Red Hat JBoss Enterprise Application Platform 8.0

Secure storage of credentials in JBoss EAP

Guide to securely storing credentials in credential stores

Last Updated: 2024-02-21

Red Hat JBoss Enterprise Application Platform 8.0 Secure storage of credentials in JBoss EAP

Guide to securely storing credentials in credential stores

Legal Notice

Abstract

Guide to securely storing credentials in credential stores.

Table of Contents

PROVIDING FEEDBACK ON JBOSS EAP DOCUMENTATION

To report an error or to improve our documentation, log in to your Red Hat Jira account and submit an issue. If you do not have a Red Hat Jira account, then you will be prompted to create an account.

Procedure

1. Click the following link to **create a ticket**.
2. Enter a brief description of the issue in the **Summary**.
3. Provide a detailed description of the issue or enhancement in the **Description**. Include a URL to where the issue occurs in the documentation.
4. Clicking **Submit** creates and routes the issue to the appropriate documentation team.

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. CREDENTIALS AND CREDENTIAL STORES IN ELYTRON

1.1. TYPES OF 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.

1.1.1. KeyStoreCredentialStore/credential-store in Elytron

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

- Creating a **credential-store** for a standalone server
- Creating a **credential-store** for a managed domain
- Creating a **credential-store** using the WildFly Elytron tool
- Creating a **credential-store** using the Bouncy Castle provider
- **credential-store** attributes

1.1.2. PropertiesCredentialStore/secret-key-credential-store in Elytron

To start properly, JBoss EAP requires an initial key to unlock certain secure resources. Use the PropertiesCredentialStore to provide this initial 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
```

Additional resources

- Creating a **secret-key-credential-store** for a standalone server
- Creating a **secret-key-credential-store** using WildFly Elytron tool

- **secret-key-credential-store** attributes

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.

Additional resources

- Credential stores provided by Elytron
- Credential types supported by credential stores

1.3. CREDENTIAL TYPES SUPPORTED BY ELYTRON CREDENTIAL STORES

The following table illustrates which credential type is supported by which credential store:

Credential type	KeyStoreCredentialStore/credential-store	PropertiesCredentialStore/secret-key-credential-store
PasswordCredential	Yes	No
KeyPairCredential	Yes	No
SecretKeyCredential	Yes	Yes

Additional resources

- Credential types in Elytron
- Credential stores provided by Elytron

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

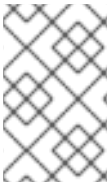
You can do these operation only on modifiable credential stores. All credential store types are modifiable by default.

1.4.1. Creating a credential-store for a standalone server

Create a **credential-store** 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

- KeyStoreCredentialStore/**credential-store** in Elytron
- Credential store operations using the JBoss EAP management CLI
- **credential-store** attributes

1.4.2. Creating a credential-store for a managed domain

You can create a **credential-store** 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 **credential-store** in each host controller and add credentials to each **credential-store**.
- Copy a populated **credential-store** from one host controller to all the other host controllers.
- Save your **credential-store** file in your Network File System (NFS), then use that file for all the **credential-store** resources you create.

Alternatively, you can create a **credential-store** file with credentials on a host controller without using the WildFly Elytron tool.

NOTE

You don't have to define a **credential-store** resource on every server, because every server on the same profile contains your **credential-store** file. You can find the **credential-store** file in the server **data** directory, **relative-to=jboss.server.data.dir**.

IMPORTANT

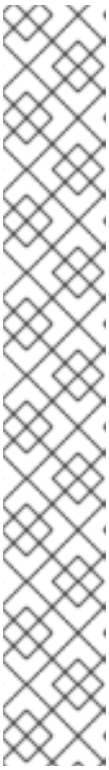
You cannot have the same name for a **credential-store** and a **secret-key-credential-store** because they implement the same Elytron capability: **org.wildfly.security.credential-store**.

The following procedure describes how to use the NFS to provide the **credential-store** file to all host controllers.

Procedure

1. Use the WildFly Elytron tool to create a **credential-store** storage file. For more information on this, see WildFly Elytron tool **credential-store** operations.

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 **credential-store** resources.



NOTE

To maintain consistency, for a **credential-store** file that multiple resources and host controllers use and which you stored in your NFS, you must use the **credential-store** in read-only mode. Additionally, make sure you provide an absolute path for your **credential-store** file.

Syntax

```
/profile=<profile_name>/subsystem=elytron/credential-store=<name_of_credential_store>:add(path=<absolute_path_to_store_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 **credential-store** 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** in Elytron
- Credential store operations using the WlidFly Elytron tool
- **credential-store** attributes

1.4.3. Creating a secret-key-credential-store for a standalone server

Create a **secret-key-credential-store** using the management CLI. When you create a **secret-key-credential-store**, JBoss EAP generates a secret key by default. The name of the generated key is **key** and its size is 256-bit.

Prerequisites

- JBoss EAP is running.
- You have provided at least read/write access to the directory containing the **secret-key-credential-store** for the user account under which JBoss EAP is running.

Procedure

- Use the following command to create a **secret-key-credential-store** 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"}
  ```

1.4.4. Adding a PasswordCredential to a credential-store

Add a plain text password for those resources that require one as a PasswordCredential to the **credential-store** 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 **credential-store**.
  For information about creating a **credential-store**, see Creating a **credential-store** for a standalone server.

Procedure

- Add a new PasswordCredential to a **credential-store**:

  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 **credential-store**:

  Syntax

  ```
  /subsystem=elytron/credential-store=<name_of_credential_store>:read-aliases()
  ```

  Example

  ```
  /subsystem=elytron/credential-store=exampleKeyStoreCredentialStore:read-aliases()
  {
      "outcome" => "success",
      "result" => ["passwordcredentialalias"]
  }
  ```

Additional resources

- KeyStoreCredentialStore/**credential-store** in Elytron
- **credential-store** attributes

1.4.5. Generating a SecretKeyCredential in a credential-store

Generate a SecretKeyCredential in a **credential-store**. 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 **credential-store**.
  For information about creating a **credential-store**, see Creating a **credential-store** for a standalone server.

Procedure

- Generate a SecretKeyCredential in a **credential-store** 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 **credential-store**:

Syntax

```
/subsystem=elytron/credential-store=<name_of_credential_store>:read-aliases()
```

Example

```
/subsystem=elytron/credential-store=exampleKeyStoreCredentialStore:read-aliases()
{
    "outcome" => "success",
    "result" => [
        "secretkeycredentialalias"
    ]
}
```

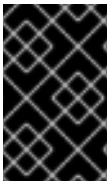
Additional resources

- KeyStoreCredentialStore/**credential-store** in Elytron
- **credential-store** attributes

1.4.6. Generating a SecretKeyCredential in a secret-key-credential-store

Generate a SecretKeyCredential in a **secret-key-credential-store**. 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 **secret-key-credential-store**.



IMPORTANT

Make sure that you create a backup of either **secret-key-credential-store**, SecretKeyCredential, or both, because JBoss EAP cannot decrypt or retrieve lost Elytron credentials.

You can use the **export** operation on the **secret-key-credential-store** to get the value of the SecretKeyCredential. You can then save this value as a backup.

Prerequisites

- You have created a **secret-key-credential-store**.
  For information about creating a **secret-key-credential-store**, see Creating a **secret-key-credential-store** for a standalone server.

Procedure

- Generate a SecretKeyCredential in a **secret-key-credential-store** 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** in Elytron
- **secret-key-credential-store** attributes

1.4.7. Importing a SecretKeyCredential to a secret-key-credential-store

You can import a SecretKeyCredential created outside of the **secret-key-credential-store** into an

Elytron **secret-key-credential-store**. Suppose you exported a SecretKeyCredential from another credential store – a **credential-store**, for example – you can import it to the **secret-key-credential-store**.

Prerequisites

- You have created a **secret-key-credential-store**.
  For information about creating a **secret-key-credential-store**, see Creating a **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 **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** in Elytron
- **secret-key-credential-store** attributes

1.4.8. Listing the credentials in a credential-store

To view all the credentials stored in the **credential-store**, you can list them using the management CLI.

Procedure

- List the credentials stored in a **credential-store** using the following management CLI command:

  Syntax

  ```
  /subsystem=elytron/credential-store=<name_of_credential_store>:read-aliases()
  ```

  Example

  ```
  {
      "outcome" => "success",
      "result" => [
          "passwordcredentialalias",
          "secretkeycredentialalias"
      ]
  }
  ```

Additional resources

- KeyStoreCredentialStore/**credential-store** in Elytron
- **credential-store** attributes

1.4.9. Exporting a SecretKeyCredential from a credential-store

You can export an existing SecretKeyCredential from a **credential-store** to use the SecretKeyCredential or to create a backup of the SecretKeyCredential.

Prerequisites

- You have generated a SecretKeyCredential the **credential-store**.
  For information about generating a SecretKeyCredential in a **credential-store**, see Generating a SecretKeyCredential in a **credential-store**.

Procedure

- Export a SecretKeyCredential from the **credential-store** 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+8JkoDCE6mFyA3cCIbSAZaXq5wgYejj1scYgdDqWiw=="}
}
```

Additional resources

- KeyStoreCredentialStore/**credential-store** in Elytron
- **credential-store** attributes

1.4.10. Exporting a SecretKeyCredential from a secret-key-credential-store

You can export an existing SecretKeyCredential from a **secret-key-credential-store** to use the SecretKeyCredential or to create a backup of the SecretKeyCredential.

Prerequisites

- You have either generated a SecretKeyCredential in the **secret-key-credential-store** or imported one to it.
  For information on generating a SecretKeyCredential in a **secret-key-credential-store**, Generating a SecretKeyCredential in a **secret-key-credential-store**.

  For information on importing a SecretKeyCredential to a **secret-key-credential-store**, see Importing a SecretKeyCredential to a **secret-key-credential-store**.

Procedure

- Export a SecretKeyCredential from the **secret-key-credential-store** using the following management CLI command:

  #### Syntax

  ```
  /subsystem=elytron/secret-key-credential-store=<name_of_credential_store>:export-secret-key(alias=<alias>)
  ```

  #### Example

  ```
  /subsystem=elytron/secret-key-credential-store=examplePropertiesCredentialStore:export-secret-key(alias=secretkeycredentialalias)
  {
      "outcome" => "success",
      "result" => {"key" => "RUxZAUtxXcYvz0aukZu+odOynIr0ByLhC72iwzlJsi+ZPmONgA=="}
  }
  ```

Additional resources

- PropertiesCredentialStore/**secret-key-credential-store** in Elytron
- **secret-key-credential-store** attributes

1.4.11. Removing a credential from credential-store

You can store every credential type in the **credential-store** 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 **credential-store** using the following management CLI command:

Syntax

```
/subsystem=elytron/credential-store=<name_of_credential_store>:remove-
alias(alias=<alias>, entry-type=<credential_type>)
```

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")
            ],
            "operation" => "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 'secretKeyCredentialAl
ias' 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** in Elytron
- **credential-store** attributes

1.4.12. Removing a credential from the secret-key-credential-store

You can store only the SecretKeyCredential type in a **secret-key-credential-store**. This means that, when you remove a credential from a **secret-key-credential-store**, you don't have to specify an **entry-type**.

Procedure

- Remove a SecretKeyCredential from the **secret-key-credential-store** 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** in Elytron
- **secret-key-credential-store** attributes

1.5. CREDENTIAL STORE OPERATIONS USING THE WILDFLY ELYTRON TOOL

You can perform various operations on credential stores offline using the WildFly Elytron tool.

1.5.1. Creating a credential-store using the WildFly Elytron tool

In Elytron, you can create a **credential-store** offline where you can save all the credential types.

Procedure

- Create a **credential-store** 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

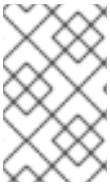
- KeyStoreCredentialStore/**credential-store** in Elytron
- Generating masked encrypted strings using the WildFly Elytron tool
- WildFly Elytron tool **credential-store** operations

1.5.2. Creating a credential-store using the Bouncy Castle provider

Create a **credential-store** using the Bouncy Castle provider.

Prerequisites

- Make sure that your environment is configured to use Bouncy Castle.



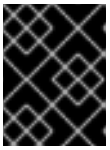
NOTE

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 **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>
```

3. Define the **credential-store** 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_credenti
al_store>"
```

Additional resources

- KeyStoreCredentialStore/**credential-store** in Elytron
- WildFly Elytron tool **credential-store** operations

1.5.3. Creating a secret-key-credential-store using WildFly Elytron tool

In Elytron, you can create a **secret-key-credential-store** 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** in Elytron
- WildFly Elytron tool **secret-key-credential-store** operations

1.5.4. WildFly Elytron tool credential-store operations

You can do various **credential-store** tasks using the WildFly Elytron tool, including the following:

Add a PasswordCredential

You can add a PasswordCredential to a **credential-store** 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_pa$$_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 **credential-store** 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

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --import-secret-key=imported --location=<path_to_credential_store> --password=<store_password>
```

Example

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --import-secret-key=imported --location=../cred_stores/example-credential-store.jceks --password=storePassword
```

Enter the secret key you want to import.

List all the credentials

You can list the credentials in the **credential-store** 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 **credential-store** 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

```
$ 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+i6DtcbkZHfybBJxPeS9mlVOmEYwjjmEA==
```

Remove a credential

You can remove a credential from a credential store using the following command:

Syntax

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "<path_to_store_file>" --password
<store_password> --remove <alias>
```

Example

```
$ 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** in Elytron
- Credential types in Elytron

1.5.5. WildFly Elytron tool **secret-key-credential-store** operations

You can do the following **secret-key-credential-store** operations for SecretKeyCredential using the WildFly Elytron tool:

Generate a SecretKeyCredential

You can generate a **SecteKeyCredential** in a **secret-key-credential-store** 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>" --type PropertiesCredentialStore
```

Example

```
$ 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

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --import-secret-key=imported --location=<path_to_credential_store> --type PropertiesCredentialStore
```

Example

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --import-secret-key=imported --location "standalone/configuration/properties-credential-store.cs" --type PropertiesCredentialStore
```

List all the credentials

You can list the credentials in the **secret-key-credential-store** using the following WildFly Elytron tool command:

Syntax

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "<path_to_store_file>"  --aliases --type
PropertiesCredentialStore
```

Example

```
$ 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 **secret-key-credential-store** using the following command:

Syntax

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --export-secret-key=<alias> --location
"<path_to_credential_store>"  --type PropertiesCredentialStore
```

Example

```
$ 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

```
$ EAP_HOME/bin/elytron-tool.sh credential-store --location "<path_to_store_file>" --remove
<alias> --type PropertiesCredentialStore
```

Example

```
$ 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** in Elytron
- Credential types in Elytron

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 **credential-store** using the WildFly Elytron tool.

Procedure

- Add the credential store to your running JBoss EAP server with the following management CLI command:

  Syntax

  ```
  /subsystem=elytron/credential-store=<store_name>:add(location="<path_to_store_file>",credential-reference={clear-text=<store_password>})
  ```

  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

- **credential-store** attributes

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**.

```
$ 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:

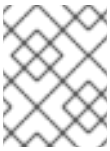
```
$ EAP_HOME/bin/elytron-tool.sh credential-store --import-key-pair example --private-key-location
/home/user/.ssh/id_rsa --location=<path_to_store_file>
```

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*:

```
$ 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
AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBMfncZuHmR7uglb0M96ieAr
RFtp42xPn9+ugukbY8dyjOXoi
cZrYRyy9+X68fylEWBMzyg+nhjWkxJlJ2M2LAGY=
```



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.

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 version="1.0" encoding="UTF-8"?>
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.6">

    <credential-stores>
      <credential-store name="${credential_store_name}">
        <protection-parameter-credentials>
          <clear-password password="${credential_store_password}"/>
        </protection-parameter-credentials>
        <attributes>
          <attribute name="path" value="${path_to_credential_store}"/>
        </attributes>
      </credential-store>
    </credential-stores>

    <authentication-rules>
      <rule use-configuration="${configuration_file_name}"/>
    </authentication-rules>
```

```
    <authentication-configurations>
      <configuration name="${configuration_file_name}">
        <credentials>
          <credential-store-reference store="${credential_store_name}" alias="${alias_of_key_pair}"/>
        </credentials>
      </configuration>
    </authentication-configurations>

  </authentication-client>
</configuration>
```

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

1.5.9. Generating masked encrypted strings using the WildFly Elytron tool

You can use the WildFly Elytron tool to generate 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-8VzWsSNwBaR676g8ujiIDdFKwSjOBHCHgnKf17nun3v;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.

1.6. 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.

1.7. 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

```
<configuration>
  <authentication-client xmlns="urn:elytron:client:1.7">
    ...
    <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>
```

1. A name for the credential store for use within the Elytron Client configuration file.
2. The password for the credential store.
3. The path to the credential store file.
4. The credential reference for a sensitive string stored in the credential store.

Additional resources

- Credential store operations using the WildFly Elytron tool .

1.8. CREATING FIPS 140-2 COMPLIANT CREDENTIAL STORES

You can configure Federal Information Processing Standard (FIPS) 140-2 compliant credential store in Elytron. FIPS 140-2, is a computer security standard, developed by a U.S. Government industry working group to validate the quality of cryptographic modules. FIPS publications (including 140-2) can be found

at the URL: http://csrc.nist.gov/publications/PubsFIPS.html.

You can configure FIPS 140-2 compliant credential store in Elytron using two different providers:

- SunPKCS#11 provider and Network Security Services (NSS) database.
  For more information, see Creating FIPS 140-2 compliant credential store using a SunPKCS#11 provider and NSS database.
- BouncyCastle providers.
  For more information, see Creating FIPS 140-2 compliant credential store using BouncyCastle providers.

IMPORTANT

JBoss EAP itself is not FIPS-certified. The level of FIPS support in JBoss EAP is that JBoss EAP can be used with FIPS-certified cryptographic implementations. The implementations tested are BouncyCastle and SunPKCS#11.

1.8.1. Creating FIPS 140-2 compliant credential store using a SunPKCS#11 provider and NSS database

NSS is a set of libraries that support cross-platform security-enabled client and server applications. You can use SunPKCS#11 provider with NSS library in JBoss EAP to implement FIPS 140-2 compliant cryptography. For information about NSS, see Mozilla docs - Network Security Services (NSS) . For information about SunPKCS#11 provider, see PKCS#11 Reference Guide.

1.8.1.1. JDKs that support FIPS when using a SunPKCS#11 provider and NSS database

Not all Java Development Kit (JDK) vendors support configuring the SunPKCS#11 security provider with a Network Security Services (NSS) software token, implemented by the NSS library, required for Federal Information Processing Standard (FIPS) 140-2 compliance. Ensure that your JDK supports it before configuring FIPS with the SunPKCS#11 provider and NSS database in JBoss EAP.

The following is a list of supported JDKs for JBoss EAP that support configuring SunPKCS#11 security:

- OpenJDK 11
- OpenJDK 17

Additional resources

- Configuring OpenJDK 11 on RHEL with FIPS
- Configuring OpenJDK 17 on RHEL with FIPS

1.8.1.2. Creating FIPS 140-2 compliant credential store using a SUNPKCS#11 provider and NSS database in FIPS enabled RHEL

Starting with Red Hat Enterprise Linux 8.4, if you enable the Federal Information Processing Standard (FIPS) system-wide crypto policy, FIPS for Java is also enabled automatically. You can use the default Network Security Services (NSS) database to create a FIPS 140-2 compliant credential store.

In this procedure **$JAVA_HOME** refers to the JDK installation path. Run the commands in this procedure as the root user.

Prerequisites

- FIPS is enabled in RHEL.
  You can check whether FIPS is enabled using the following command:

```
# fips-mode-setup --check
```

  For information about enabling FIPS in RHEL, see the following resources:

  - Installing the system in FIPS mode in the Red Hat Enterprise Linux documentation.
  - Switching the system to FIPS mode in the Red Hat Enterprise Linux documentation.

- NSS tools are installed.
  In Red Hat Enterprise Linux you can install NSS tools using the DNF package manager as follows:

```
# dnf install -y nss-tools
```

- Your Java Development Kit (JDK) supports configuration of PKCS#11 with an NSS library.
  For information about JDKs that support FIPS, see JDKs that support FIPS.
- JBoss EAP is running.

Procedure

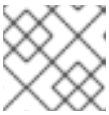
1. Update the value of **nssDbMode** in the **$JAVA_HOME/conf/security/nss.fips.cfg** file to **readWrite**.

   Example nss.fips.cfg contents

```
name = NSS-FIPS
nssLibraryDirectory = /usr/lib64
nssSecmodDirectory = sql:/etc/pki/nssdb
nssDbMode = readWrite
nssModule = fips

attributes(*,CKO_SECRET_KEY,CKK_GENERIC_SECRET)={ CKA_SIGN=true }
```

2. Generate an AES secret key to encrypt the credential stores.



   NOTE

   You must use store password **NONE** in the command.

   Syntax

```
# keytool -genseckey -keystore NONE -storetype PKCS11 -storepass NONE -alias
<key_alias> -keyalg <symmetric_key_algorithm> -keysize <key_size>
```

   Example

```
# keytool -genseckey -keystore NONE -storetype PKCS11 -storepass NONE -alias
exampleKeyAlias -keyalg AES -keysize 256
```

3. Verify that you can read the secret key.

```
# keytool -list -storetype pkcs11 -storepass NONE

Keystore type: PKCS11
Keystore provider: SunPKCS11-NSS-FIPS

Your keystore contains 1 entry

exampleKeyAlias, SecretKeyEntry,
```

4. Update the value of **nssDbMode** in the **$JAVA_HOME/conf/security/nss.fips.cfg** file to **readOnly**.

Example nss.fips.cfg contents

```
name = NSS-FIPS
nssLibraryDirectory = /usr/lib64
nssSecmodDirectory = sql:/etc/pki/nssdb
nssDbMode = readOnly
nssModule = fips

attributes(*,CKO_SECRET_KEY,CKK_GENERIC_SECRET)={ CKA_SIGN=true }
```

5. Add the SunJCE provider to the list of providers in the management CLI.

   a. Add a provider loader for SunJCE.

```
/subsystem=elytron/provider-loader=SunJCE:add(class-names=
[com.sun.crypto.provider.SunJCE])
{"outcome" => "success"}
```

   b. Configure Elytron and SunJCE in an aggregate provider.

Syntax

```
/subsystem=elytron/aggregate-providers=<aggregate_provider_name>:add(providers=
[elytron,SunJCE])
```

Example

```
/subsystem=elytron/aggregate-providers=exampleAggregateProvider:add(providers=
[elytron,SunJCE])
{"outcome" => "success"}
```

NOTE

The providers are called in the order that they are defined in the command.

6. Use the SunJCE provider to create a credential store in Elytron.

Syntax

```
/subsystem=elytron/credential-store=<credential_store_name>:add(implementation-properties={keyStoreType => PKCS11, external => true, keyAlias => <key_alias>, externalPath => <path_where_credential_store_is_to_be_saved>}, modifiable=true, credential-reference={clear-text=<password>}, create=true, other-providers=<aggregate_provider_name>)
```

Example

```
/subsystem=elytron/credential-store=exampleFipsCredentialStore:add(implementation-properties={keyStoreType => PKCS11, external => true, keyAlias => exampleKeyAlias, externalPath => /home/ashwin/example.store}, modifiable=true, credential-reference={clear-text=secret}, create=true, other-providers=exampleAggregateProvider)
{"outcome" => "success"}
```

Verification

1. Add an alias to the credential store.

Syntax

```
/subsystem=elytron/credential-store=<credential_store_name>:add-alias(alias=<alias>, secret-value=<secret_value>)
```

Example

```
/subsystem=elytron/credential-store=exampleFipsCredentialStore:add-alias(alias=exampleAlias, secret-value=secret)
{"outcome" => "success"}
```

2. List the aliases in the credential store.

Syntax

```
/subsystem=elytron/credential-store=<credential_store_name>:read-aliases()
```

Example

```
/subsystem=elytron/credential-store=exampleFipsCredentialStore:read-aliases()
{
    "outcome" => "success",
    "result" => ["examplealias"]
}
```

The created credential store is FIPS 140-2 compliant.

Additional resources

- **aggregate-providers** attributes

- **credential-store** attributes
- **credential-store** implementation properties

1.8.2. Creating FIPS 140-2 compliant credential store using BouncyCastle providers

BouncyCastle provides lightweight cryptography APIs for Java and C#. You can use the following BouncyCastle providers with JBoss EAP for creating FIPS 140-2 compliant credential stores:

- BouncyCastle FIPS provider for the Java Cryptography Extension (JCE) and the Java Cryptography Architecture (JCA).
- BouncyCastle FIPS provider for the Java Secure Socket Extension (JSSE).

For information about BouncyCastle, see The Legion of the Bouncy Castle.

1.8.2.1. Creating FIPS 140-2 compliant credential store using BouncyCastle providers

Starting with Red Hat Enterprise Linux 8.4, if you enable the Federal Information Processing Standard (FIPS) system-wide crypto policy, OpenJDK automatically enables different security providers. One of the security providers is the SunPKCS11 provider configured in FIPS mode. You can instead use the BouncyCastle providers to create a Federal Information Processing Standard (FIPS) 140-2 compliant credential store by following the below steps.

Prerequisites

- FIPS is enabled in RHEL.
  You can check whether FIPS is enabled using the following command:

  ```
  # fips-mode-setup --check
  ```

  For information about enabling FIPS in RHEL, see the following resources:

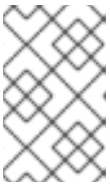
  - Installing the system in FIPS mode in the Red Hat Enterprise Linux documentation.
  - Switching the system to FIPS mode in the Red Hat Enterprise Linux documentation.
- Your Java Development Kit (JDK) supports the configuration of FIPS using BouncyCastle providers.
  For more information, see Java Related Questions on the Legion of the Bouncy Castle - FIPS Resources Page.

Procedure

1. Download the BouncyCastle jars from the following links:
   - bc-fips-*N* jar: Bouncy Castle Provider (FIPS Distribution) Maven .
   - bctls-fips-*N* jar: Bouncy Castle TLS/JSSE APIs (FIPS Distribution) .
     Where *N* stands for the BouncyCastle FIPS provider version.

     You can find information about the latest certified version of BouncyCastle FIPS provider that conforms to your environment at The Legion of the Bouncy Castle.
2. Create a configuration file called **java.security** with the following contents:
   -

```
fips.provider.1=org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider
fips.provider.2=org.bouncycastle.jsse.provider.BouncyCastleJsseProvider fips:BCFIPS
fips.provider.3=SUN
fips.provider.4=SunEC
fips.provider.5=com.sun.net.ssl.internal.ssl.Provider
```



NOTE

Do not modify the FIPS providers in the default **java.security** file. It is recommended to use your own **java.security** properties file as described in this procedure.

3. Generate an AES secret key to encrypt the credential stores.

Syntax

```
$ keytool -J-Djava.security.properties=<java_security_file> -genseckey -keystore
"<keystore_name>" -storetype BCFKS -storepass <store_password> -alias <key_alias> -
keyalg <symmetric_key_algorithm> -keysize <key_size> -keypass <key_password>  -
provider org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider -providerpath
<path_to_bc-fips_jar>  -dname "<certificate_contents>" -validity <validity_in_days>
```

Example

```
$ keytool -J-Djava.security.properties=<path_to_java_security_file>/java.security -genseckey
-keystore "examplekeystore.bcfks" -storetype BCFKS -storepass password -alias
exampleKeyAlias -keyalg AES -keysize 256 -keypass password -provider
org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider -providerpath <path_to_bc-
fips_jar>/bc-fips-1.0.2.jar -dname "CN=localhost" -validity 365
```

4. Verify that you can read the secret key.

Syntax

```
$ keytool -J-Djava.security.properties=<java_security_file> -list -keystore <keystore_name> -
storetype BCFKS -storepass <store_password> -provider
org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider -providerpath <path_to_bc-
fips_jar>
```

Example

```
$ keytool -J-Djava.security.properties=<path_to_java_security_file>/java.security -list -
keystore examplekeystore.bcfks -storetype BCFKS -storepass password -provider
org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider -providerpath <path_to_bc-
fips_jar>/bc-fips-1.0.2.jar
```

Example output

```
Keystore type: BCFKS
Keystore provider: BCFIPS
```

```
Your keystore contains 1 entry

exampleKeyAlias, Mar 1, 2023, SecretKeyEntry,
```

5. Start the server.
6. Configure JBoss EAP to use BouncyCastle providers using the management CLI.
   a. Add SunJCE provider to the list of providers.

```
/subsystem=elytron/provider-loader=SunJCE:add(class-names=
[com.sun.crypto.provider.SunJCE])
```

   b. Add BouncyCastle provider jar as a module in JBoss EAP.

Syntax

```
module add --name=org.bouncycastle.fips --resources=<path_to_bc-
fips_jar>:<path_to_bctls-fips_jar>
```

Example

```
module add --name=org.bouncycastle.fips --resources=<path_to_bc-fips_jar>/bc-fips-
1.0.2.jar:<path_to_bctls-fips_jar>/bctls-fips-1.0.2.jar
```

   c. Add the provider loader for the BouncyCastle providers.

Syntax

```
/subsystem=elytron/provider-
loader=<provider_loader_name>:add(module=org.bouncycastle.fips)
```

Example

```
/subsystem=elytron/provider-
loader=exampleProviderLoader:add(module=org.bouncycastle.fips)
```

   d. Configure BouncyCastle, SunJCE and combined-providers in an aggregate provider.

Syntax

```
/subsystem=elytron/aggregate-providers=<aggregate_provider_name>:add(providers=
[<provider_loader_name>,SunJCE,combined-providers])
```

Example

```
/subsystem=elytron/aggregate-providers=exampleAggregateProvider:add(providers=
[exampleProviderLoader,SunJCE,combined-providers])
```

NOTE

The providers are called in the order that they are defined in the command.

e. Reload the server.

```
reload
```

7. Use the BouncyCastle providers to create a credential store in Elytron.

Syntax

```
/subsystem=elytron/credential-store=<credential_store_name>:add(credential-reference=
{clear-text=<key_and_keystore_password>},implementation-properties=
{keyAlias=<key_alias>,external=true,externalPath=<path_to_BCFKS_credential_store>,keyS
toreType=BCFKS},create=true,path=<path_to_keystore>,modifiable=true, other-
providers=<aggregate_provider_name>)
```

Example

```
/subsystem=elytron/credential-store=exampleFipsCredentialStore:add(credential-reference=
{clear-text=password},implementation-properties={keyAlias=exampleKeyAlias,external=true,
externalPath=credentialStore.bcfks, keyStoreType=BCFKS}, create=true,
path=__<path_to_keystore>__/examplekeystore.bcfks, modifiable=true, other-
providers=exampleAggregateProvider)
```

Verification

1. Add an alias to the credential store.

Syntax

```
/subsystem=elytron/credential-store=<credential_store_name>:add-alias(alias=<alias>,
secret-value=<secret_value>)
```

Example

```
/subsystem=elytron/credential-store=exampleFipsCredentialStore:add-
alias(alias=exampleAlias, secret-value=secret)
```

2. List the aliases in the credential store.

Syntax

```
/subsystem=elytron/credential-store=<credential_store_name>:read-aliases()
```

Example

```
/subsystem=elytron/credential-store=exampleFipsCredentialStore:read-aliases()
{
    "outcome" => "success",
    "result" => ["examplealias"]
}
```

The created credential store is FIPS 140-2 compliant.

Additional resources

- **aggregate-providers** attributes
- **credential-store** attributes
- **credential-store** implementation properties
- **provider-loader** attributes
- To learn more about the **module add** command, you can run the **--help** command in the JBoss EAP management CLI:

```
module add --help
```

CHAPTER 2. PROVIDING AN INITIAL KEY TO JBOSS EAP TO UNLOCK SECURED RESOURCES

2.1. 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+L5zx9LmCRLyh5fjdfl1WM7lhfthKjeoEU+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+L5zx9LmCRLyh5fjdfl1WM7lhfthKjeoEU+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"=>{
      ...
      "expression-allowed"=>true,
      ...
    }]
}
```

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

2.2. 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 created a PropertiesCredentialStore and a secret key in it.
  For more information, see Creating a PropertiesCredentialStore/**secret-key-credential-store** for a standalone server.

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>})
   ```

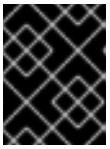
   #### Example

```
/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.

```
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:RUxZAUMQgtpG7oFlHR2j1Gkn3GKlHff+HR8GcMX1QXHvx2uGur
l=}"}
}
```

**${ENC::exampleResolver:RUxZAUMQgtpG7oFlHR2j1Gkn3GKlHff+HR8GcMX1QXHvx2uGu
rl=}** 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

2.3. 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**:

  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:RUxZAUMQgtpG7oFIHR2j1Gkn3GKIHff+HR8GcMX1QXHvx2uGurI=}})
{"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.

CHAPTER 3. REFERENCE

3.1. AGGREGATE-PROVIDERS ATTRIBUTES

You can configure **aggregate-providers** by setting the **providers** attributes.

Table 3.1. aggregate-providers Attributes

Attribute	Description
providers	The list of providers to aggregate. Elytron uses the first suitable provider found on the list.

3.2. CREDENTIAL-STORE ATTRIBUTES

You can configure **credential-store** by setting its attributes.

Table 3.2. credential-store Attributes

Attribute	Description
create	Specifies whether the credential store should create storage when it does not exist. The default values is **false**.
credential-reference	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**.
implementation-properties	Map of credentials store implementation-specific properties.
modifiable	Whether you can modify the credential store. The default value is **true**.
other-providers	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.
path	The file name of the credential store.
provider-name	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.
providers	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.

Attribute	Description
relative-to	The base path this credential store path is relative to.
type	Type of the credential store, for example, **KeyStoreCredentialStore**.

3.3. CREDENTIAL-STORE IMPLEMENTATION PROPERTIES

You can configure the **credential-store** implementation by setting its attributes.

Table 3.3. credential-store implementation properties

Attribute	Description
cryptoAlg	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**.
external	Whether data is stored to external storage and encrypted by the **keyAlias**. Defaults to **false**.
externalPath	Specifies path to external storage. This attribute is only valid if **external** is enabled.
keyAlias	The secret key alias within the credential store that is used to encrypt or decrypt data to the external storage.
keyStoreType	The keystore type, such as **PKCS11**. Defaults to **KeyStore.getDefaultType()**.

3.4. EXPRESSION=ENCRYPTION ATTRIBUTES

You can configure **expression=encryption** by setting its attributes.

Table 3.4. expression=encryption Attributes

Attribute	Description
default-resolver	Optional attribute. The resolver to use when an encrypted expression is defined without one. For example if you set "exampleResolver" as the **default-resolver** and you create an encrypted expression with the command **/subsystem=elytron/expression=encryption:create-expression(clear-text=TestPassword)**, Elytron uses "exampleResolver" as the resolver for this encrypted expression.

Attribute	Description
prefix	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.
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.

3.5. PROVIDER-LOADER ATTRIBUTES

You can configure **provider-loader** by setting its attributes.

Table 3.5. provider-loader attributes

Attribute	Description
argument	An argument to be passed into the constructor as the **Provider** is instantiated.
class-names	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.
configuration	The key and value configuration to be passed to the provider to initialize it.
module	The name of the module to load the provider from.
path	The path of the file to use to initialize the providers.
relative-to	The base path of the configuration file.

3.6. SECRET-KEY-CREDENTIAL-STORE ATTRIBUTES

You can configure **secret-key-credential-store** by setting its attributes.

Table 3.6. secret-key-credential-store Attributes

Attribute	Description
create	Set the value to **false** if you do not want Elytron to create one if it doesn't already exist. Defaults to **true**.
default-alias	The alias name for a key generated by default. The default value is **key**.
key-size	The size of a generated key. The default size is 256 bits. You can set the value to one of the following: • **128** • **192** • **256**
path	The path to the credential store.
populate	If a credential store does not contain a **default-alias**, this attribute indicates whether Elytron should create one. The default is **true**.
relative-to	A reference to a previously defined path that the attribute **path** is relative to.