BLOCK::ATTRIBUTE_NAME.

4.1.15.1. Converting a single password vault to a Credential Store using the WildFly Elytron
tool
Convert a single password vault to a Credential Store using the WildFly Elytron tool.
Procedure

203


Convert the password vault to a credential store using the following command:

```
$ EAP_HOME/bin/elytron-tool.sh vault --keystore "<path_to_vault_file>" --keystore-password <vault_password> --enc-dir "<path_to_vault_directory>" --salt <salt> --iteration <iteration_count> --alias <vault_alias>
```

For example, you can also specify the new credential store’s file name and location with the `--location` argument:

```
$ EAP_HOME/bin/elytron-tool.sh vault --keystore ../vaults/vault1.keystore --keystore-password vault11 --enc-dir ../vaults/ --salt 1234abcd --iteration 120 --alias my_vault --location ../cred_stores/vault1_converted.cred_store
```

**NOTE**

You can also use the `--summary` argument to print a summary of the management CLI commands used to convert it. Note that even if a plain text password is used, it is masked in the summary output. The default `salt` and `iteration` values are used unless they are specified in the command.

### 4.1.15.2. Bulk converting password vault to a credential store using the WildFly Elytron tool

Bulk convert multiple password vaults to credential stores.

**Procedure**

1. Put the details of the vaults you want to convert into a description file in the following format:

```
keystore:<path_to_vault_file>
keystore-password:<vault_password>
enc-dir:<path_to_vault_directory>
salt:<salt>  1
iteration:<iteration_count>  2
location:<path_to_converted_cred_store>  2
alias:<vault_alias>
properties:<parameter1>=<value1>;<parameter2>=<value2>;  3
```

1. **salt** and **iteration** can be omitted if you are providing a plain text password for the vault.
2. Specifies the location and file name for the converted credential store.
3. Optional: Specifies a list of optional parameters separated by semicolons (;). See `EAP_HOME/bin/elytron-tool.sh vault --help` for a list of available parameters.

For example:

```
keystore:/vaults/vault1/vault1.keystore
keystore-password:vault11
enc-dir:/vaults/vault1/
salt:1234abcd
iteration:120
location:/cred_stores/vault1_converted.cred_store
alias:my_vault
```
2. Run the bulk convert command with your description file from the previous step:

```
$ EAP_HOME/bin/elytron-tool.sh vault --bulk-convert vaultdescriptions.txt
```

For more information, use the `EAP_HOME/bin/elytron-tool.sh vault --help` command for a detailed listing of available options.

### 4.1.16. Example of using a credential store with Elytron client

Clients connecting to JBoss EAP, such as Jakarta Enterprise Beans, can authenticate using Elytron Client. Users without access to a running JBoss EAP server can create and modify credential stores using the WildFly Elytron tool, and then clients can use Elytron Client to access sensitive strings inside a credential store.

The following example shows you how to use a credential store in an Elytron Client configuration file.

**Example custom-config.xml with a Credential Store**

```xml
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    ...
    <credential-stores>
      <credential-store name="my_store"> ¹
        <protection-parameter-credentials>
          <credential-store-reference clear-text="pass123"/> ²
        </protection-parameter-credentials>
        <attributes>
          <attribute name="location" value="/path/to/my_store.jceks"/> ³
        </attributes>
      </credential-store>
      ...
    </credential-stores>
    ...
    <authentication-configurations>
      <configuration name="my_user">
        <set-host name="localhost"/>
        <set-user-name name="my_user"/>
        <set-mechanism-realm name="ManagementRealm"/>
        <use-provider-sasl-factory/>
        <credentials>
          <credential-store-reference store="my_store" alias="my_user"/> ⁴
        </credentials>
      </configuration>
    </authentication-configurations>
  </authentication-client>
</configuration>
```
A name for the credential store for use within the Elytron Client configuration file.

The password for the credential store.

The path to the credential store file.

The credential reference for a sensitive string stored in the credential store.

Additional resources

- Configuring client authentication using Elytron Client
- Creating and modifying credential stores offline with the WildFly Elytron tool.

4.2. PASSWORD VAULT

Configuration of JBoss EAP and associated applications requires potentially sensitive information, such as user names and passwords. Instead of storing the password as plain text in configuration files, the password vault feature can be used to mask the password information and store it in an encrypted keystore. Once the password is stored, references can be included in management CLI commands or applications deployed to JBoss EAP.

The password vault uses the Java keystore as its storage mechanism. Password vault consists of two parts: storage and key storage. Java keystore is used to store the key, which is used to encrypt or decrypt sensitive strings in Vault storage.

**IMPORTANT**

The keytool utility, provided by the Java Runtime Environment (JRE), is utilized for this steps. Locate the path for the file, which on Red Hat Enterprise Linux is `/usr/bin/keytool`.

JCEKS keystore implementations differ between Java vendors so the keystore must be generated using the keytool utility from the same vendor as the JDK used. Using a keystore generated by the keytool from one vendor’s JDK in a JBoss EAP 7 instance running on a JDK from a different vendor results in the following exception:

```java
java.io.IOException: com.sun.crypto.provider.SealedObjectForKeyProtector
```

4.2.1. Set Up a Password Vault

Follow the steps below to set up and use a Password Vault.

1. Create a directory to store the keystore and other encrypted information.
   The rest of this procedure assumes that the directory is `EAP_HOME/vault/`. Since this directory will contain sensitive information it should be accessible to only limited users. At a minimum the user account under which JBoss EAP is running requires read-write access.

2. Determine the parameters to use with keytool utility.
   Decide on values for the following parameters:

   - alias
The alias is a unique identifier for the vault or other data stored in the keystore. Aliases are case-insensitive.

storetype
The storetype specifies the keystore type. The value `jceks` is recommended.

keyalg
The algorithm to use for encryption. Use the documentation for the JRE and operating system to see which other choices are available.

keysize
The size of an encryption key impacts how difficult it is to decrypt through brute force. For information on appropriate values, see the documentation distributed with the keytool utility.

storepass
The value of storepass is the password that is used to authenticate to the keystore so that the key can be read. The password must be at least 6 characters long and must be provided when the keystore is accessed. If this parameter is omitted, the keytool utility will prompt for it to be entered after the command has been executed.

keypass
The value of keypass is the password used to access the specific key and must match the value of the storepass parameter.

validity
The value of validity is the period (in days) for which the key will be valid.

keystore
The value of keystore is the file path and file name in which the keystore’s values are to be stored. The keystore file is created when data is first added to it. Ensure the correct file path separator is used: `/` (forward slash) for Red Hat Enterprise Linux and similar operating systems, `\` (backslash) for Windows Server. The keytool utility has many other options. See the documentation for the JRE or the operating system for more details.

3. Run the keytool command, ensuring `keypass` and `storepass` contain the same value.

```
$ keytool -genseckey -alias vault -storetype jceks -keyalg AES -keysize 128 -storepass vault22 -keypass vault22 -keystore EAP_HOME/vault/vault.keystore
```

This results in a keystore that has been created in the file `EAP_HOME/vault/vault.keystore`. It stores a single key, with the alias `vault`, which will be used to store encrypted strings, such as passwords, for JBoss EAP.

4.2.2. Initialize the Password Vault

The password vault can be initialized either interactively, where you are prompted for each parameter’s value, or non-interactively, where all parameter values are provided on the command line. Each method gives the same result, so either may be used.

The following parameters will be needed:

**keystore URL (KEYSTORE_URL)**

The file system path or URI of the keystore file. The examples use `EAP_HOME/vault/vault.keystore`.

**keystore password (KEYSTORE_PASSWORD)**
The password used to access the keystore.

**Salt (SALT)**

The salt value is a random string of eight characters used, together with the iteration count, to encrypt the content of the keystore.

**keystore Alias (KEYSTORE_ALIAS)**

The alias by which the keystore is known.

**Iteration Count (ITERATION_COUNT)**

The number of times the encryption algorithm is run.

**Directory to store encrypted files (ENC_FILE_DIR)**

The path in which the encrypted files are to be stored. This is typically the directory containing the password vault. It is convenient but not mandatory to store all of your encrypted information in the same place as the keystore. This directory should be only accessible to limited users. At a minimum the user account under which JBoss EAP 7 is running requires read-write access. The keystore should be located in the directory you created when you set up the password vault. Note that the trailing backslash or forward slash on the directory name is required. Ensure the correct file path separator is used: / (forward slash) for Red Hat Enterprise Linux and similar operating systems, \ (backslash) for Windows Server.

**Vault Block (VAULT_BLOCK)**

The name to be given to this block in the password vault.

**Attribute (ATTRIBUTE)**

The name to be given to the attribute being stored.

**Security Attribute (SEC-ATTR)**

The password which is being stored in the password vault.

To run the password vault command non-interactively, the `vault` script located in `EAP_HOME/bin/` can be invoked with parameters for the relevant information:

```
$ vault.sh --keystore KEYSTORE_URL --keystore-password KEYSTORE_PASSWORD --alias KEYSTORE_ALIAS --vault-block VAULT_BLOCK --attribute ATTRIBUTE --sec-attr SEC-ATTR --enc-dir ENC_FILE_DIR --iteration ITERATION_COUNT --salt SALT
```

**Example: Initializing Password Vault**

```
$ vault.sh --keystore EAP_HOME/vault/vault.keystore --keystore-password vault22 --alias vault --vault-block vb --attribute password --sec-attr 0penS3sam3 --enc-dir EAP_HOME/vault/ --iteration 120 --salt 1234abcd
```

**Example: Output**

```
========================================================================
JBoss Vault

JBoss Vault

JBOSS_HOME: EAP_HOME

JAVA: java

========================================================================
```

208
To run the password vault command interactively, the following steps are required:

1. Launch the password vault command interactively.
   Run `EAP_HOME/bin/vault.sh` on Red Hat Enterprise Linux and similar operating systems or `EAP_HOME/bin/vault.bat` on Windows Server. Start a new interactive session by typing 0 (zero).

2. Complete the prompted parameters.
   Follow the prompts to input the required parameters.

3. Make a note of the masked password information.
   The masked password, salt, and iteration count are printed to standard output. Make a note of them in a secure location. They are required to add entries to the Password Vault. Access to the keystore file and these values could allow an attacker access to obtain access to sensitive information in the Password Vault.

4. Exit the interactive console
   Type 2 (two) to exit the interactive console.

**Example: Input and Output**

Please enter a Digit:  0: Start Interactive Session 1: Remove Interactive Session 2: Exit

0
Starting an interactive session
Enter directory to store encrypted files: EAP_HOME/vault/
Enter Keystore URL: EAP_HOME/vault/vault.keystore
Enter Keystore password: vault22
Enter Keystore password again: vault22
Values match
Enter 8 character salt: 1234abcd
Enter iteration count as a number (Eg: 44): 120
Enter Keystore Alias: vault
Initializing Vault
The keystore password has been masked for use in configuration files and deployments. In addition, the vault is initialized and ready to use.

4.2.3. Use a Password Vault

Before passwords and other sensitive attributes can be masked and used in configuration files, JBoss EAP 7 must be made aware of the password vault which stores and decrypts them.

The following command can be used to configure JBoss EAP 7 to use the password vault:

```
/core-service=vault:add(vault-options=[
("KEYSTORE_URL" => "PATH_TO_KEYSTORE"),
("KEYSTORE_PASSWORD" => "MASKED_PASSWORD"),
("KEYSTORE_ALIAS" => "ALIAS"),
("SALT" => "SALT"),
("ITERATION_COUNT" => "ITERATION_COUNT"),
("ENC_FILE_DIR" => "ENC_FILE_DIR")
])
```

```
/core-service=vault:add(vault-options=[
("KEYSTORE_URL" => "EAP_HOME/vault/vault.keystore"),
("KEYSTORE_PASSWORD" => "MASK-5dOaAVafCSd"),
("KEYSTORE_ALIAS" => "vault"),
("SALT" => "1234abcd"),
("ITERATION_COUNT" => "120"),
("ENC_FILE_DIR" => "EAP_HOME/vault")
])
```

NOTE

If Microsoft Windows Server is being used, use two backslashes (\\) in the file path instead using one. For example, C:\\data\\vault\\vault.keystore. This is because a single backslash character (\) is used for character escaping.

4.2.4. Store a Sensitive String in the Password Vault

Including passwords and other sensitive strings in plaintext configuration files is a security risk. Store these strings instead in the Password Vault for improved security, where they can then be referenced in configuration files, management CLI commands and applications in their masked form.

Sensitive strings can be stored in the Password Vault either interactively, where the tool prompts for each parameter's value, or non-interactively, where all the parameters' values are provided on the command line. Each method gives the same result, so either may be used. Both of these methods are invoked using the vault script.

To run the password vault command non-interactively, the vault script (located in EAP_HOME/bin/) can be invoked with parameters for the relevant information:
NOTE
The keystore password must be given in plaintext form, not masked form.

After invoking the vault script, a message prints to standard output, showing the vault block, attribute name, masked string, and advice about using the string in your configuration. Make note of this information in a secure location. An extract of sample output is as follows:

```
NOTE
The keystore password must be given in plaintext form, not masked form.
```

```
$ vault.sh --keystore KEystore_URL --keystore-password KEystore_PASSWORD --alias KEystore_ALIAS --vault-block VAULT_BLOCK --attribute ATTRIBUTE --sec-attr SEC-ATTR --enc-dir ENC_FILE_DIR --iteration ITERATION_COUNT --salt SALT
```

```
Example: Output
```

```
Jboss Vault

JBOSS_HOME: EAP_HOME

JAVA: java

```

```
INFO: PBOX00361: Default Security Vault Implementation Initialized and Ready
WFLYSEC0047: Secured attribute value has been stored in Vault.
Please make note of the following:
*******************************************************************************
Vault Block: vb
Attribute Name: password
Configuration should be done as follows:
VAULT::vb::password::1
*******************************************************************************
WFLYSEC0048: Vault Configuration in WildFly configuration file:
*******************************************************************************
...
</extensions>
<vault>
  <vault-option name="KEYSTORE_URL" value="/vault/vault.keystore"/>
  <vault-option name="KEYSTORE_PASSWORD" value="MASK-5dOaAVafCSd"/>
  <vault-option name="KEYSTORE_ALIAS" value="vault"/>
  <vault-option name="SALT" value="1234abcd"/>
  <vault-option name="ITERATION_COUNT" value="120"/>
  <vault-option name="ENC_FILE_DIR" value="/vault/"/>
</vault>
</management> ...
*******************************************************************************
```

```
After invoking the vault script, a message prints to standard output, showing the vault block, attribute name, masked string, and advice about using the string in your configuration. Make note of this information in a secure location. An extract of sample output is as follows:

```
 Vault Block: vb
```

---

211
To run the password vault command interactively, the following steps are required:

1. Launch the Password Vault command interactively.
   
   Launch the operating system’s command line interface and run `EAP_HOME/bin/vault.sh` (on Red Hat Enterprise Linux and similar operating systems) or `EAP_HOME/bin/vault.bat` (on Microsoft Windows Server). Start a new interactive session by typing 0 (zero).

2. Complete the prompted parameters.
   
   Follow the prompts to input the required parameters. These values must match those provided when the Password Vault was created.

   **NOTE**
   
   The keystore password must be given in plaintext form, not masked form.

3. Complete the prompted parameters about the sensitive string.
   
   Enter 0 (zero) to start storing the sensitive string. Follow the prompts to input the required parameters.

4. Make note of the information about the masked string.
   
   A message prints to standard output, showing the vault block, attribute name, masked string, and advice about using the string in the configuration. Make note of this information in a secure location. An extract of sample output is as follows:

```
Vault Block: ds_Example1
Attribute Name: password
Configuration should be done as follows:
VAULT::ds_Example1::password::1
```

5. Exit the interactive console.
   
   Type 2 (two) to exit the interactive console.

**Example: Input and Output**

```
JBoss Vault
JBOSS_HOME: EAP_HOME
JAVA: java

Please enter a Digit:: 0: Start Interactive Session 1: Remove Interactive Session 2: Exit
0
Starting an interactive session
Enter directory to store encrypted files: EAP_HOME/vault/
Enter Keystore URL: EAP_HOME/vault/vault.keystore
Enter Keystore password:
Enter Keystore password again:
Values match
```
4.2.5. Use an Encrypted Sensitive String in Configuration

Any sensitive string which has been encrypted can be used in a configuration file or management CLI command in its masked form, providing expressions are allowed.

To confirm if expressions are allowed within a particular subsystem, run the following management CLI command against that subsystem:

```bash
/subsystem=SUBSYSTEM:read-resource-description(recursive=true)
```

From the output of running this command, look for the value of the `expressions-allowed` parameter. If this is true, then expressions can be used within the configuration of this subsystem.

Use the following syntax to replace any plaintext string with the masked form.
$\text{VAULT::VAULT BLOCK::ATTRIBUTE NAME::MASKED STRING}$

Example: Datasource Definition Using a Password in Masked Form

```xml
...<subsystem xmlns="urn:jboss:domain:datasources:5.0">
  <datasources>
    <datasource jndi-name="java:jboss/datasources/ExampleDS" enabled="true" use-java-context="true" pool-name="H2DS">
      <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1</connection-url>
      <driver>h2</driver>
      <pool></pool>
      <security>
        <user-name>sa</user-name>
        <password>${VAULT::ds_ExampleDS::password::1}</password>
      </security>
    </datasource>
  </datasources>
</subsystem>
```

4.2.6. Use an Encrypted Sensitive String in an Application

Encrypted strings stored in the password vault can be used in an application’s source code. The below example is an extract of a servlet’s source code, illustrating the use of a masked password in a datasource definition, instead of the plaintext password. The plaintext version is commented out so that you can see the difference.

Example: Servlet Using a Vaulted Password

```java
@DataSourceDefinition(
    name = "java:jboss/datasources/LoginDS",
    user = "sa",
    password = "VAULT::DS::thePass::1",
    className = "org.h2.jdbcx.JdbcDataSource",
    url = "jdbc:h2:tcp://localhost/mem:test"
)
/*old (plaintext) definition
@DataSourceDefinition(
    name = "java:jboss/datasources/LoginDS",
    user = "sa",
    password = "sa",
    className = "org.h2.jdbcx.JdbcDataSource",
    url = "jdbc:h2:tcp://localhost/mem:test"
)*/
```

4.2.7. Check if a Sensitive String is in the Password Vault
Before attempting to store or use a sensitive string in the Password Vault it can be useful to first confirm if it is already stored.

This check can be done either interactively, where the user is prompted for each parameter’s value, or non-interactively, where all parameters’ values are provided on the command line. Each method gives the same result, so either may be used. Both of these methods are invoked using the `vault` script.

Use the non-interactive method to provide all parameters’ values at once. For a description of all parameters, see Initialize the Password Vault. To run the password vault command non-interactively, the `vault` script located in `EAP_HOME/bin/` can be invoked with parameters for the relevant information:

```
$ vault.sh --keystore KEYSTORE_URL --keystore-password KEYSTORE_PASSWORD --alias KEYSTORE_ALIAS --check-sec-attr --vault-block VAULT_BLOCK --attribute ATTRIBUTE --enc-dir ENC_FILE_DIR --iteration ITERATION_COUNT --salt SALT
```

Substitute the placeholder values with the actual values. The values for parameters `KEYSTORE_URL`, `KEYSTORE_PASSWORD` and `KEYSTORE_ALIAS` must match those provided when the password vault was created.

**NOTE**

The keystore password must be given in plaintext form, not masked form.

If the sensitive string is stored in the vault block specified, the following message will be displayed:

```
Password already exists.
```

If the value is not stored in the specified block, the following message will be displayed:

```
Password doesn’t exist.
```

To run the password vault command interactively, the following steps are required:

1. Launch the password vault command interactively.
   Run `EAP_HOME/bin/vault.sh` (on Red Hat Enterprise Linux and similar operating systems) or `EAP_HOME/bin/vault.bat` (on Windows Server). Start a new interactive session by typing `0` (zero).

2. Complete the prompted parameters. Follow the prompts to input the required authentication parameters. These values must match those provided when the password vault was created.

   **NOTE**

   When prompted for authentication, the keystore password must be given in plaintext form, not masked form.

   - Enter 1 (one) to select **Check whether a secured attribute exists**
   - Enter the name of the vault block in which the sensitive string is stored.
   - Enter the name of the sensitive string to be checked.
If the sensitive string is stored in the vault block specified, a confirmation message like the following will be output:

- A value exists for (VAULT_BLOCK, ATTRIBUTE)

If the sensitive string is not stored in the specified block, a message like the following will be output:

- No value has been stored for (VAULT_BLOCK, ATTRIBUTE)

**Example: Check For a Sensitive String Interactively**

```plaintext
JBoss Vault
JBOSS_HOME: EAP_HOME
JAVA: java

Please enter a Digit::  0: Start Interactive Session  1: Remove Interactive Session  2: Exit

0
Starting an interactive session
Enter directory to store encrypted files:EAP_HOME/vault
Enter Keystore URL:EAP_HOME/vault/vault.keystore
Enter Keystore password:
Enter Keystore password again:
Values match
Enter 8 character salt:1234abcd
Enter iteration count as a number (Eg: 44):120
Enter Keystore Alias:vault
Initializing Vault
INFO: PBOX000361: Default Security Vault Implementation Initialized and Ready
Vault Configuration in AS7 config file:

```
<extensions>
  <vault>
    <vault-option name="KEYSTORE_URL" value="EAP_HOME/vault/vault.keystore"/>
    <vault-option name="KEYSTORE_PASSWORD" value="MASK-5dOaAVafCSd"/>
    <vault-option name="KEYSTORE_ALIAS" value="vault"/>
    <vault-option name="SALT" value="1234abcd"/>
    <vault-option name="ITERATION_COUNT" value="120"/>
    <vault-option name="ENC_FILE_DIR" value="EAP_HOME/vault/"/>
  </vault>
</extensions>
```

Vault is initialized and ready for use
Handshake with Vault complete
Please enter a Digit::  0: Store a secured attribute  1: Check whether a secured attribute exists  2: Remove secured attribute  3: Exit

1
Task: Verify whether a secured attribute exists
Enter Vault Block:vb
Enter Attribute Name:password
4.2.8. Remove a Sensitive String from the Password Vault

For security reasons it is best to remove sensitive strings from the Password Vault when they are no longer required. For example, if an application is being decommissioned, any sensitive strings used in datasource definitions should be removed at the same time.

**IMPORTANT**

As a prerequisite, before removing a sensitive string from the Password Vault, confirm if it is used in the configuration of JBoss EAP.

This operation can be done either interactively, where the user is prompted for each parameter’s value, or non-interactively, where all parameters’ values are provided on the command line. Each method gives the same result, so either may be used. Both of these methods are invoked using the `vault` script.

Use the non-interactive method to provide all parameters’ values at once. For a description of all parameters, see [Initialize the Password Vault](#). To run the password vault command non-interactively, the `vault` script (located in `EAP_HOME/bin`) can be invoked with parameters for the relevant information:

```
$ vault.sh --keystore KEYSTORE_URL --keystore-password KEYSTORE_PASSWORD --alias KEYSTORE_ALIAS --remove-sec-attr --vault-block VAULT_BLOCK --attribute ATTRIBUTE --enc-dir ENC_FILE_DIR --iteration ITERATION_COUNT --salt SALT
```

Substitute the placeholder values with the actual values. The values for parameters `KEYSTORE_URL`, `KEYSTORE_PASSWORD` and `KEYSTORE_ALIAS` must match those provided when the password Vault was created.

**NOTE**

The keystore password must be given in plaintext form, not masked form.

If the sensitive string is successfully removed, a confirmation message like the following will be displayed:

```
Secured attribute [VAULT_BLOCK::ATTRIBUTE] has been successfully removed from vault
```

If the sensitive string is not removed, a message like the following will be displayed:

```
Secured attribute [VAULT_BLOCK::ATTRIBUTE] was not removed from vault, check whether it exist
```

**Example: Output**

```
$ ./vault.sh --keystore EAP_HOME/vault/vault.keystore --keystore-password vault22 --alias vault --remove-sec-attr --vault-block vb --attribute password --enc-dir EAP_HOME/vault/ --iteration 120 --salt 1234abcd

========================================================================
JBoss Vault
JBOSST_HOME: EAP_HOME
```
Remove a Sensitive String Interactively
To run the password vault command interactively, the following steps are required:

1. Launch the password vault command interactively.
   Run `EAP_HOME/bin/vault.sh` (on Red Hat Enterprise Linux and similar operating systems) or `EAP_HOME\bin\vault.bat` (on Microsoft Windows Server). Start a new interactive session by typing 0 (zero).

2. Complete the prompted parameters.
   Follow the prompts to input the required authentication parameters. These values must match those provided when the password vault was created.

   **NOTE**

   When prompted for authentication, the keystore password must be given in plaintext form, not masked form.

   - Enter 2 (two) to choose option Remove secured attribute.
   - Enter the name of the vault block in which the sensitive string is stored.
   - Enter the name of the sensitive string to be removed.

If the sensitive string is successfully removed, a confirmation message like the following will be displayed:

Secured attribute [VAULT_BLOCK::ATTRIBUTE] has been successfully removed from vault

If the sensitive string is not removed, a message like the following will be displayed:

Secured attribute [VAULT_BLOCK::ATTRIBUTE] was not removed from vault, check whether it exist

Example: Output

```
JAVA: java
========================================================================
INFO: PBOX000361: Default Security Vault Implementation Initialized and Ready
Secured attribute [vb::password] has been successfully removed from vault
Secured attribute [VAULT_BLOCK::ATTRIBUTE] has been successfully removed from vault
Secured attribute [VAULT_BLOCK::ATTRIBUTE] was not removed from vault, check whether it exist
```

Please enter a Digit:: 0: Start Interactive Session  1: Remove Interactive Session  2: Exit
0
Starting an interactive session
Enter directory to store encrypted files:EAP_HOME/vault/
Enter Keystore URL:EAP_HOME/vault/vault.keystore
Enter Keystore password:
Enter Keystore password again:
Values match
Enter 8 character salt:1234abcd
Enter iteration count as a number (Eg: 44):120
Enter Keystore Alias:vault
Initializing Vault
4.2.9. Configure Red Hat JBoss Enterprise Application Platform to Use a Custom Implementation of the Password Vault

In addition to using the provided password vault implementation, a custom implementation of SecurityVault can also be used.

IMPORTANT

As a prerequisite, ensure that the password vault has been initialized. For more information, see Initialize the Password Vault.

To use a custom implementation for the password vault:

1. Create a class that implements the interface SecurityVault.

2. Create a module containing the class from the previous step, and specify a dependency on org.picketbox where the interface is SecurityVault.

3. Enable the custom password vault in the JBoss EAP configuration by adding the vault element with the following attributes:
   
   - code - The fully qualified name of class that implements SecurityVault.
   
   - module - The name of the module that contains the custom class.

   Optionally, the vault-options parameters can be used to initialize the custom class for a password vault.

Example: Use vault-options Parameters to Initialize the Custom Class

```
/core-service=vault:add(code="custom.vault.implementation.CustomSecurityVault",
```
4.2.10. Obtain Keystore Password From External Source

The **EXT**, **EXTC**, **CMD**, **CMDC** or **CLASS** methods can be used in vault configuration for obtaining the Java keystore password.

**<vault-option name="KEYSTORE_PASSWORD" value="METHOD_TO_OBTAIN_PASSWORD"/>**

The description for the methods are listed as:

**{EXT}...**

Refers to the exact command, where the ... is the exact command. For example:

**{EXT}/usr/bin/getmypassword --section 1 --query company**, run the **/usr/bin/getmypassword** command, which displays the password on standard output and use it as password for Security Vault’s keystore. In this example, the command is using two options: **--section 1** and **--query company**.

**{EXTC[expiration_in_millis]}...**

Refers to the exact command line that is passed to the **Runtime.exec(String)** method to execute a platform command. The first line of the command output is used as the password. **EXTC** variant caches the passwords for **expiration_in_millis** milliseconds. Default cache expiration is **0 = infinity**. For example:

**{EXTC:120000}/usr/bin/getmypassword --section 1 --query company** verifies if the cache contains **/usr/bin/getmypassword** output, if it contains the output then use it. If it does not contain the output, run the command to output it to cache and use it. In this example, the cache expires in 2 minutes, that is 120000 milliseconds.

**{CMD}... or {CMDC[expiration_in_millis]}...**

The general command is a string delimited by , (comma) where the first part is the actual command and further parts represents the parameters. The comma can be backslashed to keep it as a part of the parameter. For example, **{CMD}/usr/bin/getmypassword,--section,1,--query,company**.

**{CLASS[@jboss_module_spec]}classname[ctorargs]**

Where the [ctorargs] is an optional string delimited by the : (colon) from the classname is passed to the classname ctor. The cttorargs is a comma delimited list of strings. For example, **{CLASS@org.test.passwd}org.test.passwd.ExternamPassworProvider**. In this example, the **org.test.passwd.ExternamPassworProvider** class is loaded from **org.test.passwd** module and uses the **toCharArray()** method to get the password. If **toCharArray()** is not available the **toString()** method is used. The **org.test.passwd.ExternamPassworProvider** class must have the default constructor.
CHAPTER 5. JAVA SECURITY MANAGER

By defining a Java security policy you can configure the Java Security Manager to manage the external boundary of the Java Virtual Machine (JVM).

5.1. ABOUT THE JAVA SECURITY MANAGER

The Java Security Manager is a class that manages the external boundary of the Java Virtual Machine (JVM) sandbox, controlling how code executing within the JVM can interact with resources outside the JVM. When the Java Security Manager is activated, the Java API checks with the security manager for approval before executing a wide range of potentially unsafe operations. The Java Security Manager uses a security policy to determine whether a given action will be allowed or denied.

5.2. ABOUT JAVA SECURITY POLICY

A Java security policy is a set of defined permissions for different classes of code. The Java Security Manager compares actions requested by applications against the security policy. If an action is allowed by the policy, the Security Manager will permit that action to take place. If the action is not allowed by the policy, the Security Manager will deny that action.

IMPORTANT

Previous versions of JBoss EAP defined policies using an external file, e.g. EAP_HOME/bin/server.policy. JBoss EAP 7 defines Java Security Policies in two ways: the security-manager subsystem and through XML files in the individual deployments. The security-manager subsystem defines minimum and maximum permission for ALL deployments, while the XML files specify the permissions requested by the individual deployment.

5.2.1. About defining policies in the security manager subsystem

The security-manager subsystem allows you to define shared or common permissions for all deployments. This is accomplished by defining minimum and maximum permission sets. All deployments will be granted at the least all permissions defined in the minimum permission. The deployment process fails for a deployment if it requests a permission that exceeds the ones defined in the maximum permission set.

Example: Management CLI command for updating minimum permission set

```
/subsystem=security-manager/deployment-permissions=default:write-attribute(name=minimum-permissions, value=[{class="java.util.PropertyPermission", actions="read", name="*"}])
```

Example: Management CLI command for updating maximum permission set

```
/subsystem=security-manager/deployment-permissions=default:write-attribute(name=maximum-permissions, value=[{class="java.util.PropertyPermission", actions="read,write", name="*"}, {class="java.io.FilePermission", actions="read,write", name="/\-"}])
```

NOTE

If the maximum permission set is not defined, its value defaults to java.security.AllPermission.
5.2.2. About defining policies in the deployment

In JBoss EAP 7, you can add a `META-INF/permissions.xml` to your deployment. This file allows you to specify the permissions needed by the deployment.

If a minimum permissions set is defined in the `security-manager` subsystem and a `META-INF/permissions.xml` is added to your deployment, then the union of those permissions is granted. If the permissions requested in the `permissions.xml` exceed the maximum policies defined in the `security-manager` subsystem, its deployment will not succeed. If both `META-INF/permissions.xml` and `META-INF/jboss-permissions.xml` are present in the deployment, then only the permissions requested in the `META-INF/jboss-permissions.xml` are granted.

The specification dictates that `permissions.xml` cover the entire application or top-level deployment module. In cases where you wish to define specific permissions for a subdeployment, you can use the JBoss EAP-specific `META-INF/jboss-permissions.xml`. It follows the same exact format as `permissions.xml` and will apply only to the deployment module in which it is declared.

Example: Sample permissions.xml

```xml
<permissions version="7">
    <permission>
        <class-name>java.util.PropertyPermission</class-name>
        <name>*</name>
        <actions>read</actions>
    </permission>
</permissions>
```

Additional resources

- JSR 342 `META-INF/permissions.xml` file.

5.2.3. About defining policies in modules

You can restrict the permissions of a module by adding a `<permissions>` element to the `module.xml` file. The `<permissions>` element contains zero or more `<grant>` elements, which define the permission to grant to the module. Each `<grant>` element contains the following attributes:

- **permission**
  The qualified class name of the permission to grant.

- **name**
  The permission name to provide to the permission class constructor.

- **actions**
  The (optional) list of actions, required by some permission types.

Example: module.xml with Defined Policies

```xml
<module xmlns="urn:jboss:module:1.5" name="org.jboss.test.example">
    <permissions>
```

If the `<permissions>` element is present, the module will be restricted to only the permissions you have listed. If the `<permissions>` element is not present, there will be no restrictions on the module.

5.3. RUN JBOSS EAP WITH THE JAVA SECURITY MANAGER

You can run JBoss EAP with the Java Security Manager in two different ways. There are two ways to run the Java Security Manager:

- Using the `-secmgr` flag with startup configuration script.
- Using the Startup Configuration File.

**IMPORTANT**

Previous version of JBoss EAP allowed for the use of the `-Djava.security.manager` Java system property as well as custom security managers. Neither of these are supported in JBoss EAP 7. In addition, the Java Security Manager policies are now defined within the `security-manager` subsystem, meaning external policy files and the `-Djava.security.policy` Java system property are not supported JBoss EAP 7.

**IMPORTANT**

Before starting JBoss EAP with the Java Security Manager enabled, you need make sure all security policies are defined in the `security-manager` subsystem.

5.3.1. Using the `-secmgr` flag with startup configuration script.

You can run JBoss EAP with the Java Security Manager. To do this, use the `secmgr` option during startup.

**Procedure**

- Include the `-secmgr` flag when starting up your JBoss EAP instance.

**Example of how to include the `-secmgr` flag**

```
./standalone.sh -secmgr
```

5.3.2. Using the startup configuration file

You can run JBoss EAP with the Java Security Manager. To do this, you have to modify the startup configuration file.

**IMPORTANT**

The domain or standalone server must be completely stopped before you edit any configuration files.
NOTE

If you are using JBoss EAP in a managed domain, you must perform the following procedure on each physical host or instance in your domain.

Procedure

- Enable the Java Security Manager using the startup configuration file, you need to edit either the `standalone.conf` or `domain.conf` file, depending if you are running a standalone instance or managed domain. If running in Windows, the `standalone.conf.bat` or `domain.conf.bat` files are used instead.

  Uncomment the `SECMGR="true"` line in the configuration file:

```
Example standalone.conf or domain.conf

# Uncomment this to run with a security manager enabled
SECMGR="true"

Example standalone.conf.bat or domain.conf.bat

rem # Uncomment this to run with a security manager enabled
set "SECMGR=true"
```

5.4. CONSIDERATIONS BEFORE MOVING FROM PREVIOUS VERSIONS

When moving applications from a previous version of JBoss EAP to JBoss EAP 7 running with the Java Security Manager enabled, you need to be aware of the changes in how policies are defined as well as the necessary configuration needed with both the JBoss EAP configuration and the deployment. Here are the changes that you should be aware of:

- In previous versions of JBoss EAP, policies were defined in an external configuration file. In JBoss EAP 7, policies are defined using the `security-manager` subsystem and with `permissions.xml` or `jboss-permissions.xml` contained in the deployment.

- You could use `-Djava.security.manager` and `-Djava.security.policy` Java system properties during JBoss EAP startup. In previous versions of JBoss EAP, these are no longer supported and the `secmgr` flag should be used instead to enable JBoss EAP to run with the Java Security Manager.

- Custom security managers are not supported in JBoss EAP 7.

Additional resources

- Defining a Java Security Policy
- How to run JBoss EAP with the Java Security Manager
APPENDIX A. REFERENCE MATERIAL

A.1. ELYTRON SUBSYSTEM COMPONENTS REFERENCE

Table A.1. add-prefix-role-mapper Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix</td>
<td>The prefix to add to each role.</td>
</tr>
</tbody>
</table>

Table A.2. add-suffix-role-mapper Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>suffix</td>
<td>The suffix to add to each role.</td>
</tr>
</tbody>
</table>

Table A.3. aggregate-http-server-mechanism-factory Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>http-server-mechanism-factories</td>
<td>The list of HTTP server factories to aggregate.</td>
</tr>
</tbody>
</table>

Table A.4. aggregate-principal-decoder Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>principal-decoders</td>
<td>The list of principal decoders to aggregate.</td>
</tr>
</tbody>
</table>

Table A.5. aggregate-principal-transformer Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>principal-transformers</td>
<td>The list of principal transformers to aggregate.</td>
</tr>
</tbody>
</table>

Table A.6. aggregate-providers Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>providers</td>
<td>The list of referenced Provider resources to aggregate.</td>
</tr>
</tbody>
</table>

Table A.7. aggregate-realm Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>authentication-realm</td>
<td>Reference to the security realm to use for authentication steps. This is used for obtaining or validating credentials.</td>
</tr>
<tr>
<td>authorization-realm</td>
<td>Reference to the security realm to use for loading the identity for authorization steps.</td>
</tr>
<tr>
<td>authorization-realms</td>
<td>Reference to the security realms to aggregate for loading the identity for authorization steps. For information about using multiple authentication realms, see Configure Authentication and Authorization Using Multiple Identity Stores in the How to Configure Identity Management guide.</td>
</tr>
</tbody>
</table>

**NOTE**

The `authorization-realm` and `authorization-realms` attributes are mutually exclusive. Define only one of the two attributes in a realm.

### Table A.8. aggregate-role-mapper Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>role-mappers</td>
<td>The list of role mappers to aggregate.</td>
</tr>
</tbody>
</table>

### Table A.9. aggregate-sasl-server-factory Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sasl-server-factories</td>
<td>The list of SASL server factories to aggregate.</td>
</tr>
</tbody>
</table>

### Table A.10. authentication-configuration Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anonymous</td>
<td>If <code>true</code> anonymous authentication is allowed. The default is <code>false</code>.</td>
</tr>
<tr>
<td>authentication-name</td>
<td>The authentication name to use.</td>
</tr>
<tr>
<td>authorization-name</td>
<td>The authorization name to use.</td>
</tr>
<tr>
<td>credential-reference</td>
<td>The credential to use for authentication. This can be in clear text or as a reference to a credential stored in a <code>credential-store</code>.</td>
</tr>
</tbody>
</table>
Table A.11. authentication-context Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extends</td>
<td>An existing authentication configuration to extend.</td>
</tr>
<tr>
<td>host</td>
<td>The host to use.</td>
</tr>
<tr>
<td>kerberos-security-factory</td>
<td>Reference to a kerberos security factory used to obtain a GSS kerberos credential.</td>
</tr>
<tr>
<td>mechanism-properties</td>
<td>Configuration properties for the SASL authentication mechanism.</td>
</tr>
<tr>
<td>port</td>
<td>The port to use.</td>
</tr>
<tr>
<td>protocol</td>
<td>The protocol to use.</td>
</tr>
<tr>
<td>realm</td>
<td>The realm to use.</td>
</tr>
<tr>
<td>sasl-mechanism-selector</td>
<td>The SASL mechanism selector string. For more information about the grammar required for the sasl-mechanism-selector, see sasl-mechanism-selector Grammar in How to Configure Server Security for JBoss EAP.</td>
</tr>
<tr>
<td>security-domain</td>
<td>Reference to a security domain to obtain a forwarded identity.</td>
</tr>
</tbody>
</table>

Table A.12. authentication-context match-rules Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>authentication-configuration</td>
<td>Reference to the authentication configuration to use for a successful match.</td>
</tr>
<tr>
<td>match-abstract-type</td>
<td>The abstract type to match against.</td>
</tr>
<tr>
<td>match-abstract-type-authority</td>
<td>The abstract type authority to match against.</td>
</tr>
<tr>
<td>match-host</td>
<td>The host to match against.</td>
</tr>
<tr>
<td>match-local-security-domain</td>
<td>The local security domain to match against.</td>
</tr>
</tbody>
</table>
### Table A.13. caching-realm Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>match-no-user</td>
<td>If true, rule will match against no user.</td>
</tr>
<tr>
<td>match-path</td>
<td>The patch to match against.</td>
</tr>
<tr>
<td>match-port</td>
<td>The port to match against.</td>
</tr>
<tr>
<td>match-protocol</td>
<td>The protocol to match against.</td>
</tr>
<tr>
<td>match-urn</td>
<td>The URN to match against.</td>
</tr>
<tr>
<td>match-user</td>
<td>The user to match against.</td>
</tr>
<tr>
<td>ssl-context</td>
<td>Reference to the ssl-context to use for a successful match.</td>
</tr>
</tbody>
</table>

### Table A.14. case-principal-transformer attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>upper-case</td>
<td>An optional attribute that converts a principal transformer’s name to uppercase characters when set as true, which is the default setting. Set the attribute to false to convert the principal transformer’s name to lowercase characters.</td>
</tr>
</tbody>
</table>

### Table A.15. certificate-authority-account Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias</td>
<td>The alias of certificate authority account key in the keystore. If the alias does not already exist in the keystore, a certificate authority account key will be automatically generated and stored as a PrivateKeyEntry under the alias.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>certificate-author</td>
<td>The name of the certificate authority to use. The default, and only allowed value, is <strong>LetsEncrypt</strong>.</td>
</tr>
<tr>
<td>contact-urls</td>
<td>A list of URLs that the certificate authority can contact about any issues related to this account.</td>
</tr>
<tr>
<td>credential-reference</td>
<td>The credential to be used when accessing the certificate authority account key.</td>
</tr>
<tr>
<td>key-store</td>
<td>The keystore that contains the certificate authority account key.</td>
</tr>
</tbody>
</table>

**Table A.16. chained-principal-transformer Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>principal-transformers</td>
<td>List of principal transformers to chain.</td>
</tr>
</tbody>
</table>

**Table A.17. client-ssl-context Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cipher-suite-filter</td>
<td>The filter to apply to specify the enabled cipher suites. This filter takes a list of items delimited by colons, commas, or spaces. Each item may be a OpenSSL–style cipher suite name, a standard SSL/TLS cipher suite name, or a keyword such as <strong>TLSv1.2</strong> or <strong>DES</strong>. A full list of keywords as well as additional details on creating a filter can be found in the Javadoc for the <strong>CipherSuiteSelector</strong> class. The default value is <strong>DEFAULT</strong>, which corresponds to all known cipher suites that do not have <strong>NULL</strong> encryption and excludes any cipher suites that have no authentication.</td>
</tr>
<tr>
<td>key-manager</td>
<td>Reference to the <strong>key-manager</strong> to use within the <strong>SSLContext</strong>.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>protocols</td>
<td>The enabled protocols. Allowed options: SSLv2, SSLv3, TLSv1, TLSv1.1, TLSv1.2, TLSv1.3. This defaults to enabling TLSv1, TLSv1.1, TLSv1.2, and TLSv1.3.</td>
</tr>
<tr>
<td>provider-name</td>
<td>The name of the provider to use. If not specified, all providers from providers will be passed to the SSLContext.</td>
</tr>
<tr>
<td>providers</td>
<td>The name of the providers to obtain the Provider[] to use to load the SSLContext.</td>
</tr>
<tr>
<td>session-timeout</td>
<td>The timeout for SSL sessions.</td>
</tr>
<tr>
<td>trust-manager</td>
<td>Reference to the trust-manager to use within the SSLContext.</td>
</tr>
</tbody>
</table>

**WARNING**

Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.

---

**Table A.18. concatenating-principal-decoder Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>joiner</td>
<td>The string that will be used to join the values in the principal-decoders attribute.</td>
</tr>
<tr>
<td>principal-decoders</td>
<td>The list of principal decoders to concatenate.</td>
</tr>
</tbody>
</table>

**Table A.19. configurable-http-server-mechanism-factory Attributes**
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filters</td>
<td>The list of filters to be applied in order to enable or disable mechanisms based on the name.</td>
</tr>
<tr>
<td>http-server-mechanism-factory</td>
<td>Reference to the http server factory to be wrapped.</td>
</tr>
<tr>
<td>properties</td>
<td>Custom properties to be passed in to the HTTP server factory calls.</td>
</tr>
</tbody>
</table>

Table A.20. configurable-http-server-mechanism-factory filters Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern-filter</td>
<td>Filter based on a regular expression pattern.</td>
</tr>
<tr>
<td>enabling</td>
<td>If true the filter will be enabled if the mechanism matches. This defaults to true.</td>
</tr>
</tbody>
</table>

Table A.21. configurable-sasl-server-factory Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filters</td>
<td>List of filters to be evaluated sequentially and combined using or.</td>
</tr>
<tr>
<td>properties</td>
<td>Custom properties to be passed in to the SASL server factory calls.</td>
</tr>
<tr>
<td>protocol</td>
<td>The protocol passed into the factory when creating the mechanism.</td>
</tr>
<tr>
<td>sasl-server-factory</td>
<td>Reference to the SASL server factory to be wrapped.</td>
</tr>
<tr>
<td>server-name</td>
<td>The server name passed into the factory when creating the mechanism.</td>
</tr>
</tbody>
</table>

Table A.22. configurable-sasl-server-factory filters Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabling</td>
<td>If true the filter will be enabled if the factory matches. This defaults to true.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>predefined-filter</td>
<td>A predefined filter to use to filter the mechanism name. Allowed values are <code>HASH_MD5</code>, <code>HASH_SHA</code>, <code>HASH_SHA_256</code>, <code>HASH_SHA_384</code>, <code>HASH_SHA_512</code>, <code>GS2</code>, <code>SCRAM</code>, <code>DIGEST</code>, <code>IEC_ISO_9798</code>, <code>EAP</code>, <code>MUTUAL</code>, <code>BINDING</code>, and <code>RECOMMENDED</code>.</td>
</tr>
<tr>
<td>pattern-filter</td>
<td>A filter for the mechanism name based on a regular expression.</td>
</tr>
</tbody>
</table>

**Table A.23. constant-permission-mapper Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>permission-sets</td>
<td>The permission sets to assign in the event of a match. Permission sets can be used to assign permissions to an identity.</td>
</tr>
</tbody>
</table>

**NOTE**

The `permissions` attribute is deprecated, and is replaced by `permission-sets`.

**Table A.24. constant-principal-decoder Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>The constant value the principal decoder will always return.</td>
</tr>
</tbody>
</table>

**Table A.25. constant-principal-transformer Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>The constant value this principal transformer will always return.</td>
</tr>
</tbody>
</table>

**Table A.26. constant-realm-mapper Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>realm-name</td>
<td>Reference to the realm that will be returned.</td>
</tr>
</tbody>
</table>

**Table A.27. constant-role-mapper Attributes**
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>roles</td>
<td>The list of roles that will be returned.</td>
</tr>
</tbody>
</table>

**Table A.28. credential-store Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>Specifies whether the credential store should create storage when it does not exist. The default value is <em>false</em>.</td>
</tr>
<tr>
<td>credential-reference</td>
<td>The reference to the credential used to create protection parameter. This can be in clear text or as a reference to a credential stored in a credential-store.</td>
</tr>
<tr>
<td>implementation-properties</td>
<td>Map of credentials store implementation-specific properties.</td>
</tr>
<tr>
<td>modifiable</td>
<td>Whether you can modify the credential store. The default value is <em>true</em>.</td>
</tr>
<tr>
<td>other-providers</td>
<td>The name of the providers to obtain the providers to search for the one that can create the required Jakarta Connectors objects within the credential store. This is valid only for keystore-based credential store. If this is not specified, then the global list of providers is used instead.</td>
</tr>
<tr>
<td>path</td>
<td>The file name of the credential store.</td>
</tr>
<tr>
<td>provider-name</td>
<td>The name of the provider to use to instantiate the CredentialStoreSpi. If the provider is not specified, then the first provider found that can create an instance of the specified type will be used.</td>
</tr>
<tr>
<td>providers</td>
<td>The name of the providers to obtain the providers to search for the one that can create the required credential store type. If this is not specified, then the global list of providers is used instead.</td>
</tr>
<tr>
<td>relative-to</td>
<td>The base path this credential store path is relative to.</td>
</tr>
<tr>
<td>type</td>
<td>Type of the credential store, for example, KeyStoreCredentialStore.</td>
</tr>
</tbody>
</table>

**Table A.29. credential-store alias**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry-type</td>
<td>Type of credential entry stored in the credential store.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>secret-value</td>
<td>Secret value such as password.</td>
</tr>
</tbody>
</table>

Table A.30. credential-store KeyStoreCredentialStore implementation properties

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cryptoAlg</td>
<td>Cryptographic algorithm name to be used to encrypt decrypt entries at external storage. This attribute is only valid if external is enabled. Defaults to AES.</td>
</tr>
<tr>
<td>external</td>
<td>Whether data is stored to external storage and encrypted by the keyAlias. Defaults to false.</td>
</tr>
<tr>
<td>externalPath</td>
<td>Specifies path to external storage. This attribute is only valid if external is enabled.</td>
</tr>
<tr>
<td>keyAlias</td>
<td>The secret key alias within the credential store that is used to encrypt or decrypt data to the external storage.</td>
</tr>
<tr>
<td>keyStoreType</td>
<td>The keystore type, such as PKCS11. Defaults to KeyStore.getDefaultType().</td>
</tr>
</tbody>
</table>

Table A.31. custom-credential-security-factory Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>The class name of the implementation of the custom security factory.</td>
</tr>
<tr>
<td>configuration</td>
<td>The optional key and value configuration for the custom security factory.</td>
</tr>
<tr>
<td>module</td>
<td>The module to use to load the custom security factory.</td>
</tr>
</tbody>
</table>

Table A.32. custom-modifiable-realm Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>The class name of the implementation of the custom realm.</td>
</tr>
<tr>
<td>configuration</td>
<td>The optional key and value configuration for the custom realm.</td>
</tr>
<tr>
<td>module</td>
<td>The module to use to load the custom realm.</td>
</tr>
</tbody>
</table>

Table A.33. custom-permission-mapper Attributes
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class-name</td>
<td>Fully qualified class name of the permission mapper.</td>
</tr>
<tr>
<td>configuration</td>
<td>The optional key and value configuration for the permission mapper.</td>
</tr>
<tr>
<td>module</td>
<td>Name of the module to use to load the permission mapper.</td>
</tr>
</tbody>
</table>

Table A.34. custom-principal-decoder Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class-name</td>
<td>Fully qualified class name of the principal decoder.</td>
</tr>
<tr>
<td>configuration</td>
<td>The optional key and value configuration for the principal decoder.</td>
</tr>
<tr>
<td>module</td>
<td>Name of the module to use to load the principal decoder.</td>
</tr>
</tbody>
</table>

Table A.35. custom-principal-transformer Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class-name</td>
<td>Fully qualified class name of the principal transformer.</td>
</tr>
<tr>
<td>configuration</td>
<td>The optional key and value configuration for the principal transformer.</td>
</tr>
<tr>
<td>module</td>
<td>Name of the module to use to load the principal transformer.</td>
</tr>
</tbody>
</table>

Table A.36. custom-realm Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class-name</td>
<td>Fully qualified class name of the custom realm.</td>
</tr>
<tr>
<td>configuration</td>
<td>The optional key and value configuration for the custom realm.</td>
</tr>
<tr>
<td>module</td>
<td>Name of the module to use to load the custom realm.</td>
</tr>
</tbody>
</table>

Table A.37. custom-realm-mapper Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class-name</td>
<td>Fully qualified class name of the realm mapper.</td>
</tr>
</tbody>
</table>
### Table A.38. custom-role-decoder Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class-name</td>
<td>Fully qualified class name of the role decoder.</td>
</tr>
<tr>
<td>configuration</td>
<td>The optional key and value configuration for the role decoder.</td>
</tr>
<tr>
<td>module</td>
<td>Name of the module to use to load the role decoder.</td>
</tr>
</tbody>
</table>

### Table A.39. custom-role-mapper Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class-name</td>
<td>Fully qualified class name of the role mapper.</td>
</tr>
<tr>
<td>configuration</td>
<td>The optional key and value configuration for the role mapper.</td>
</tr>
<tr>
<td>module</td>
<td>Name of the module to use to load the role mapper.</td>
</tr>
</tbody>
</table>

### Table A.40. dir-context Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>authentication-context</td>
<td>The authentication context to obtain login credentials to connect to the LDAP server. Can be omitted if authentication-level is none, which is equivalent to anonymous authentication.</td>
</tr>
<tr>
<td>authentication-level</td>
<td>The authentication level, meaning security level or authentication mechanism, to use. Corresponds to SECURITY_AUTHENTICATION or java.naming.security.authentication environment property. Allowed values are none, simple and sasl_mech format. The sasl_mech format is a space-separated list of SASL mechanism names.</td>
</tr>
<tr>
<td>connection-timeout</td>
<td>The timeout for connecting to the LDAP server in milliseconds.</td>
</tr>
<tr>
<td>credential-reference</td>
<td>The credential reference to authenticate and connect to the LDAP server. This can be omitted if authentication-level is none, which is equivalent to anonymous authentication.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>enable-connection-pooling</td>
<td>If true connection pooling is enabled. This defaults to false.</td>
</tr>
<tr>
<td>module</td>
<td>Name of module that will be used as the class loading base.</td>
</tr>
<tr>
<td>principal</td>
<td>The principal to authenticate and connect to the LDAP server. This can be omitted if authentication-level is none which is equivalent to anonymous authentication.</td>
</tr>
<tr>
<td>properties</td>
<td>The additional connection properties for the DirContext.</td>
</tr>
<tr>
<td>read-timeout</td>
<td>The read timeout for an LDAP operation in milliseconds.</td>
</tr>
<tr>
<td>referral-mode</td>
<td>The mode used to determine if referrals should be followed. Allowed values are FOLLOW, IGNORE, and THROW. This defaults to IGNORE.</td>
</tr>
<tr>
<td>ssl-context</td>
<td>The name of the SSL context used to secure connection to the LDAP server.</td>
</tr>
<tr>
<td>url</td>
<td>The connection URL.</td>
</tr>
</tbody>
</table>

**Table A.41. expression=encryption Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default-resolver</td>
<td>Optional attribute. The resolver to use when an encrypted expression is defined without one. For example if you set &quot;exampleResolver&quot; as the default-resolver and you create an encrypted expression with the command /subsystem=elytron/expression=encryption:create-expression(clear-text=TestPassword), Elytron uses &quot;exampleResolver&quot; as the resolver for this encrypted expression.</td>
</tr>
<tr>
<td>prefix</td>
<td>The prefix to use within an encrypted expression. Default is ENC. This attribute is provided for those cases where ENC might already be defined. You shouldn’t change this value unless it conflicts with an already defined ENC prefix.</td>
</tr>
</tbody>
</table>
### resolvers

A list of defined resolvers. A resolver has the following attributes:

- **name** - The name of the individual configuration used to reference it.
- **credential-store** - Reference to the credential store instance that contains the secret key this resolver uses.
- **secret-key** - The alias of the secret key Elytron should use from within a given credential store.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resolvers</td>
<td>A list of defined resolvers. A resolver has the following attributes:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>encoded</td>
<td>Whether the identity names should be stored encoded (Base32) in file names.</td>
</tr>
<tr>
<td>levels</td>
<td>The number of levels of directory hashing to apply. The default value is 2.</td>
</tr>
<tr>
<td>path</td>
<td>The path to the file containing the realm.</td>
</tr>
<tr>
<td>relative-to</td>
<td>The predefined relative path to use with <code>path</code>. For example <code>jboss.server.config.dir</code>.</td>
</tr>
</tbody>
</table>

### Table A.42. filesystem-realm Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>encoded</td>
<td>Whether the identity names should be stored encoded (Base32) in file names.</td>
</tr>
<tr>
<td>levels</td>
<td>The number of levels of directory hashing to apply. The default value is 2.</td>
</tr>
<tr>
<td>path</td>
<td>The path to the file containing the realm.</td>
</tr>
<tr>
<td>relative-to</td>
<td>The predefined relative path to use with <code>path</code>. For example <code>jboss.server.config.dir</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias-filter</td>
<td>A filter to apply to the aliases returned from the <code>key-store</code>. It can either be a comma-separated list of aliases to return or one of the following formats:</td>
</tr>
<tr>
<td></td>
<td>- ALL:<code>-alias1:</code>-alias2</td>
</tr>
<tr>
<td></td>
<td>- NONE:<code>+alias1:</code>+alias2</td>
</tr>
</tbody>
</table>

- **NOTE**

  The `alias-filter` attribute is case sensitive. Because the use of mixed-case or uppercase aliases, such as `elytronAppServer`, might not be recognized by some keystore providers, it is recommended to use lowercase aliases, such as `elytronappserver`.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key-store</td>
<td>Reference to the <code>key-store</code> to filter.</td>
</tr>
</tbody>
</table>
### Table A.44. generate-key-pair attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithm</td>
<td>Specifies the encryption algorithm, such as RSA, DSA, or EC. The default value is RSA.</td>
</tr>
<tr>
<td>size</td>
<td>Specifies the size of the private key in bits. The default size values in bits for the key pair types are as follows: RSA is 2048, DSA is 2048, and EC is 256.</td>
</tr>
</tbody>
</table>

### Table A.45. http-authentication-factory Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>http-server-mechanism-factory</td>
<td>The <code>HttpServerAuthenticationMechanismFactory</code> to associate with this resource.</td>
</tr>
<tr>
<td>mechanism-configurations</td>
<td>The list of mechanism-specific configurations.</td>
</tr>
<tr>
<td>security-domain</td>
<td>The security domain to associate with this resource.</td>
</tr>
</tbody>
</table>

### Table A.46. http-authentication-factory mechanism-configurations Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>credential-security-factory</td>
<td>The security factory to use to obtain a credential as required by the mechanism.</td>
</tr>
<tr>
<td>final-principal-transformer</td>
<td>A final principal transformer to apply for this mechanism realm.</td>
</tr>
<tr>
<td>host-name</td>
<td>The host name this configuration applies to.</td>
</tr>
<tr>
<td>mechanism-name</td>
<td>This configuration will only apply where a mechanism with the name specified is used. If this attribute is omitted then this will match any mechanism name.</td>
</tr>
<tr>
<td>mechanism-realm-configurations</td>
<td>The list of definitions of the realm names as understood by the mechanism.</td>
</tr>
<tr>
<td>pre-realm-principal-transformer</td>
<td>A principal transformer to apply before the realm is selected.</td>
</tr>
<tr>
<td>post-realm-principal-transformer</td>
<td>A principal transformer to apply after the realm is selected.</td>
</tr>
<tr>
<td>protocol</td>
<td>The protocol this configuration applies to.</td>
</tr>
<tr>
<td>realm-mapper</td>
<td>The realm mapper to be used by the mechanism.</td>
</tr>
</tbody>
</table>
### Table A.47. http-authentication-factory mechanism-configurations mechanism-realm-configurations Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>final-principal-transformer</td>
<td>A final principal transformer to apply for this mechanism realm.</td>
</tr>
<tr>
<td>post-realm-principal-transformer</td>
<td>A principal transformer to apply after the realm is selected.</td>
</tr>
<tr>
<td>pre-realm-principal-transformer</td>
<td>A principal transformer to apply before the realm is selected.</td>
</tr>
<tr>
<td>realm-mapper</td>
<td>The realm mapper to be used by the mechanism.</td>
</tr>
<tr>
<td>realm-name</td>
<td>The name of the realm to be presented by the mechanism.</td>
</tr>
</tbody>
</table>

### Table A.48. identity-realm Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute-name</td>
<td>The name of the attribute associated with this identity.</td>
</tr>
<tr>
<td>attribute-values</td>
<td>The list of values associated with the identities attribute.</td>
</tr>
<tr>
<td>identity</td>
<td>The identity available from the security realm.</td>
</tr>
</tbody>
</table>

### Table A.49. import-key-pair attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key-passphrase</td>
<td>Optional attribute. Sets the passphrase to decrypt the private key.</td>
</tr>
<tr>
<td>private-key-location</td>
<td>The path to a file containing a private key. Only specify if you have not already specified the <code>private-key-string</code> attribute.</td>
</tr>
<tr>
<td>private-key-string</td>
<td>Sets the private key as a string. Only specify if you have not already specified the <code>private-key-location</code> attribute.</td>
</tr>
<tr>
<td>public-key-location</td>
<td>Required if private key is in any format other than OpenSSH. The path to a file containing a public key. Only specify if you have not already specified the <code>public-key-string</code> attribute.</td>
</tr>
<tr>
<td>public-key-string</td>
<td>Required if private key is in any format other than OpenSSH. Sets the public key as a string. Only specify if you have not already specified the <code>public-key-location</code> attribute.</td>
</tr>
</tbody>
</table>

### Table A.50. jaspi-configuration Attributes
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>application-context</td>
<td>Used when registering this configuration with the AuthConfigFactory. Can be omitted to allow wildcard matching.</td>
</tr>
<tr>
<td>description</td>
<td>Is used to provide a description to the AuthConfigFactory.</td>
</tr>
<tr>
<td>layer</td>
<td>Used when registering this configuration with the AuthConfigFactory. Can be omitted to allow wildcard matching.</td>
</tr>
<tr>
<td>name</td>
<td>A name that allows the resource to be referenced in the management model.</td>
</tr>
</tbody>
</table>

Table A.51. jaspi-configuration server-auth-module Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class-name</td>
<td>The fully qualified class name of the ServerAuthModule.</td>
</tr>
<tr>
<td>flag</td>
<td>The control flag to indicate how this module operates in relation to the other modules.</td>
</tr>
<tr>
<td>module</td>
<td>The module to load the ServerAuthModule from.</td>
</tr>
<tr>
<td>options</td>
<td>Configuration options to be passed into the ServerAuthModule on initialization.</td>
</tr>
</tbody>
</table>

Table A.52. jdbc-realm Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>principal-query</td>
<td>The list of authentication queries used to authenticate users based on specific key types.</td>
</tr>
</tbody>
</table>

Table A.53. jdbc-realm principal-query Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute-mapping</td>
<td>The list of attribute mappings defined for this resource.</td>
</tr>
<tr>
<td>bcrypt-mapper</td>
<td>A key mapper that maps a column returned from a SQL query to a Bcrypt key type.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>clear-password-mapper</td>
<td>A key mapper that maps a column returned from a SQL query to a clear password key type. This has a <code>password-index</code> child element that is the column index from an authentication query that represents the user password.</td>
</tr>
<tr>
<td>data-source</td>
<td>The name of the datasource used to connect to the database.</td>
</tr>
<tr>
<td>salted-simple-digest-mapper</td>
<td>A key mapper that maps a column returned from a SQL query to a <strong>Salted Simple Digest</strong> key type.</td>
</tr>
<tr>
<td>scram-mapper</td>
<td>A key mapper that maps a column returned from a SQL query to a <strong>SCRAM</strong> key type.</td>
</tr>
<tr>
<td>simple-digest-mapper</td>
<td>A key mapper that maps a column returned from a SQL query to a <strong>Simple Digest</strong> key type.</td>
</tr>
<tr>
<td>sql</td>
<td>The SQL statement used to obtain the keys as table columns for a specific user and map them accordingly with their type.</td>
</tr>
</tbody>
</table>

**Table A.54. jdbc-realm principal-query attribute-mapping Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>The column index from a query that representing the mapped attribute.</td>
</tr>
<tr>
<td>to</td>
<td>The name of the identity attribute mapped from a column returned from a SQL query.</td>
</tr>
</tbody>
</table>

**Table A.55. jdbc-realm principal-query bcrypt-mapper Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iteration-count-index</td>
<td>The column index from an authentication query that represents the password’s iteration count, if supported.</td>
</tr>
<tr>
<td>password-index</td>
<td>The column index from an authentication query that represents the user password.</td>
</tr>
<tr>
<td>salt-index</td>
<td>The column index from an authentication query that represents the password’s salt, if supported.</td>
</tr>
</tbody>
</table>

**Table A.56. jdbc-realm principal-query salted-simple-digest-mapper Attributes**
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>password-index</td>
<td>The column index from an authentication query that represents the user password.</td>
</tr>
<tr>
<td>salt-index</td>
<td>The column index from an authentication query that represents the password’s salt, if supported.</td>
</tr>
</tbody>
</table>

### Table A.57. jdbc-realm principal-query simple-digest-mapper Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithm</td>
<td>The algorithm for a specific password key mapper. Allowed values are simple-digest-md2, simple-digest-md5, simple-digest-sha-1, simple-digest-sha-256, simple-digest-sha-384, and simple-digest-sha-512. The default is simple-digest-md5.</td>
</tr>
<tr>
<td>password-index</td>
<td>The column index from an authentication query that represents the user password.</td>
</tr>
</tbody>
</table>

### Table A.58. jdbc-realm principal-query scram-mapper Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithm</td>
<td>The algorithm for a specific password key mapper. The allowed values are scram-sha-1 and scram-sha-256. The default value is scram-sha-256.</td>
</tr>
<tr>
<td>iteration-count-index</td>
<td>The column index from an authentication query that represents the password’s iteration count, if supported.</td>
</tr>
<tr>
<td>password-index</td>
<td>The column index from an authentication query that represents the user password.</td>
</tr>
<tr>
<td>salt-index</td>
<td>The column index from an authentication query that represents the password’s salt, if supported.</td>
</tr>
</tbody>
</table>

### Table A.59. kerberos-security-factory Attributes
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>If true the JAAS step of obtaining the credential will have debug logging enabled. Defaults to false.</td>
</tr>
<tr>
<td>mechanism-names</td>
<td>The mechanism names the credential should be usable with. Names will be converted to OIDs and used together with OIDs from mechanism-oids attribute.</td>
</tr>
<tr>
<td>mechanism-oids</td>
<td>The list of mechanism OIDs the credential should be usable with.</td>
</tr>
<tr>
<td>minimum-remaining-lifetime</td>
<td>The amount of time in seconds a cached credential can have before it is recreated.</td>
</tr>
<tr>
<td>obtain-kerberos-ticket</td>
<td>Should the KerberosTicket also be obtained and associated with the credential. This is required to be true where credentials are delegated to the server.</td>
</tr>
<tr>
<td>options</td>
<td>The Krb5LoginModule additional options.</td>
</tr>
<tr>
<td>path</td>
<td>The path of the keytab to load to obtain the credential.</td>
</tr>
<tr>
<td>principal</td>
<td>The principal represented by the keytab.</td>
</tr>
<tr>
<td>relative-to</td>
<td>The relative path to the keytab.</td>
</tr>
<tr>
<td>request-lifetime</td>
<td>How much lifetime should be requested for newly created credentials.</td>
</tr>
<tr>
<td>required</td>
<td>Whether the keytab file with an adequate principal is required to exist at the time the service starts.</td>
</tr>
<tr>
<td>server</td>
<td>If true this factory is used for the server-side portion of Kerberos authentication. If false it is used for the client-side. Defaults to true.</td>
</tr>
<tr>
<td>wrap-gss-credential</td>
<td>Whether generated GSS credentials should be wrapped to prevent improper disposal.</td>
</tr>
</tbody>
</table>

Table A.60. key-manager Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithm</td>
<td>The name of the algorithm to use to create the underlying KeyManagerFactory. This is provided by the JDK. For example, a JDK that uses SunJSSE provides the PKIX and SunX509 algorithms. More details on SunJSSE can be found in the Java Secure Socket Extension (JSSE) Reference Guide.</td>
</tr>
</tbody>
</table>
## key-store Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias-filter</td>
<td>A filter to apply to the aliases returned from the keystore. This can either be a comma-separated list of aliases to return or one of the following formats:</td>
</tr>
<tr>
<td></td>
<td>- <strong>ALL:</strong>-alias1:-alias2</td>
</tr>
<tr>
<td></td>
<td>- <strong>NONE:</strong>+alias1:+alias2</td>
</tr>
<tr>
<td>credential-reference</td>
<td>The credential reference to decrypt keystore item. This can be specified in clear text or as a reference to a credential stored in a <strong>credential-store</strong>. This is not a password of the keystore.</td>
</tr>
<tr>
<td>key-store</td>
<td>Reference to the <strong>key-store</strong> to use to initialize the underlying <strong>KeyManagerFactory</strong>.</td>
</tr>
<tr>
<td>provider-name</td>
<td>The name of the provider to use to create the underlying <strong>KeyManagerFactory</strong>.</td>
</tr>
<tr>
<td>providers</td>
<td>Reference to obtain the <strong>Provider[]</strong> to use when creating the underlying <strong>KeyManagerFactory</strong>.</td>
</tr>
</tbody>
</table>

### Table A.61. key-store Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias-filter</td>
<td>A filter to apply to the aliases returned from the keystore, can either be a comma-separated list of aliases to return or one of the following formats:</td>
</tr>
<tr>
<td></td>
<td>- <strong>ALL:</strong>-alias1:-alias2</td>
</tr>
<tr>
<td></td>
<td>- <strong>NONE:</strong>+alias1:+alias2</td>
</tr>
<tr>
<td>credential-reference</td>
<td>The password to use to access the keystore. This can be specified in clear text or as a reference to a credential stored in a <strong>credential-store</strong>.</td>
</tr>
<tr>
<td>path</td>
<td>The path to the keystore file.</td>
</tr>
</tbody>
</table>

**NOTE**

The **alias-filter** attribute is case sensitive. Because the use of mixed-case or uppercase aliases, such as **elytronAppServer**, might not be recognized by some keystore providers, it is recommended to use lowercase aliases, such as **elytronappserver**.
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>provider-name</td>
<td>The name of the provider to use to load the keystore. Setting this attribute disables searching for the first provider that can create a keystore of the specified type.</td>
</tr>
<tr>
<td>providers</td>
<td>A reference to the providers that should be used to obtain the list of provider instances to search. If not specified, the global list of providers will be used instead.</td>
</tr>
<tr>
<td>relative-to</td>
<td>The base path this store is relative to. This can be a full path or predefined path such as <code>jboss.server.config.dir</code>.</td>
</tr>
<tr>
<td>required</td>
<td>If true the keystore file referenced is required to exist at the time the keystore service starts. The default value is false.</td>
</tr>
<tr>
<td>type</td>
<td>The type of the keystore, for example, JKS.</td>
</tr>
</tbody>
</table>

**NOTE**

The following keystore types are detected automatically:

- JKS
- JCEKS
- PKCS12
- BKS
- BCFKS
- UBER

You must manually specify the other keystore types.

A full list of keystore types can be found in the [Java Cryptography Architecture Standard Algorithm Name Documentation for JDK 8](https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardAlgorithmNames.html).

### Table A.62. key-store-realm Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key-store</td>
<td>Reference to the keystore used to back this security realm.</td>
</tr>
</tbody>
</table>

### Table A.63. ldap-key-store Attributes
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias-attribute</td>
<td>The name of LDAP attribute where the item alias will be stored.</td>
</tr>
<tr>
<td>certificate-attribute</td>
<td>The name of LDAP attribute where the certificate will be stored.</td>
</tr>
<tr>
<td>certificate-chain-attribute</td>
<td>The name of LDAP attribute where the certificate chain will be stored.</td>
</tr>
<tr>
<td>certificate-chain-encoding</td>
<td>The encoding of the certificate chain.</td>
</tr>
<tr>
<td>certificate-type</td>
<td>The type of the certificate.</td>
</tr>
<tr>
<td>dir-context</td>
<td>The name of the dir-context which will be used to communication with LDAP server.</td>
</tr>
<tr>
<td>filter-alias</td>
<td>The LDAP filter for obtaining an item in the keystore by alias.</td>
</tr>
<tr>
<td>filter-certificate</td>
<td>The LDAP filter for obtaining an item in the keystore by certificate.</td>
</tr>
<tr>
<td>filter-iterate</td>
<td>The LDAP filter for iterating over all items of the keystore.</td>
</tr>
<tr>
<td>key-attribute</td>
<td>The name of LDAP attribute where the key will be stored.</td>
</tr>
<tr>
<td>key-type</td>
<td>The type of keystore that is stored in a serialized manner in the LDAP attribute. For example, JKS. A full list of keystore types can be found in the Java Cryptography Architecture Standard Algorithm Name Documentation for JDK 8.</td>
</tr>
<tr>
<td>new-item-template</td>
<td>Configuration for item creation. This defines how the LDAP entry of newly created keystore item will look.</td>
</tr>
<tr>
<td>search-path</td>
<td>The path in LDAP where the keystore items will be searched.</td>
</tr>
<tr>
<td>search-recursive</td>
<td>If the LDAP search should be recursive.</td>
</tr>
<tr>
<td>search-time-limit</td>
<td>The time limit in milliseconds for obtaining keystore items from LDAP. Defaults to 10000.</td>
</tr>
</tbody>
</table>

Table A.64. ldap-key-store new-item-template Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-item-attributes</td>
<td>The LDAP attributes which will be set for newly created items. This takes a list of items with name and value pairs.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>new-item-path</td>
<td>The path in LDAP where the newly created keystore items will be stored.</td>
</tr>
<tr>
<td>new-item-rdn</td>
<td>The name of LDAP RDN for the newly created items.</td>
</tr>
</tbody>
</table>

Table A.65. ldap-realm Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow-blank-password</td>
<td>Whether this realm supports blank password direct verification. A blank password attempt will be rejected otherwise.</td>
</tr>
<tr>
<td>dir-context</td>
<td>The name of the <code>dir-context</code> which will be used to connect to the LDAP server.</td>
</tr>
<tr>
<td>direct-verification</td>
<td>If <code>true</code> this realm supports verification of credentials by directly connecting to LDAP as the account being authenticated; otherwise, the password is retrieved from the LDAP server and verified in JBoss EAP. If enabled, the JBoss EAP server must be able to obtain the plain user password from the client, which requires either the <code>PLAIN</code> SASL or <code>BASIC</code> HTTP mechanism be used for authentication. Defaults to <code>false</code>.</td>
</tr>
<tr>
<td>identity-mapping</td>
<td>The configuration options that define how principals are mapped to their corresponding entries in the underlying LDAP server.</td>
</tr>
</tbody>
</table>

Table A.66. ldap-realm identity-mapping Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute-mapping</td>
<td>List of attribute mappings defined for this resource.</td>
</tr>
<tr>
<td>filter-name</td>
<td>The LDAP filter for getting identity by name.</td>
</tr>
<tr>
<td>iterator-filter</td>
<td>The LDAP filter for iterating over identities of the realm.</td>
</tr>
<tr>
<td>new-identity-attributes</td>
<td>The list of attributes of newly created identities and is required for modifiability of the realm. This is a list of name and value pair objects.</td>
</tr>
<tr>
<td>otp-credential-mapper</td>
<td>The credential mapping for OTP credential.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>rdn-identifier</td>
<td>The RDN part of the principal’s DN to be used to obtain the principal’s name from an LDAP entry. This is also used when creating new identities.</td>
</tr>
<tr>
<td>search-base-dn</td>
<td>The base DN to search for identities.</td>
</tr>
<tr>
<td>use-recursive-search</td>
<td>If true identity search queries are recursive. Defaults to false.</td>
</tr>
<tr>
<td>user-password-mapper</td>
<td>The credential mapping for a credential similar to userPassword.</td>
</tr>
<tr>
<td>x509-credential-mapper</td>
<td>The configuration allowing to use LDAP as storage of X509 credentials. If none of the -from child attributes are defined, then this configuration will be ignored. If more than one -from child attribute is defined, then the user certificate must match all the defined criteria.</td>
</tr>
</tbody>
</table>

Table A.67. ldap-realm identity-mapping attribute-mapping Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract-rdn</td>
<td>The RDN key to use as the value for an attribute, in case the value in its raw form is in X.500 format.</td>
</tr>
<tr>
<td>filter</td>
<td>The filter to use to obtain the values for a specific attribute.</td>
</tr>
<tr>
<td>filter-base-dn</td>
<td>The name of the context where the filter should be performed.</td>
</tr>
<tr>
<td>from</td>
<td>The name of the LDAP attribute to map to an identity attribute. If not defined, DN of entry is used.</td>
</tr>
<tr>
<td>reference</td>
<td>The name of LDAP attribute containing DN of entry to obtain value from.</td>
</tr>
<tr>
<td>role-recursion</td>
<td>Maximum depth for recursive role assignment. Use 0 to specify no recursion. Defaults to 0.</td>
</tr>
<tr>
<td>role-recursion-name</td>
<td>Determine the LDAP attribute of role entry which will be a substitute for &quot;{0}&quot; in filter-name when searching roles of role.</td>
</tr>
<tr>
<td>search-recursive</td>
<td>If true attribute LDAP search queries are recursive. Defaults to true.</td>
</tr>
</tbody>
</table>
The name of the identity attribute mapped from a specific LDAP attribute. If not provided, the name of the attribute is the same as define in `from`. If the `from` is not defined too, value `dn` is used.

### Table A.68. ldap-realm identity-mapping user-password-mapper Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>The name of the LDAP attribute to map to an identity attribute. If not defined, DN of entry is used.</td>
</tr>
<tr>
<td>verifiable</td>
<td>If <code>true</code> password can be used to verify the user. Defaults to <code>true</code>.</td>
</tr>
<tr>
<td>writable</td>
<td>If <code>true</code> password can be changed. Defaults to <code>false</code>.</td>
</tr>
</tbody>
</table>

### Table A.69. ldap-realm identity-mapping otp-credential-mapper Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithm-from</td>
<td>The name of the LDAP attribute of OTP algorithm.</td>
</tr>
<tr>
<td>hash-from</td>
<td>The name of the LDAP attribute of OTP hash function.</td>
</tr>
<tr>
<td>seed-from</td>
<td>The name of the LDAP attribute of OTP seed.</td>
</tr>
<tr>
<td>sequence-from</td>
<td>The name of the LDAP attribute of OTP sequence number.</td>
</tr>
</tbody>
</table>

### Table A.70. ldap-realm identity-mapping x509-credential-mapper Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>certificate-from</td>
<td>The name of the LDAP attribute to map to an encoded user certificate. If not defined, encoded certificate will not be checked.</td>
</tr>
<tr>
<td>digest-algorithm</td>
<td>The digest algorithm, which is the hash function, used to compute digest of the user certificate. Will be used only if <code>digest-from</code> has been defined.</td>
</tr>
<tr>
<td>digest-from</td>
<td>The name of the LDAP attribute to map to a user certificate digest. If not defined, certificate digest will not be checked.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>serial-number-from</td>
<td>The name of the LDAP attribute to map to a serial number of user certificate. If not defined, serial number will not be checked.</td>
</tr>
<tr>
<td>subject-dn-from</td>
<td>The name of the LDAP attribute to map to a subject DN of user certificate. If not defined, subject DN will not be checked.</td>
</tr>
</tbody>
</table>

**Table A.71. logical-permission-mapper Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>Reference to the permission mapper to use to the left of the operation.</td>
</tr>
<tr>
<td>logical-operation</td>
<td>The logical operation to use to combine the permission mappers. Allowed values are <strong>and</strong>, <strong>or</strong>, <strong>xor</strong>, and <strong>unless</strong>.</td>
</tr>
<tr>
<td>right</td>
<td>Reference to the permission mapper to use to the right of the operation.</td>
</tr>
</tbody>
</table>

**Table A.72. logical-role-mapper Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>Reference to a role mapper to be used on the left side of the operation.</td>
</tr>
<tr>
<td>logical-operation</td>
<td>The logical operation to be performed on the role mapper mappings. Allowed values are: <strong>and</strong>, <strong>minus</strong>, <strong>or</strong>, and <strong>xor</strong>.</td>
</tr>
<tr>
<td>right</td>
<td>Reference to a role mapper to be used on the right side of the operation.</td>
</tr>
</tbody>
</table>

**Table A.73. mapped-regex-realm-mapper Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delegate-realm-mapper</td>
<td>The realm mapper to delegate to if there is no match using the pattern.</td>
</tr>
<tr>
<td>pattern</td>
<td>The regular expression which must contain at least one capture group to extract the realm from the name.</td>
</tr>
<tr>
<td>realm-map</td>
<td>Mapping of realm name extracted using the regular expression to a defined realm name.</td>
</tr>
</tbody>
</table>
Table A.74. mechanism-provider-filtering-sasl-server-factory Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabling</td>
<td>If true no provider loaded mechanisms are enabled unless matched by one of the filters. This defaults to true.</td>
</tr>
<tr>
<td>filters</td>
<td>The list of filters to apply when comparing the mechanisms from the providers. A filter matches when all of the specified values match the mechanism and provider pair.</td>
</tr>
<tr>
<td>sasl-server-factory</td>
<td>Reference to a SASL server factory to be wrapped by this definition.</td>
</tr>
</tbody>
</table>

Table A.75. mechanism-provider-filtering-sasl-server-factory filters Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mechanism-name</td>
<td>The name of the SASL mechanism this filter matches with.</td>
</tr>
<tr>
<td>provider-name</td>
<td>The name of the provider this filter matches.</td>
</tr>
<tr>
<td>provider-version</td>
<td>The version to use when comparing the provider’s version.</td>
</tr>
<tr>
<td>version-comparison</td>
<td>The equality to use when evaluating the Provider’s version. The allowed values are less-than and greater-than. The default value is less-than.</td>
</tr>
</tbody>
</table>

Table A.76. online-certificate-status-protocol Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>responder</td>
<td>Override the OCSP Responder URI resolved from the certificate.</td>
</tr>
<tr>
<td>responder-certificate</td>
<td>Alias for responder certificate located in responder-keystore or trust-manager keystore if responder-keystore is not defined.</td>
</tr>
<tr>
<td>responder-keystore</td>
<td>Alternative keystore for responder certificate. responder-certificate must be defined.</td>
</tr>
<tr>
<td>prefer-crls</td>
<td>When both OCSP and CRL mechanisms are configured, OCSP mechanism is called first. When prefer-crls is set to true, the CRL mechanism is called first.</td>
</tr>
</tbody>
</table>

Table A.77. permission-set permission Attributes
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>The action to pass to the permission as it is constructed.</td>
</tr>
<tr>
<td>class-name</td>
<td>The fully qualified class name of the permission.</td>
</tr>
<tr>
<td>module</td>
<td>The module to use to load the permission.</td>
</tr>
<tr>
<td>target-name</td>
<td>The target name to pass to the permission as it is constructed.</td>
</tr>
</tbody>
</table>

Table A.78. periodic-rotating-file-audit-log Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoflush</td>
<td>Specifies if the output stream requires flushing after every audit event. If you do not define the attribute, the value of the synchronized is the default value.</td>
</tr>
<tr>
<td>format</td>
<td>Use SIMPLE for human readable text format, or JSON for storing individual events in JSON.</td>
</tr>
<tr>
<td>path</td>
<td>Defines the location of the log files.</td>
</tr>
<tr>
<td>relative-to</td>
<td>Optional attribute. Defines the location of the log files.</td>
</tr>
<tr>
<td>suffix</td>
<td>Optional attribute. Adds a date suffix to a rotated log. You must use the java.time.format.DateTimeFormatter format. For example yyyy-MM-dd.</td>
</tr>
<tr>
<td>synchronized</td>
<td>Default value is true. Specifies that the file descriptor gets synchronized after every audit event.</td>
</tr>
</tbody>
</table>

Table A.79. properties-realm Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>groups-attribute</td>
<td>The name of the attribute in the returned AuthorizationIdentity that should contain the group membership information for the identity.</td>
</tr>
<tr>
<td>groups-properties</td>
<td>The properties file containing the users and their groups.</td>
</tr>
<tr>
<td>users-properties</td>
<td>The properties file containing the users and their passwords.</td>
</tr>
</tbody>
</table>

Table A.80. properties-realm users-properties Attributes
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>digest-realm-name</td>
<td>The default realm name to use for digested passwords if one is not discovered in the properties file.</td>
</tr>
<tr>
<td>path</td>
<td>The path to the file containing the users and their passwords. The file should contain realm name declaration.</td>
</tr>
<tr>
<td>plain-text</td>
<td>If true the passwords in properties file stored in plain text. If false they are pre-hashed, taking the form \texttt{HEX(MD5(username&quot;:&quot;realm&quot;:&quot;password))}. Defaults to false.</td>
</tr>
<tr>
<td>relative-to</td>
<td>The predefined path the path is relative to.</td>
</tr>
</tbody>
</table>

Table A.81. properties-realm groups-properties Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>The path to the file containing the users and their groups.</td>
</tr>
<tr>
<td>relative-to</td>
<td>The predefined path the path is relative to.</td>
</tr>
</tbody>
</table>

Table A.82. provider-http-server-mechanism-factory Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>providers</td>
<td>The providers to use to locate the factories. If not specified, the globally registered list of providers will be used.</td>
</tr>
</tbody>
</table>

Table A.83. provider-loader Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>An argument to be passed into the constructor as the Provider is instantiated.</td>
</tr>
<tr>
<td>class-names</td>
<td>The list of the fully qualified class names of providers to load. These are loaded after the service-loader discovered providers, and any duplicates will be skipped.</td>
</tr>
<tr>
<td>configuration</td>
<td>The key and value configuration to be passed to the provider to initialize it.</td>
</tr>
<tr>
<td>module</td>
<td>The name of the module to load the provider from.</td>
</tr>
<tr>
<td>path</td>
<td>The path of the file to use to initialize the providers.</td>
</tr>
<tr>
<td>relative-to</td>
<td>The base path of the configuration file.</td>
</tr>
</tbody>
</table>
### Table A.84. provider-sasl-server-factory Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>providers</td>
<td>The providers to use to locate the factories. If not specified, the globally registered list of providers will be used.</td>
</tr>
</tbody>
</table>

### Table A.85. regex-principal-transformer Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern</td>
<td>The regular expression to use to locate the portion of the name to be replaced.</td>
</tr>
<tr>
<td>replace-all</td>
<td>If true all occurrences of the pattern matched are replaced. If false only the first occurrence is replaced. Defaults to false.</td>
</tr>
<tr>
<td>replacement</td>
<td>The value to be used as the replacement.</td>
</tr>
</tbody>
</table>

### Table A.86. regex-role-mapper Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern</td>
<td>The regular expression to use to match roles. You can use group capturing if you want to use a portion of the original role in the replacement. For example, to capture a string after a hyphen in roles such as “app-admin”, “batch-admin”, use the pattern .*-([a-zA-Z])$.</td>
</tr>
<tr>
<td>replacement</td>
<td>The string to replace the match. You can use a fixed string or refer to captured groups from the regular expression specified in the pattern attribute. For the example, in the pattern above, you can use $1 to refer to the first captured group – which is the string &quot;admin&quot;, for the roles &quot;app-admin&quot; and &quot;batch-admin&quot;.</td>
</tr>
<tr>
<td>keep-non-mapped</td>
<td>Set the value to true to preserve the roles that do not match the regular expression specified in the pattern attribute.</td>
</tr>
</tbody>
</table>

### Table A.87. regex-validating-principal-transformer Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>match</td>
<td>If true the name must match the given pattern to make validation successful. If false the name must not match the given pattern to make validation successful. This defaults to true.</td>
</tr>
<tr>
<td>pattern</td>
<td>The regular expression to use for the principal transformer.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mechanism-configurations</td>
<td>The list of mechanism specific configurations.</td>
</tr>
<tr>
<td>sasl-server-factory</td>
<td>The SASL server factory to associate with this resource.</td>
</tr>
<tr>
<td>security-domain</td>
<td>The security domain to associate with this resource.</td>
</tr>
</tbody>
</table>

Table A.89. sasl-authentication-factory mechanism-configurations Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>credential-security-factory</td>
<td>The security factory to use to obtain a credential as required by the mechanism.</td>
</tr>
<tr>
<td>final-principal-transformer</td>
<td>A final principal transformer to apply for this mechanism realm.</td>
</tr>
<tr>
<td>host-name</td>
<td>The host name this configuration applies to.</td>
</tr>
<tr>
<td>mechanism-name</td>
<td>This configuration will only apply where a mechanism with the name specified is used. If this attribute is omitted then this will match any mechanism name.</td>
</tr>
<tr>
<td>mechanism-realm-configurations</td>
<td>The list of definitions of the realm names as understood by the mechanism.</td>
</tr>
<tr>
<td>protocol</td>
<td>The protocol this configuration applies to.</td>
</tr>
<tr>
<td>post-realm-principal-transformer</td>
<td>A principal transformer to apply after the realm is selected.</td>
</tr>
<tr>
<td>pre-realm-principal-transformer</td>
<td>A principal transformer to apply before the realm is selected.</td>
</tr>
<tr>
<td>realm-mapper</td>
<td>The realm mapper to be used by the mechanism.</td>
</tr>
</tbody>
</table>

Table A.90. sasl-authentication-factory mechanism-configurations mechanism-realm-configurations Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>final-principal-transformer</td>
<td>A final principal transformer to apply for this mechanism realm.</td>
</tr>
<tr>
<td>post-realm-principal-transformer</td>
<td>A principal transformer to apply after the realm is selected.</td>
</tr>
<tr>
<td>pre-realm-principal-transformer</td>
<td>A principal transformer to apply before the realm is selected.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>realm-mapper</td>
<td>The realm mapper to be used by the mechanism.</td>
</tr>
<tr>
<td>realm-name</td>
<td>The name of the realm to be presented by the mechanism.</td>
</tr>
</tbody>
</table>

### Table A.91. secret-key-credential-store Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>Set the value to <strong>false</strong> if you do not want ELytron to create one if it doesn’t already exist. Defaults to <strong>true</strong>.</td>
</tr>
<tr>
<td>default-alias</td>
<td>The alias name for a key generated by default. The default value is <strong>key</strong>.</td>
</tr>
<tr>
<td>key-size</td>
<td>The size of a generated key. The default size is 256 bits. You can set the value to one of the following:</td>
</tr>
<tr>
<td></td>
<td>- 128</td>
</tr>
<tr>
<td></td>
<td>- 192</td>
</tr>
<tr>
<td></td>
<td>- 256</td>
</tr>
<tr>
<td>path</td>
<td>The path to the credential store.</td>
</tr>
<tr>
<td>populate</td>
<td>If a credential store does not contain a <strong>default-alias</strong>, this attribute indicates whether Elytron should create one. The default is <strong>true</strong>.</td>
</tr>
<tr>
<td>relative-to</td>
<td>A reference to a previously defined path that the attribute <strong>path</strong> is relative to.</td>
</tr>
</tbody>
</table>

### Table A.92. server-ssl-context Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>authentication-optional</td>
<td>If <strong>true</strong> rejecting of the client certificate by the security domain will not prevent the connection. This allows a fall through to use other authentication mechanisms, such as form login, when the client certificate is rejected by security domain. This has an effect only when the security domain is set. This defaults to <strong>false</strong>.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cipher-suite-filter</td>
<td>The filter to apply to specify the enabled cipher suites. This filter takes a list of items delimited by colons, commas, or spaces. Each item may be an OpenSSL-style cipher suite name, a standard SSL/TLS cipher suite name, or a keyword such as TLSv1.2 or DES. A full list of keywords as well as additional details on creating a filter can be found in the Javadoc for the CipherSuiteSelector class. The default value is DEFAULT, which corresponds to all known cipher suites that do not have NULL encryption and excludes any cipher suites that have no authentication.</td>
</tr>
<tr>
<td>final-principal-transformer</td>
<td>A final principal transformer to apply for this mechanism realm.</td>
</tr>
<tr>
<td>key-manager</td>
<td>Reference to the key managers to use within the SSLContext.</td>
</tr>
<tr>
<td>maximum-session-cache-size</td>
<td>The maximum number of SSL/TLS sessions to be cached.</td>
</tr>
<tr>
<td>need-client-auth</td>
<td>If true a client certificate is required on SSL handshake. Connection without trusted client certificate will be rejected. This defaults to false.</td>
</tr>
<tr>
<td>post-realm-principal-transformer</td>
<td>A principal transformer to apply after the realm is selected.</td>
</tr>
<tr>
<td>pre-realm-principal-transformer</td>
<td>A principal transformer to apply before the realm is selected.</td>
</tr>
<tr>
<td>protocols</td>
<td>The enabled protocols. Allowed options are SSLv2, SSLv3, TLSv1, TLSv1.1, TLSv1.2, TLSv1.3. This defaults to enabling TLSv1, TLSv1.1, TLSv1.2, and TLSv1.3.</td>
</tr>
<tr>
<td>provider-name</td>
<td>The name of the provider to use. If not specified, all providers from providers will be passed to the SSLContext.</td>
</tr>
<tr>
<td>providers</td>
<td>The name of the providers to obtain the Provider[] to use to load the SSLContext.</td>
</tr>
<tr>
<td>realm-mapper</td>
<td>The realm mapper to be used for SSL authentication.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>security-domain</td>
<td>The security domain to use for authentication during SSL/TLS session establishment.</td>
</tr>
<tr>
<td>session-timeout</td>
<td>The timeout for SSL/TLS sessions.</td>
</tr>
<tr>
<td>trust-manager</td>
<td>Reference to the trust-manager to use within the SSLContext.</td>
</tr>
<tr>
<td>use-cipher-suites-order</td>
<td>If true the cipher suites order defined on the server will be used. If false the cipher suites order presented by the client will be used. Defaults to true.</td>
</tr>
<tr>
<td>want-client-auth</td>
<td>If true a client certificate will be requested, but not required, on SSL handshake. If a security domain is referenced and supports X509 evidence, this will be set to true automatically. This is ignored when need-client-auth is set. This defaults to false.</td>
</tr>
<tr>
<td>wrap</td>
<td>If true, the returned SSLEngine, SSLSocket, and SSLServerSocket instances will be wrapped to protect against further modification. This defaults to false.</td>
</tr>
</tbody>
</table>

**NOTE**

The realm mapper and principal transformer attributes for a server-ssl-context apply only for the SASL EXTERNAL mechanism, where the certificate is verified by the trust manager. HTTP CLIENT-CERT authentication settings are configured in an http-authentication-factory.

<table>
<thead>
<tr>
<th>Table A.93. service-loader-http-server-mechanism-factory Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
</tr>
<tr>
<td>module</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table A.94. service-loader-sasl-server-factory Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
</tr>
<tr>
<td>module</td>
</tr>
</tbody>
</table>

<p>| Table A.95. simple-permission-mapper Attributes |</p>
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapping-mode</td>
<td>The mapping mode that should be used in the event of multiple matches. Allowed values are, and, or, xor, unless, and first. The default is first.</td>
</tr>
<tr>
<td>permission-mappings</td>
<td>The list of defined permission mappings.</td>
</tr>
</tbody>
</table>

**Table A.96. simple-permission-mapper permission-mappings Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>permission-sets</td>
<td>The permission sets to assign in the event of a match. Permission sets can be used to assign permissions to an identity.</td>
</tr>
<tr>
<td>principals</td>
<td>The list of principals to compare when mapping permissions, if the identities principal matches any one in the list it is a match.</td>
</tr>
<tr>
<td>roles</td>
<td>The list of roles to compare when mapping permissions, if the identity is a member of any one in the list it is a match.</td>
</tr>
</tbody>
</table>

**IMPORTANT**

The permissions attribute is deprecated, and is replaced by permission-sets.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delegate-realm-mapper</td>
<td>The realm mapper to delegate to if there is no match using the pattern.</td>
</tr>
<tr>
<td>pattern</td>
<td>The regular expression which must contain at least one capture group to extract the realm from the name.</td>
</tr>
</tbody>
</table>

**Table A.98. simple-role-decoder Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>The name of the attribute from the identity to map directly to roles.</td>
</tr>
</tbody>
</table>
Table A.99. source-address-role-decoder Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern</td>
<td>A regular expression that specifies the IP address of a client or the IP addresses of clients to match.</td>
</tr>
<tr>
<td>source-address</td>
<td>Specifies the IP address of the client.</td>
</tr>
<tr>
<td>roles</td>
<td>Provides the list of roles to assign to a user if the IP address of the client matches the values specified in the pattern attribute or the source-address attribute.</td>
</tr>
</tbody>
</table>

**NOTE**

You must specify at least one IP address in either the source-address attribute or the pattern attribute. Otherwise, you cannot make authorization decisions based on the IP address of a client.

Table A.100. syslog-audit-log Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>The format that audit events should be recorded in.</td>
</tr>
<tr>
<td></td>
<td>Supported values:</td>
</tr>
<tr>
<td></td>
<td>• JSON</td>
</tr>
<tr>
<td></td>
<td>• SIMPLE</td>
</tr>
<tr>
<td></td>
<td>Default value:</td>
</tr>
<tr>
<td></td>
<td>• SIMPLE</td>
</tr>
<tr>
<td>host-name</td>
<td>The host name to be embedded into all events sent to the syslog server.</td>
</tr>
<tr>
<td>port</td>
<td>The listening port on the syslog server.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| reconnect-attempts   | The maximum number of times that Elytron will attempt to send successive messages to a syslog server before closing the connection. The value of this attribute is only valid when the transmission protocol used is UDP. Supported values:  
  - Any positive **integer** value.  
  - `-1` indicates infinite reconnect attempts.  
  Default value:  
  - 0                                                                                                                                                      |
| server-address       | IP address of the syslog server, or a name that can be resolved by Java’s `InetAddress.getByName()` method.                                    |
| ssl-context          | The SSL context to use when connecting to the syslog server. This attribute is only required if `transport` is set to **SSL_TCP**.               |
| syslog-format        | The RFC format to be used for describing the audit event. Supported values:  
  - **RFC3164**  
  - **RFC5424**  
  Default value:  
  - **RFC5424**                                                                                                                                 |
| transport            | The transport layer protocol to use to connect to the syslog server. Supported values:  
  - **SSL_TCP**  
  - **TCP**  
  - **UDP**  
  Default value:  
  - **TCP**                                                                                                                                 |

Table A.101. File audit logger attributes
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoflush</td>
<td>Specifies if the output stream requires flushing after every audit event. If you do not define the attribute, the value of the <code>synchronized</code> is the default value.</td>
</tr>
<tr>
<td>format</td>
<td>Default value is <code>SIMPLE</code>. Use <code>SIMPLE</code> for human readable text format, or <code>JSON</code> for storing individual events in <code>JSON</code>.</td>
</tr>
<tr>
<td>path</td>
<td>Defines the location of the log files</td>
</tr>
<tr>
<td>relative-to</td>
<td>Optional attribute. Defines the location of the log files</td>
</tr>
<tr>
<td>synchronized</td>
<td>Default value is <code>true</code>. Specifies that the file descriptor gets synchronized after every audit event.</td>
</tr>
</tbody>
</table>

Table A.102. Size rotating file audit logging attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoflush</td>
<td>Specifies if the output stream requires flushing after every audit event. If you do not define the attribute, the value of the <code>synchronized</code> is the default value.</td>
</tr>
<tr>
<td>format</td>
<td>Default value is <code>SIMPLE</code>. Use <code>SIMPLE</code> for human readable text format, or <code>JSON</code> for storing individual events in <code>JSON</code>.</td>
</tr>
<tr>
<td>max-backup-index</td>
<td>The maximum number of files to back up when rotating. The default value is <code>1</code>.</td>
</tr>
<tr>
<td>path</td>
<td>Defines the location of the log files</td>
</tr>
<tr>
<td>relative-to</td>
<td>Optional attribute. Defines the location of the log files.</td>
</tr>
<tr>
<td>rotate-on-boot</td>
<td>By default, Elytron does not create a new log file when you restart a server. Set this attribute to <code>true</code> to rotate the log on server restart.</td>
</tr>
<tr>
<td>rotate-size</td>
<td>The maximum size that the log file can reach before Elytron rotates the log. The default is <code>10m</code> for 10 megabytes. You can also define the maximum size of the log with k, g, b, or t units. You can specify units in either uppercase or lowercase characters.</td>
</tr>
<tr>
<td>suffix</td>
<td>Optional attribute. Adds a date suffix to a rotated log. You must use the <code>java.text.format.DateTimeFormatter</code> format. For example <code>yyyy-MM-dd-HH</code>.</td>
</tr>
</tbody>
</table>
synchronized

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronized</td>
<td>Default value is <strong>true</strong>. Specifies that the file descriptor gets synchronized after every audit event.</td>
</tr>
</tbody>
</table>

### Table A.103. token-realm Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jwt</td>
<td>A token validator to be used in conjunction with a token-based realm that handles security tokens based on the JWT/JWS standard.</td>
</tr>
<tr>
<td>oauth2-introspection</td>
<td>A token validator to be used in conjunction with a token-based realm that handles OAuth2 Access Tokens and validates them using an endpoint compliant with the RFC-7662 OAuth2 Token Introspection specification.</td>
</tr>
<tr>
<td>principal-claim</td>
<td>The name of the claim that should be used to obtain the principal’s name. The default is <strong>username</strong>.</td>
</tr>
</tbody>
</table>

### Table A.104. token-realm jwt Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>audience</td>
<td>A list of strings representing the audiences supported by this configuration. During validation JWT tokens must have an <strong>aud</strong> claim that contains one of the values defined here.</td>
</tr>
<tr>
<td>certificate</td>
<td>The name of the certificate with a public key to load from the keystore that is defined by the <strong>key-store</strong> attribute.</td>
</tr>
<tr>
<td>client-ssl-context</td>
<td>The SSL context to use for a remote JSON Web Key (JWK). This enables you to use the URL from the <strong>jku</strong> (JSON Key URL) header parameter to fetch public keys for token verification.</td>
</tr>
<tr>
<td>host-name-verification-policy</td>
<td>A policy that defines how host names should be verified when using remote JSON Web Keys. You can set either of the following values for the attribute:</td>
</tr>
</tbody>
</table>

  - **ANY**, which disables hostname verification.
  - **DEFAULT**, which denies connections where a certificate mismatch occurs between a server name from the certificate and the connecting host. |
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>issuer</td>
<td>A list of strings representing the issuers supported by this configuration. During validation JWT tokens must have an iss claim that contains one of the values defined here.</td>
</tr>
<tr>
<td>key-store</td>
<td>The keystore from which the certificate with a public key should be loaded. This attribute, along with the certificate attribute, can also be used as an alternative to the public-key.</td>
</tr>
<tr>
<td>public-key</td>
<td>A public key in PEM Format. During validation, if a public key is provided, the signature will be verified based on the key value provided by this attribute. Alternatively, you can define a key-store and a certificate to configure the public key. This alternative key is used to verify tokens without the kid (Key ID) claim.</td>
</tr>
</tbody>
</table>

Table A.105. token-realm oauth2-introspection Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>client-id</td>
<td>The identifier of the client on the OAuth2 Authorization Server.</td>
</tr>
<tr>
<td>client-secret</td>
<td>The secret of the client.</td>
</tr>
<tr>
<td>client-ssl-context</td>
<td>The SSL context to be used if the introspection endpoint is using HTTPS.</td>
</tr>
<tr>
<td>host-name-verification-policy</td>
<td>A policy that defines how host names should be verified when using HTTPS. You can set either of the following values for the attribute:</td>
</tr>
<tr>
<td></td>
<td>- ANY, which disables hostname verification.</td>
</tr>
<tr>
<td></td>
<td>- DEFAULT, which denies connections where a certificate mismatch occurs between a server name from the certificate and the connecting host.</td>
</tr>
<tr>
<td>introspection-url</td>
<td>The URL of token introspection endpoint.</td>
</tr>
</tbody>
</table>

Table A.106. trust-manager Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>algorithm</td>
<td>The name of the algorithm to use to create the underlying TrustManagerFactory. This is provided by the JDK. For example, a JDK that uses SunJSSE provides the PKIX and SunX509 algorithms. More details on SunJSSE can be found in the Java Secure Socket Extension (JSSE) Reference Guide.</td>
</tr>
<tr>
<td>alias-filter</td>
<td>A filter to apply to the aliases returned from the keystore. This can either be a comma-separated list of aliases to return or one of the following formats:</td>
</tr>
<tr>
<td></td>
<td>- ALL:-alias1:-alias2</td>
</tr>
<tr>
<td></td>
<td>- NONE:+alias1:+alias2</td>
</tr>
<tr>
<td>certificate-revocation-list</td>
<td>Enables the certificate revocation list that can be checked by a trust manager. The attributes of certificate-revocation-list are:</td>
</tr>
<tr>
<td></td>
<td>- maximum-cert-path - The maximum number of non-self-issued intermediate certificates that can exist in a certification path. The default value is 5. (Deprecated. Use maximum-cert-path in trust-manager).</td>
</tr>
<tr>
<td></td>
<td>- path - The path to the configuration file that is used to initialize the provider.</td>
</tr>
<tr>
<td></td>
<td>- relative-to - The base path of the certificate revocation list file.</td>
</tr>
<tr>
<td></td>
<td>See Using a Certificate Revocation List for more information.</td>
</tr>
<tr>
<td>key-store</td>
<td>Reference to the key-store to use to initialize the underlying TrustManagerFactory.</td>
</tr>
<tr>
<td>maximum-cert-path</td>
<td>The maximum number of non-self-issued intermediate certificates that can exist in a certification path. The default value is 5.</td>
</tr>
<tr>
<td></td>
<td>This attribute has been moved to trust-manager from certificate-revocation-list inside trust-manager in JBoss EAP 7.3. For backward compatibility, the attribute is also present in certificate-revocation-list. Going forward, use maximum-cert-path in trust-manager.</td>
</tr>
</tbody>
</table>

**NOTE**

Define maximum-cert-path in either trust-manager or in certificate-revocation-list not in both.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>only-leaf-cert</td>
<td>Check revocation status of only the leaf certificate. This is an optional attribute. The default value is <strong>false</strong>.</td>
</tr>
<tr>
<td>provider-name</td>
<td>The name of the provider to use to create the underlying <strong>TrustManagerFactory</strong>.</td>
</tr>
<tr>
<td>providers</td>
<td>Reference to obtain the <strong>Provider[]</strong> to use when creating the underlying <strong>TrustManagerFactory</strong>.</td>
</tr>
<tr>
<td>soft-fail</td>
<td>When set to <strong>true</strong>, certificates with an unknown revocation status are accepted. This is an optional attribute. The default value is <strong>false</strong>.</td>
</tr>
</tbody>
</table>

**Table A.107. x500-attribute-principal-decoder Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute-name</td>
<td>The name of the X.500 attribute to map. This can also be defined using the <strong>oid</strong> attribute.</td>
</tr>
<tr>
<td>convert</td>
<td>When set to <strong>true</strong>, the principal decoder will attempt to convert a principal to a <strong>X500Principal</strong>, if it is not already of that type. If the conversion fails, the original value is used as the principal.</td>
</tr>
<tr>
<td>joiner</td>
<td>The joining string. The default value is a period (<code>.</code>).</td>
</tr>
<tr>
<td>maximum-segments</td>
<td>The maximum number of occurrences of the attribute to map. The default value is <strong>2147483647</strong>.</td>
</tr>
<tr>
<td>oid</td>
<td>The OID of the X.500 attribute to map. This can also be defined using the <strong>attribute-name</strong> attribute.</td>
</tr>
<tr>
<td>required-attributes</td>
<td>The list of attribute names of the attributes that must be present in the principal</td>
</tr>
<tr>
<td>required-oids</td>
<td>The list of OIDs of the attributes that must be present in the principal.</td>
</tr>
<tr>
<td>reverse</td>
<td>If <strong>true</strong> the attribute values will be processed and returned in reverse order. The default value is <strong>false</strong>.</td>
</tr>
<tr>
<td>start-segment</td>
<td>The starting occurrence of the attribute you want to map. This uses a zero-based index and the default value is <strong>0</strong>.</td>
</tr>
</tbody>
</table>

**Table A.108. x509-subject-alternative-name-evidence-decoder Attributes**
### A.2. CONFIGURE YOUR ENVIRONMENT TO USE THE BOUNCYCASTLE PROVIDER

You can configure your JBoss EAP installation to use a BouncyCastle provider. The Bouncy Castle JARs are not provided by Red Hat, and must be obtained directly from Bouncy Castle.

**IMPORTANT**

Java 8 must be used when the BouncyCastle providers are specified, as the BouncyCastle APIs are only certified up to Java 8.

1. Include both BouncyCastle JARs, beginning with `bc-fips` and `bctls-fips`, on your JDK's classpath. For Java 8 this is accomplished by placing the JAR files in `$JAVA_HOME/lib/ext`.

2. Using either of the following methods, include the BouncyCastle providers in your Java security configuration file:

   - A default configuration file, `java.security`, is provided in your JDK, and can be updated to include the BouncyCastle providers. This file is used if no other security configuration files are specified. See the JDK vendor’s documentation for the location of this file.

   - Define a custom Java security configuration file and reference it by adding the `-Djava.security.properties=\path/to/java.security.properties` system property. When referenced using two equal signs the default policy is overwritten, and only the providers defined in the referenced file are used. When a single equal sign is used, as in `-Djava.security.properties=\path/to/java.security.properties`, then the providers are appended to the default security file, preferring to use the file passed in the argument when keys are specified in both files. This option is useful when having multiple JVMs running on the same host that require different security settings.
An example configuration file that defines these providers is seen below.

**Example: BouncyCastle Security Policy**

```java
# We can override the values in the JRE_HOME/lib/security/java.security
# file here. If both properties files specify values for the same key, the
# value from the command-line properties file is selected, as it is the last
# one loaded. We can reorder and change security providers in this file.
security.provider.1=org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider
security.provider.2=org.bouncycastle.jsse.provider.BouncyCastleJsseProvider fips:BCFIPS
security.provider.3=sun.security.provider.Sun
security.provider.4=com.sun.crypto.provider.SunJCE

# This is a comma-separated list of algorithm and/or algorithm:provider
# entries.
#
# securerandom.strongAlgorithms=DEFAULT:BCFIPS
```

**IMPORTANT**

If the default configuration file is updated, then every other `security.provider.X` line in this file, for example `security.provider.2`, must increase its value of `X` to ensure that this provider is given priority. Each provider must have a unique priority.

3. Configure the **elytron** subsystem to exclusively use the **BouncyCastle** providers. By default, the system is configured to use both the **elytron** and **openssl** providers. Because it also includes a TLS implementation, it is recommended to disable the OpenSSL provider to ensure the TLS implementation from Bouncy Castle is used.

```text
/subsystem=elytron:write-attribute(name=final-providers,value=elytron)
```

4. Reload the server for the changes to take effect.

```
reload
```

### A.3. SASL AUTHENTICATION MECHANISMS REFERENCE

#### A.3.1. Support Level for SASL Authentication Mechanisms

<table>
<thead>
<tr>
<th>Name</th>
<th>Support Level</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANONYMOUS</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>DIGEST-SHA-512</td>
<td>Technology Preview</td>
<td>Supported but name not currently IANA registered.</td>
</tr>
<tr>
<td>DIGEST-SHA-256</td>
<td>Technology Preview</td>
<td>Supported but name not currently IANA registered.</td>
</tr>
<tr>
<td>Name</td>
<td>Support Level</td>
<td>Comments</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>DIGEST-SHA</td>
<td>Technology Preview</td>
<td>Supported but name not currently IANA registered.</td>
</tr>
<tr>
<td>DIGEST-MD5</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>GS2-KRB5</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>GS2-KRB5-PLUS</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>GSSAPI</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>JBOSS-LOCAL-USER</td>
<td>Supported</td>
<td>Supported but name not currently IANA registered.</td>
</tr>
<tr>
<td>OAUTHBEARER</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>OTP</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>PLAIN</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>SCRAM-SHA-1</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>SCRAM-SHA-1-PLUS</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>SCRAM-SHA-256</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>SCRAM-SHA-256-PLUS</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>SCRAM-SHA-384</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>SCRAM-SHA-384-PLUS</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>SCRAM-SHA-512</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>SCRAM-SHA-512-PLUS</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>9798-U-RSA-SHA1-ENC</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>9798-M-RSA-SHA1-ENC</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>9798-U-DSA-SHA1</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>9798-M-DSA-SHA1</td>
<td>Not supported</td>
<td></td>
</tr>
</tbody>
</table>
A.3.2. SASL Authentication Mechanism Properties

You can see a list of standard Java SASL authentication mechanism properties in the Java documentation. Other JBoss EAP-specific SASL authentication mechanism properties are listed in the following tables.

Table A.109. SASL Properties Used During SASL Mechanism Negotiation or Authentication Exchange

<table>
<thead>
<tr>
<th>Property</th>
<th>Client / Server</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.sun.security.sasl.digest.realm</td>
<td>Server</td>
<td>Used by some SASL mechanisms, including the DIGEST-MD5 algorithm supplied with most Oracle JDKs, to provide the list of possible server realms to the mechanism. Each realm name must be separated by a space character (U+0020).</td>
</tr>
<tr>
<td>com.sun.security.sasl.digest.utf8</td>
<td>Client, server</td>
<td>Used by some SASL mechanisms, including the DIGEST-MD5 algorithm supplied with most Oracle JDKs, to indicate that information exchange should take place using UTF-8 character encoding instead of the default Latin-1/ISO-8859-1 encoding. The default value is true.</td>
</tr>
<tr>
<td>wildfly.sasl.authentication-timeout</td>
<td>Server</td>
<td>The amount of time, in seconds, after which a server should terminate an authentication attempt. The default value is 150 seconds.</td>
</tr>
<tr>
<td>wildfly.sasl.channel-binding-required</td>
<td>Client, server</td>
<td>Indicates that a mechanism which supports channel binding is required. A value of true indicates that channel binding is required. Any other value, or lack of this property, indicates that channel binding is not required.</td>
</tr>
<tr>
<td>wildfly.sasl.digest.alternative_protocols</td>
<td>Server</td>
<td>Supplies a separated list of alternative protocols that are acceptable in responses received from the client. The list can be space, comma, tab, or new line separated.</td>
</tr>
<tr>
<td>Property</td>
<td>Client / Server</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>wildfly.sasl.gssapi.client.delegate-credential</td>
<td>Client</td>
<td>Specifies if the GSSAPI mechanism supports credential delegation. If set to true, the credential is delegated from the client to the server. This property defaults to true if a GSSCredential is provided using the javax.security.sasl.credentials property. Otherwise, the default value is false.</td>
</tr>
<tr>
<td>wildfly.sasl.gs2.client.delegate-credential</td>
<td>Client</td>
<td>Specifies if the GS2 mechanism supports credential delegation. If set to true, the credential is delegated from the client to the server. This property defaults to true if a GSSCredential is provided using a CredentialCallback. Otherwise, the default value is false.</td>
</tr>
<tr>
<td>wildfly.sasl.local-user.challenge-path</td>
<td>Server</td>
<td>Specifies the directory in which the server generates the challenge file. The default value is the java.io.tmpdir system property.</td>
</tr>
<tr>
<td>wildfly.sasl.local-user.default-user</td>
<td>Server</td>
<td>The user name to use for silent authentication.</td>
</tr>
<tr>
<td>wildfly.sasl.local-user.quiet-auth</td>
<td>Client</td>
<td>Enables silent authentication for a local user. The default value is true. Note that the Jakarta Enterprise Beans client and naming client disables silent local authentication if this property is not explicitly defined and a callback handler or user name was specified in the client configuration.</td>
</tr>
<tr>
<td>wildfly.sasl.local-user.use-secure-random</td>
<td>Server</td>
<td>Specifies whether the server uses a secure random number generator when creating the challenge. The default value is true.</td>
</tr>
<tr>
<td>wildfly.sasl.mechanism-query-all</td>
<td>Client, server</td>
<td>Indicates that all possible supported mechanism names should be returned, regardless of the presence or absence of any other properties. This property is only effective on calls to SaslServerFactory#getMechanismNames(Map) or SaslClientFactory#getMechanismNames(Map) for Elytron-provided SASL factories.</td>
</tr>
</tbody>
</table>
**wildfly.sasl.otp.alternate-dictionary**

Client

Provides an alternate dictionary to the OTP SASL mechanism. Each dictionary word must be separated by a space character (U+0020).

**wildfly.sasl.relax-compliance**

Server

The specifications for the SASL mechanisms mandate certain behavior and verification of that behavior at the opposite side of the connection. When interacting with other SASL mechanism implementations, some of these requirements are interpreted loosely. If this property is set to **true**, checking is relaxed where differences in specification interpretation has been identified. The default value is **false**.

**wildfly.sasl.scram.min-iteration-count**

Client, server

The minimum iteration count to use for SCRAM. The default value is **4096**.

**wildfly.sasl.scram.max-iteration-count**

Client, server

The maximum iteration count to use for SCRAM. The default value is **32786**.

**wildfly.sasl.secure-rng**

Client, server

The algorithm name of a SecureRandom implementation to use. Using this property can improve security, at the cost of performance.

**wildfly.security.sasl.digest.ciphers**

Client, server

Comma-separated list of supported ciphers that directly limits the set of supported ciphers for SASL mechanisms.

---

**Table A.110. SASL Properties Used After Authentication**

<table>
<thead>
<tr>
<th>Property</th>
<th>Client / Server</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wildfly.sasl.principal</td>
<td>Client</td>
<td>Contains the negotiated client principal after a successful SASL client-side authentication.</td>
</tr>
<tr>
<td>wildfly.sasl.security-identity</td>
<td>Server</td>
<td>Contains the negotiated security identity after a successful SASL server-side authentication.</td>
</tr>
</tbody>
</table>

---

**A.4. SECURITY AUTHORIZATION ARGUMENTS**

Arguments to the `security` commands in JBoss EAP are determined by the defined mechanism. Each mechanism requires different properties, and it is recommended to use tab completion to examine the various requirements for the defined mechanism.

**Table A.111. Universal Arguments**
### Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--mechanism</td>
<td>Specifies the mechanism to enable or disable. A list of supported SASL mechanisms is available at Support Level for SASL Authentication Mechanisms, and the BASIC, CLIENT_CERT, DIGEST, DIGEST-SHA-256, and FORM HTTP authentication mechanisms are currently supported.</td>
</tr>
<tr>
<td>--no-reload</td>
<td>If specified, then the server is not reloaded after the security command is completed.</td>
</tr>
</tbody>
</table>

**Mechanism Specific Attributes**

The following attributes are only eligible for specific mechanisms. They are grouped below based on their function.

### Table A.112. key-store Realm

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--key-store-name</td>
<td>The name of the truststore as an existing keystore. This must be specified if --key-store-realm-name is not used for the EXTERNAL SASL mechanism or the CLIENT_CERT HTTP mechanism.</td>
</tr>
<tr>
<td>--key-store-realm-name</td>
<td>The name of the truststore as an existing keystore realm. This must be specified if --key-store-name is not used for the EXTERNAL SASL mechanism or the CLIENT_CERT HTTP mechanism.</td>
</tr>
<tr>
<td>--roles</td>
<td>An optional argument that defines a comma separated list of roles associated with the current identity. If no existing role mapper contains the specified list of roles, then a role mapper will be generated and assigned.</td>
</tr>
</tbody>
</table>

### Table A.113. file-system Realm

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--exposed-realm</td>
<td>The realm exposed to the user.</td>
</tr>
<tr>
<td>--file-system-realm-name</td>
<td>The name of the filesystem realm.</td>
</tr>
<tr>
<td>--user-role-decoder</td>
<td>The name of the role decoder used to extract the roles from the user’s repository. This attribute is only used if --file-system-realm-name is specified.</td>
</tr>
</tbody>
</table>

### Table A.114. Properties Realm
Table A.115. Miscellaneous Properties

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--management-interface</td>
<td>The management interface to configure for management authentication commands. This defaults to the http-interface.</td>
</tr>
<tr>
<td>--new-auth-factory-name</td>
<td>Used to specify a name for the authentication factory. If not defined, a name is automatically created.</td>
</tr>
<tr>
<td>--new-realm-name</td>
<td>Used to specify a name for the properties file realm resource. If not defined, a name is automatically created.</td>
</tr>
<tr>
<td>--new-security-domain</td>
<td>Used to specify a name for the security domain. If not defined, a name is automatically created.</td>
</tr>
<tr>
<td>--super-user</td>
<td>Configures a local user with super-user permissions. Usable with the JBOSS-LOCAL-USER mechanism.</td>
</tr>
</tbody>
</table>

A.5. ELYTRON CLIENT SIDE ONE WAY EXAMPLE

After configuring a server SSL context, it is important to test the configuration if possible. An Elytron client SSL context can be placed in a configuration file and then executed from the management CLI, allowing functional testing of the server configuration. These steps assume that the server-side configuration is completed, and the server has been reloaded if necessary.

1. If the server keystore already exists, then proceed to the next step; otherwise, create the server keystore.

   $ keytool -genkeypair -alias localhost -keyalg RSA -keysize 1024 -validity 365 -keystore server.keystore.jks -dname "CN=localhost" -keypass secret -storepass secret
2. If the server certificate has already been exported, then proceed to the next step; otherwise, export the server certificate.

```bash
$ keytool -exportcert -keystore server.keystore.jks -alias localhost -keypass secret -storepass secret -file server.cer
```

3. Import the server certificate into the client’s truststore.

```bash
$ keytool -importcert -keystore client.truststore.jks -storepass secret -alias localhost -trustcacerts -file server.cer
```

4. Define the client-side SSL context inside of `example-security.xml`. This configuration file contains an Elytron `authentication-client` that defines the authentication and SSL configuration for outbound connections. The following file demonstrates defining a client SSL context and keystore.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <key-stores>
      <key-store name="clientStore" type="jks">
        <file name="/path/to/client.truststore.jks"/>
        <key-store-clear-password password="secret"/>
      </key-store>
    </key-stores>
    <ssl-contexts>
      <ssl-context name="client-SSL-context">
        <trust-store key-store-name="clientStore"/>
      </ssl-context>
    </ssl-contexts>
    <ssl-context-rules>
      <rule use-ssl-context="client-SSL-context"/>
    </ssl-context-rules>
  </authentication-client>
</configuration>
```

5. Using the management CLI, reference the newly created file and attempt to access the server. The following command accesses the management interface and executes the `whoami` command.

```bash
$ EAP_HOME/bin/jboss-cli.sh -c --controller=remote+https://127.0.0.1:9993 -Dwildfly.config.url=/path/to/example-security.xml :whoami
```

### A.6. ELYTRON CLIENT SIDE TWO WAY EXAMPLE

After configuring a server SSL context, it is important to test the configuration if possible. An Elytron client SSL context can be placed in a configuration file and then executed from the management CLI, allowing functional testing of the server configuration. These steps assume that the server-side configuration is completed, and the server has been reloaded if necessary.

1. If the server and client keystores already exist, then proceed to the next step; otherwise, create the server and client keystores.
2. If the server and client certificates have already been exported, then proceed to the next step; otherwise, export the server and client certificates.

```
$ keytool -genkeypair -alias localhost -keyalg RSA -keysize 1024 -validity 365 -keystore server.keystore.jks -dname "CN=localhost" -keypass secret -storepass secret
$ keytool -genkeypair -alias client -keyalg RSA -keysize 1024 -validity 365 -keystore client.keystore.jks -dname "CN=client" -keypass secret -storepass secret
```

3. Import the server certificate into the client’s truststore.

```
$ keytool -importcert -keystore client.truststore.jks -storepass secret -alias localhost -trustcacerts -file server.cer
```

4. Import the client certificate into the server’s truststore.

```
$ keytool -importcert -keystore server.truststore.jks -storepass secret -alias client -trustcacerts -file client.cer
```

5. Define the client-side SSL context inside of `example-security.xml`. This configuration file contains an Elytron `authentication-client` that defines the authentication and SSL configuration for outbound connections. The following file demonstrates defining a client SSL context and keystore.

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <authentication-client xmlns="urn:elytron:client:1.2">
    <key-stores>
      <key-store name="clientStore" type="jks">
        <file name="/path/to/client.truststore.jks"/>
        <key-store-clear-password password="secret"/>
      </key-store>
      <key-stores>
        <key-store name="clientKeyStore" type="jks">
          <file name="/path/to/client.keystore.jks"/>
          <key-store-clear-password password="secret"/>
        </key-store>
        <key-stores>
          <ssl-contexts>
            <ssl-context name="client-SSL-context">
              <trust-store key-store-name="clientStore"/>
              <key-store-ssl-certificate key-store-name="clientKeyStore" alias="client">
                <key-store-clear-password password="secret"/>
              </key-store-ssl-certificate>
            </ssl-context>
          </ssl-contexts>
          <ssl-context-rules>
            <rule use-ssl-context="client-SSL-context"/>
          </ssl-context-rules>
        </ssl-contexts>
      </key-stores>
    </key-stores>
  </authentication-client>
</configuration>
```
6. Using the management CLI, reference the newly created file and attempt to access the server. The following command accesses the management interface and executes the **whoami** command.

```
$ EAP_HOME/bin/jboss-cli.sh -c --controller=remote+https://127.0.0.1:9993 -Dwildfly.config.url=/path/to/example-security.xml :whoami
```

*Revised on 2022-06-27 13:37:47 UTC*