AAA

Release master

Sep 18, 2019

Contents

Authentication, Authorization and Accounting (AAA) Documentation.

CHAPTER 1

Authentication, Authorization and Accounting (AAA) Services

1.1 Overview

Authentication, Authorization and Accounting (AAA) is a term for a framework controlling access to resources, enforcing policies to use those resources and auditing their usage. These processes are the fundamental building blocks for effective network management and security.

Authentication provides a way of identifying a user, typically by having the user enter a valid user name and valid password before access is granted. The process of authentication is based on each user having a unique set of criteria for gaining access. The AAA framework compares a user's authentication credentials with other user credentials stored in a database. If the credentials match, the user is granted access to the network. If the credentials don't match, authentication fails and access is denied.

Authorization is the process of finding out what an authenticated user is allowed to do within the system, which tasks can do, which API can call, etc. The authorization process determines whether the user has the authority to perform such actions.

Accounting is the process of logging the activity of an authenticated user, for example, the amount of data a user has sent and/or received during a session, which APIs called, etc.

1.1.1 Terms And Definitions

AAA Authentication, Authorization and Accounting.

Token A claim of access to a group of resources on the controller.

Domain A group of resources, direct or indirect, physical, logical, or virtual, for the purpose of access control.

User A person who either owns or has access to a resource or group of resources on the controller.

Role Opaque representation of a set of permissions, which is merely a unique string as admin or guest.

Credential Proof of identity such as user name and password, OTP, biometrics, or others.

Client A service or application that requires access to the controller.

Claim A data set of validated assertions regarding a user, e.g. the role, domain, name, etc.

IdP Identity Provider.

1.2 Quick Start

1.2.1 Building

Get the code:

```
git clone https://git.opendaylight.org/gerrit/aaa
```

Build it:

```
cd aaa && mvn clean install
```

1.2.2 Installing

AAA is automatically installed upon installation of odl-restconf, but you can install it yourself directly from the Karaf console through the following command:

```
feature:install odl-aaa-shiro
```

1.2.3 Pushing changes

The following are basic instructions to push your contributions to the project's GIT repository:

```
git add .
git commit -s
# make changes, add change id, etc.
git commit --amend
git push ssh://{username}@git.opendaylight.org:29418/aaa.git HEAD:refs/for/master
```

1.3 AAA Framework implementations

Since Boron release, the OpenDaylight's AAA services are based on the Apache Shiro Java Security Framework. The main configuration file for AAA is located at "etc/shiro.ini" relative to the OpenDaylight Karaf home directory.

1.3.1 Known limitations

The database (H2) used by ODL AAA Authentication store is not-cluster enabled. When deployed in a clustered environment each node needs to have its AAA user file synchronized using out of band means.

1.4 How to enable AAA

AAA is enabled through installing the odl-aaa-shiro feature. The vast majority of OpenDaylight's northbound APIs (and all RESTCONF APIs) are protected by AAA by default when installing the +odl-restconf+ feature, since the odl-aaa-shiro is automatically installed as part of them.

1.5 How to disable AAA

Edit the "etc/shiro.ini" file and replace the following:

```
/** = authcBasic
```

with

```
/** = anon
```

Then, restart the Karaf process.

1.6 How application developers can leverage AAA to provide servlet security

Previously the servlet's web.xml was edited to add the AAAShiroFilter. This has been replaced with programmatic initialization.

The Neutron project uses this new style the Neutron blueprint.xml and Neutron WebInitializer.java are helpful references.

1.7 AAA Realms

AAA plugin utilizes the Shiro Realms to support pluggable authentication & authorization schemes. There are two parent types of realms:

- AuthenticatingRealm
  - Provides no Authorization capability.
  - Users authenticated through this type of realm are treated equally.
- AuthorizingRealm
  - AuthorizingRealm is a more sophisticated AuthenticatingRealm, which provides the additional mechanisms to distinguish users based on roles.
  - Useful for applications in which roles determine allowed capabilities.

OpenDaylight contains five implementations:

- TokenAuthRealm
  - An AuthorizingRealm built to bridge the Shiro-based AAA service with the h2-based AAA implementation.
  - Exposes a RESTful web service to manipulate IdM policy on a per-node basis. If identical AAA policy is desired across a cluster, the backing data store must be synchronized using an out of band method.

  - A python script located at "etc/idmtool" is included to help manipulate data contained in the TokenAuthRealm.
  - Enabled out of the box. This is the realm configured by default.
- ODLJndiLdapRealm
  - An AuthorizingRealm built to extract identity information from IdM data contained on an LDAP server.
  - Extracts group information from LDAP, which is translated into OpenDaylight roles.
  - Useful when federating against an existing LDAP server, in which only certain types of users should have certain access privileges.
  - Disabled out of the box.
- ODLJndiLdapRealmAuthNOnly
  - The same as ODLJndiLdapRealm, except without role extraction. Thus, all LDAP users have equal authentication and authorization rights.
  - Disabled out of the box.
- ODLActiveDirectoryRealm
  - Wraps the generic ActiveDirectoryRealm provided by Shiro. This allows for enhanced logging as well as isolation of all realms in a single package, which enables easier import by consuming servlets.
  - Disabled out of the box.
- KeystoneAuthRealm
  - This realm authenticates OpenDaylight users against the OpenStack's Keystone server by using the Keystone's Identity API v3 or later.
  - Disabled out of the box.

Note: More than one Realm implementation can be specified. Realms are attempted in order until authentication succeeds or all realm sources are exhausted. Edit the **securityManager.realms = $tokenAuthRealm** property in shiro.ini and add all the realms needed separated by commas.

1.7.1 TokenAuthRealm

How it works

The TokenAuthRealm is the default Authorization Realm deployed in OpenDaylight. TokenAuthRealm uses a direct authentication mechanism as shown in the following picture:

A user presents some credentials (e.g., username/password) directly to the OpenDaylight controller token endpoint /oauth2/token and receives an access token, which then can be used to access protected resources on the controller.

How to access the H2 database

The H2 database provides an optional front-end Web interface, which can be very useful for new users. From the KARAF_HOME directory, you can run the following command to enable the user interface:

```
java -cp ./data/cache/org.eclipse.osgi/bundles/217/1/.cp/h2-1.4.185.jar
    org.h2.tools.Server -trace -pg -web -webAllowOthers -baseDir `pwd`
```

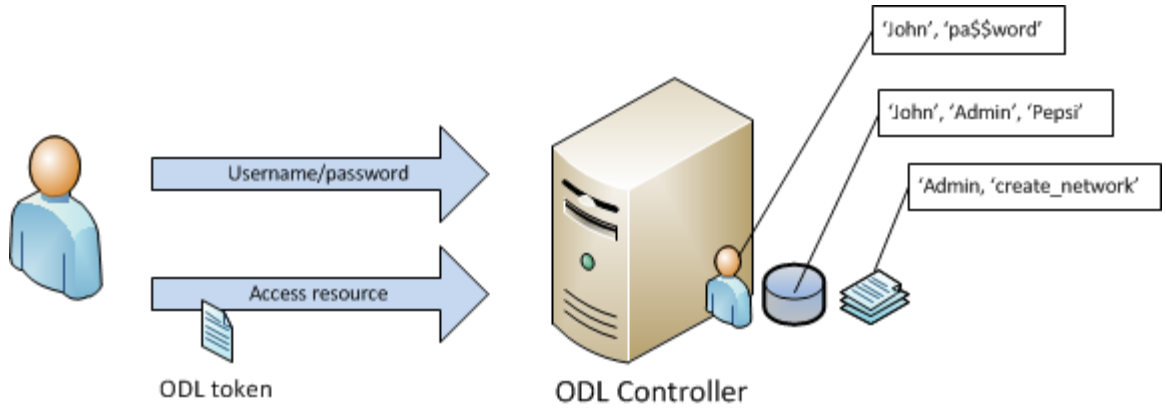

Fig. 1: TokenAuthRealm direct authentication mechanism

You can navigate to the following and login via the browser:

```
http://{IP}:8082/
```

1.7.2 ODLJndiLdapRealm

How it works

LDAP integration is provided in order to externalize identity management. This configuration allows federation with an external LDAP server. The user's OpenDaylight role parameters are mapped to corresponding LDAP attributes as specified by the groupRolesMap. Thus, an LDAP operator can provision attributes for LDAP users that support different OpenDaylight role structures.

1.7.3 ODLJndiLdapRealmAuthNOnly

How it works

This is useful for setups where all LDAP users are allowed equal access.

1.7.4 KeystoneAuthRealm

How it works

This realm authenticates OpenDaylight users against the OpenStack's Keystone server. This realm uses the Keystone's Identity API v3 or later.

As can shown on the above diagram, once configured, all the RESTCONF APIs calls will require sending **user**, **password** and optionally **domain** (1). Those credentials are used to authenticate the call against the Keystone server (2) and, if the authentication succeeds, the call will proceed to the MDSAL (3). The credentials must be provisioned in advance within the Keystone Server. The user and password are mandatory, while the domain is optional, in case it is not provided within the REST call, the realm will default to (**Default**), which is hard-coded. The default domain can be also configured through the *shiro.ini* file (see the *AAA User Guide*).

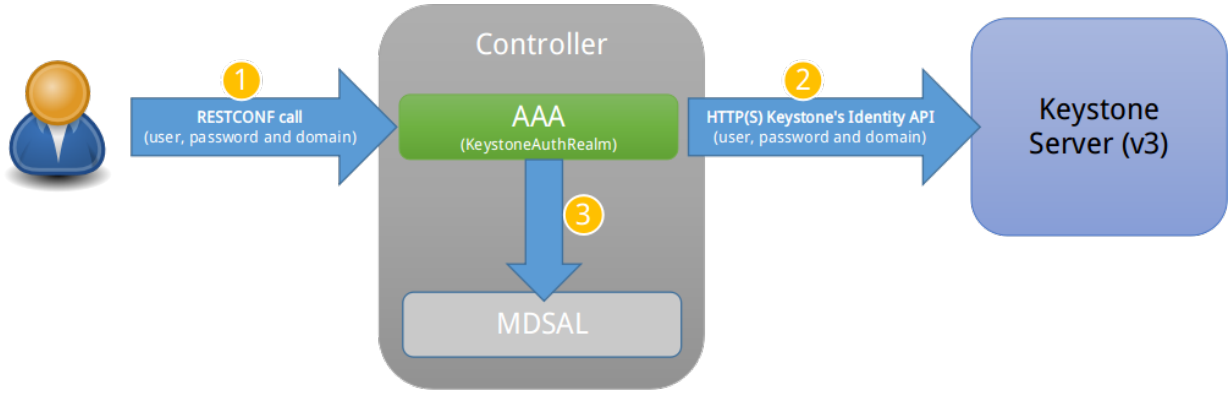

Fig. 2: KeystoneAuthRealm authentication/authorization mechanism

The protocol between the Controller and the Keystone Server (2) can be either HTTPS or HTTP. In order to use HTTPS the Keystone Server's certificate must be exported and imported on the Controller (see the *Certificate Management* section).

1.8 Authorization Configuration

OpenDaylight supports two authorization engines at present, both of which are roughly similar in behavior:

- Shiro-Based Authorization
- MDSAL-Based Dynamic Authorization

Note: The preferred mechanism for configuring AAA Authentication is the MDSAL-Based Dynamic Authorization. Read the following section.

1.8.1 Shiro-Based Static Authorization

OpenDaylight AAA has support for Role Based Access Control (RBAC) based on the Apache Shiro permissions system. Configuration of the authorization system is done off-line; authorization currently cannot be configured after the controller is started. The Authorization provided by this mechanism is aimed towards supporting coarse-grained security policies, the MDSAL-Based mechanism allows for a more robust configuration capabilities. Shiro-based Authorization describes how to configure the Authentication feature in detail.

Note: The Shiro-Based Authorization that uses the *shiro.ini* URLs section to define roles requirements is **deprecated** and **discouraged** since the changes made to the file are only honored on a controller restart.

Shiro-Based Authorization is not **cluster-aware**, so the changes made on the *shiro.ini* file have to be replicated on every controller instance belonging to the cluster.

The URL patterns are matched relative to the Servlet context leaving room for ambiguity, since many endpoints may match (i.e., "/restconf/modules" and "/auth/modules" would both match a "/modules/**" rule).

1.8.2 MDSAL-Based Dynamic Authorization

The MDSAL-Based Dynamic authorization uses the MDSALDynamicAuthorizationFilter engine to restrict access to particular URL endpoint patterns. Users may define a list of policies that are insertion-ordered. Order matters for that list of policies, since the first matching policy is applied. This choice was made to emulate behavior of the Shiro-Based Authorization mechanism.

A **policy** is a key/value pair, where the key is a **resource** (i.e., a "URL pattern") and the value is a list of **permissions** for the resource. The following describes the various elements of a policy:

- **Resource**: the resource is a string URL pattern as outlined by Apache Shiro. For more information, see http://shiro.apache.org/web.html.
- **Description**: an optional description of the URL endpoint and why it is being secured.
- **Permissions list**: a list of permissions for a particular policy. If more than one permission exists in the permissions list they are evaluated using logical "OR". A permission describes the prerequisites to perform HTTP operations on a particular endpoint. The following describes the various elements of a permission:
  - **Role**: the role required to access the target URL endpoint.
  - **Actions list**: a leaf-list of HTTP permissions that are allowed for a Subject possessing the required role.

This an example on how to limit access to the modules endpoint:

```
HTTP Operation:
put URL: /restconf/config/aaa:http-authorization/policies

headers: Content-Type: application/json Accept: application/json

body:
  { "aaa:policies":
    { "aaa:policies":
      [ { "aaa:resource": "/restconf/modules/**",
        "aaa:permissions": [ { "aaa:role": "admin",
                               "aaa:actions": [ "get",
                                                "post",
                                                "put",
                                                "patch",
                                                "delete"
                                              ]
                             }
                           ]
        }
      ]
    }
  }
```

The above example locks down access to the modules endpoint (and any URLS available past modules) to the "admin" role. Thus, an attempt from the OOB *admin* user will succeed with 2XX HTTP status code, while an attempt from the OOB *user* user will fail with HTTP status code 401, as the user *user* is not granted the "admin" role.

1.9 Accounting Configuration

Accounting is handled through the standard slf4j logging mechanisms used by the rest of OpenDaylight. Thus, one can control logging verbosity through manipulating the log levels for individual packages and classes directly through the Karaf console, JMX, or etc/org.ops4j.pax.logging.cfg. In normal operations, the default levels exposed do not provide much information about AAA services; this is due to the fact that logging can severely degrade performance.

All AAA logging is output to the standard karaf.log file. For debugging purposes (i.e., to enable maximum verbosity), issue the following command:

```
log:set TRACE org.opendaylight.aaa
```

1.9.1 Enable Successful/Unsuccessful Authentication Attempts Logging

By default, successful/unsuccessful authentication attempts are NOT logged. This is due to the fact that logging can severely decrease REST performance.

It is possible to add custom AuthenticationListener(s) to the Shiro-based configuration, allowing different ways to listen for successful/unsuccessful authentication attempts. Custom AuthenticationListener(s) must implement the org.apache.shiro.authc.AuthenticationListener interface.

1.10 Certificate Management

The **Certificate Management Service** is used to manage the keystores and certificates at the OpenDaylight distribution to easily provides the TLS communication.

The Certificate Management Service managing two keystores:

1. **OpenDaylight Keystore** which holds the OpenDaylight distribution certificate self sign certificate or signed certificate from a root CA based on generated certificate request.
2. **Trust Keystore** which holds all the network nodes certificates that shall to communicate with the OpenDaylight distribution through TLS communication.

The Certificate Management Service stores the keystores (OpenDaylight & Trust) as *.jks* files under configuration/ssl/ directory. Also the keystores could be stored at the MD-SAL datastore in case OpenDaylight distribution running at cluster environment. When the keystores are stored at MD-SAL, the Certificate Management Service rely on the **Encryption-Service** to encrypt the keystore data before storing it to MD-SAL and decrypted at runtime.

1.10.1 How to use the Certificate Management Service to manage the TLS communication

The following are the steps to configure the TLS communication within your feature or module:

1. It is assumed that there exists an already created OpenDaylight distribution project following this guide.
2. In the implementation bundle the following artifact must be added to its *pom.xml* file as dependency.

```
<dependency>
  <groupId>org.opendaylight.aaa</groupId>
  <artifactId>aaa-cert</artifactId>
  <version>0.5.0-SNAPSHOT</version>
</dependency>
```

3. Using the provider class in the implementation bundle needs to define a variable holding the Certificate Manager Service as in the following example:

```
import org.opendaylight.aaa.cert.api.ICertificateManager;
import org.opendaylight.controller.md.sal.binding.api.DataBroker;

public class UseCertManagerExampleProvider {
```

(continues on next page)

(continued from previous page)

```
  private final DataBroker dataBroker;
  private final ICertificateManager caManager;

  public EncSrvExampleProvider(final DataBroker dataBroker, final ICertificateManager␣
↪caManager) {
    this.dataBroker = dataBroker;
    this.caManager = caManager;
  }
  public SSLEngine createSSLEngine() {
    final SSLContext sslContext = caManager.getServerContext();
    if (sslContext != null) {
      final SSLEngine sslEngine = sslContext.createSSLEngine();
      sslEngine.setEnabledCipherSuites(caManager.getCipherSuites());
      // DO the Implementation
      return sslEngine;
    }
  }
  public void init() {
      // TODO
  }
  public void close() {
      // TODO
  }
}
```

4. The Certificate Manager Service provides two main methods that are needed to establish the *SSLEngine* object, *getServerContext()* and *getCipherSuites()* as the above example shows. It also provides other methods such as *getODLKeyStore()* and *getTrustKeyStore()* that gives access to the OpenDaylight and Trust keystores.

5. Now the *ICertificateManager* need to be passed as an argument to the *UseCertManagerExampleProvider* within the implementation bundle blueprint configuration. The following example shows how:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:odl="http://opendaylight.org/xmlns/blueprint/v1.0.0"
  odl:use-default-for-reference-types="true">
  <reference id="dataBroker"
    interface="org.opendaylight.controller.md.sal.binding.api.DataBroker"
    odl:type="default" />
  <reference id="aaaCertificateManager"
    interface="org.opendaylight.aaa.cert.api.ICertificateManager"
    odl:type="default-certificate-manager" />
  <bean id="provider"
    class="org.opendaylight.UseCertManagerExample.impl.UseCertManagerExampleProvider"
    init-method="init" destroy-method="close">
    <argument ref="dataBroker" />
    <argument ref="aaaCertificateManager" />
  </bean>
</blueprint>
```

6. After finishing the bundle implementation the feature module needs to be updated to include the *aaa-cert* feature in its feature bundle pom.xml file.

```
<properties>
  <aaa.version>0.5.0-SNAPSHOT</aaa.version>
</properties>
<dependency>
  <groupId>org.opendaylight.aaa</groupId>
```

(continues on next page)

(continued from previous page)

```
  <artifactId>features-aaa</artifactId>
  <version>${aaa.version}</version>
  <classifier>features</classifier>
  <type>xml</type>
</dependency>
```

7. Now, in the feature.xml file add the Certificate Manager Service feature and its repository.

```
<repository>mvn:org.opendaylight.aaa/features-aaa/{VERSION}/xml/features</repository>
```

The Certificate Manager Service feature can be included inside the implementation bundle feature as shown in the following example:

```
<feature name='odl-UseCertManagerExample' version='${project.version}'
  description='OpenDaylight :: UseCertManagerExample'>
  <feature version='${mdsal.version}'>odl-mdsal-broker</feature>
  <feature version='${aaa.version}'>odl-aaa-cert</feature>
  <bundle>mvn:org.opendaylight.UseCertManagerExample/UseCertManagerExample-impl/
↪{VERSION}</bundle>
</feature>
```

8. Now the project can be built and the OpenDaylight distribution started to continue with the configuration process. See the User Guide for more details.

1.11 Encryption Service

The **AAA Encryption Service** is used to encrypt the OpenDaylight users' passwords and TLS communication certificates. This section shows how to use the AAA Encryption Service with an OpenDaylight distribution project to encrypt data.

1. It is assumed that there exists an already created OpenDaylight distribution project following this guide.
2. In the implementation bundle the following artifact must be added to its *pom.xml* file as dependency.

```
<dependency>
  <groupId>org.opendaylight.aaa</groupId>
  <artifactId>aaa-encrypt-service</artifactId>
  <version>0.5.0-SNAPSHOT</version>
</dependency>
```

3. Using the provider class in the implementation bundle needs to define a variable holding the Encryption Service as in the following example:

```
import org.opendaylight.aaa.encrypt.AAAEncryptionService;
import org.opendaylight.controller.md.sal.binding.api.DataBroker;

public class EncSrvExampleProvider {
private final DataBroker dataBroker;
  private final AAAEncryptionService encryService;

  public EncSrvExampleProvider(final DataBroker dataBroker, final␣
↪AAAEncryptionService encryService) {
    this.dataBroker = dataBroker;
    this.encryService = encryService;
```

(continues on next page)

(continued from previous page)

```
  }
  public void init() {
    // TODO
  }
  public void close() {
    // TODO
  }
}
```

The AAAEncryptionService can be used to encrypt and decrypt any data based on project's needs.

4. Now the *AAAEncryptionService* needs to be passed as an argument to the *EncSrvExampleProvider* within the implementation bundle blueprint configuration. The following example shows how:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:odl="http://opendaylight.org/xmlns/blueprint/v1.0.0"
  odl:use-default-for-reference-types="true">
  <reference id="dataBroker"
    interface="org.opendaylight.controller.md.sal.binding.api.DataBroker"
    odl:type="default" />
  <reference id="encryService" interface="org.opendaylight.aaa.encrypt.
↪AAAEncryptionService"/>
  <bean id="provider"
    class="org.opendaylight.EncSrvExample.impl.EncSrvExampleProvider"
    init-method="init" destroy-method="close">
    <argument ref="dataBroker" />
    <argument ref="encryService" />
  </bean>
</blueprint>
```

5. After finishing the bundle implementation the feature module needs to be updated to include the *aaa-encryption-service* feature in its feature bundle pom.xml file.

```
<dependency>
  <groupId>org.opendaylight.aaa</groupId>
  <artifactId>features-aaa</artifactId>
  <version>${aaa.version}</version>
  <classifier>features</classifier>
  <type>xml</type>
</dependency>
```

It is also necessary to add the *aaa.version* in the properties section:

```
<properties>
  <aaa.version>0.5.0-SNAPSHOT</aaa.version>
</properties>
```

6. Now, in the feature.xml file add the Encryption Service feature and its repository.

```
<repository>mvn:org.opendaylight.aaa/features-aaa/{VERSION}/xml/features</repository>
```

The Encryption Service feature can be included inside the implementation bundle feature as shown in the following example:

```
<feature name='odl-EncSrvExample' version='${project.version}' description=
↪'OpenDaylight :: EncSrvExample'>
  <feature version='${mdsal.version}'>odl-mdsal-broker</feature>
```

(continues on next page)

(continued from previous page)

```
  <feature version='${aaa.version}'>odl-aaa-encryption-service</feature>
  <feature version='${project.version}'>odl-EncSrvExample-api</feature>
  <bundle>mvn:org.opendaylight.EncSrvExample/EncSrvExample-impl/{VERSION}</bundle>
</feature>
```

7. Now the project can be built and the OpenDaylight distribution started to continue with the configuration process. See the User Guide for more details.

CHAPTER 2

Authentication, Authorization and Accounting (AAA) Services

2.1 Overview

Authentication, Authorization and Accounting (AAA) is a term for a framework controlling access to resources, enforcing policies to use those resources and auditing their usage. These processes are the fundamental building blocks for effective network management and security.

Authentication provides a way of identifying a user, typically by having the user enter a valid user name and valid password before access is granted. The process of authentication is based on each user having a unique set of criteria for gaining access. The AAA framework compares a user's authentication credentials with other user credentials stored in a database. If the credentials match, the user is granted access to the network. If the credentials don't match, authentication fails and access is denied.

Authorization is the process of finding out what an authenticated user is allowed to do within the system, which tasks can do, which API can call, etc. The authorization process determines whether the user has the authority to perform such actions.

Accounting is the process of logging the activity of an authenticated user, for example, the amount of data a user has sent and/or received during a session, which APIs called, etc.

2.1.1 Terms And Definitions

AAA Authentication, Authorization and Accounting.

Token A claim of access to a group of resources on the controller.

Domain A group of resources, direct or indirect, physical, logical, or virtual, for the purpose of access control.

User A person who either owns or has access to a resource or group of resources on the controller.

Role Opaque representation of a set of permissions, which is merely a unique string as admin or guest.

Credential Proof of identity such as user name and password, OTP, biometrics, or others.

Client A service or application that requires access to the controller.

Claim A data set of validated assertions regarding a user, e.g. the role, domain, name, etc.

Grant It is the entity associating a user with his role and domain.

IdP Identity Provider.

TLS Transport Layer Security

CLI Command Line Interface

2.2 Security Framework for AAA services

Since Boron release, the OpenDaylight's AAA services are based on the Apache Shiro Java Security Framework. The main configuration file for AAA is located at "etc/shiro.ini" relative to the OpenDaylight Karaf home directory.

2.3 How to enable AAA

AAA is enabled through installing the odl-aaa-shiro feature. The vast majority of OpenDaylight's northbound APIs (and all RESTCONF APIs) are protected by AAA by default when installing the +odl-restconf+ feature, since the odl-aaa-shiro is automatically installed as part of them. In the cases that APIs are *not* protected by AAA, this will be noted in the per-project release notes.

2.4 How to disable AAA

Edit the "etc/shiro.ini" file and replace the following:

```
/** = authcBasic
```

with

```
/** = anon
```

Then restart the Karaf process.

2.5 AAA Realms

AAA plugin utilizes the Shiro Realms to support pluggable authentication & authorization schemes. There are two parent types of realms:

- AuthenticatingRealm
  - Provides no Authorization capability.
  - Users authenticated through this type of realm are treated equally.
- AuthorizingRealm
  - AuthorizingRealm is a more sophisticated AuthenticatingRealm, which provides the additional mechanisms to distinguish users based on roles.
  - Useful for applications in which roles determine allowed capabilities.

OpenDaylight contains five implementations:

- TokenAuthRealm
  - An AuthorizingRealm built to bridge the Shiro-based AAA service with the h2-based AAA implementation.
  - Exposes a RESTful web service to manipulate IdM policy on a per-node basis. If identical AAA policy is desired across a cluster, the backing data store must be synchronized using an out of band method.
  - A python script located at "etc/idmtool" is included to help manipulate data contained in the TokenAuthRealm.
  - Enabled out of the box. This is the realm configured by default.
- ODLJndiLdapRealm
  - An AuthorizingRealm built to extract identity information from IdM data contained on an LDAP server.
  - Extracts group information from LDAP, which is translated into OpenDaylight roles.
  - Useful when federating against an existing LDAP server, in which only certain types of users should have certain access privileges.
  - Disabled out of the box.
- ODLJndiLdapRealmAuthNOnly
  - The same as ODLJndiLdapRealm, except without role extraction. Thus, all LDAP users have equal authentication and authorization rights.
  - Disabled out of the box.
- ODLActiveDirectoryRealm
  - Wraps the generic ActiveDirectoryRealm provided by Shiro. This allows for enhanced logging as well as isolation of all realms in a single package, which enables easier import by consuming servlets.
- KeystoneAuthRealm
  - This realm authenticates OpenDaylight users against the OpenStack's Keystone server.
  - Disabled out of the box.

Note: More than one Realm implementation can be specified. Realms are attempted in order until authentication succeeds or all realm sources are exhausted. Edit the **securityManager.realms = $tokenAuthRealm** property in shiro.ini and add all the realms needed separated by commas.

2.5.1 TokenAuthRealm

How it works

The TokenAuthRealm is the default Authorization Realm deployed in OpenDaylight. TokenAuthRealm uses a direct authentication mechanism as shown in the following picture:

A user presents some credentials (e.g., username/password) directly to the OpenDaylight controller token endpoint /oauth2/token and receives an access token, which then can be used to access protected resources on the controller.

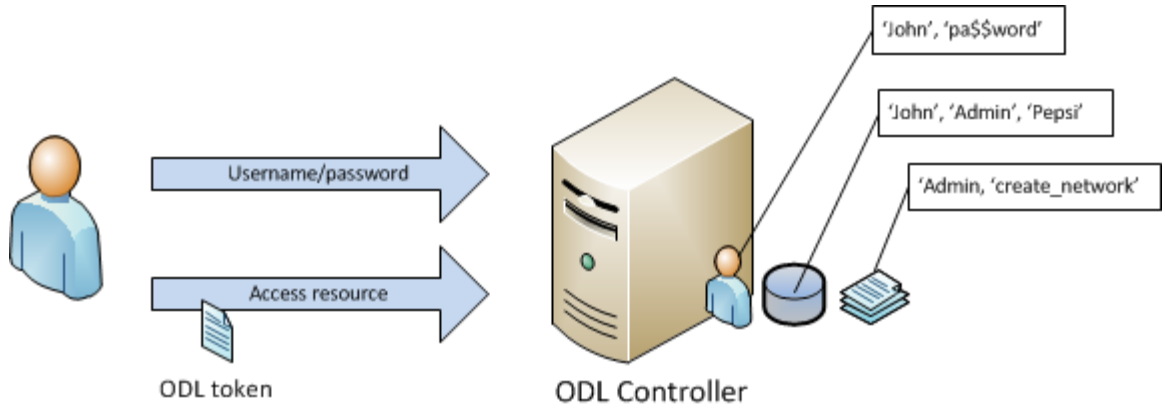

Fig. 1: TokenAuthRealm direct authentication mechanism

Configuring TokenAuthRealm

The TokenAuthRealm stores IdM data in an h2 database on each node. Thus, configuration of a cluster currently requires configuring the desired IdM policy on each node. There are two supported methods to manipulate the TokenAuthRealm IdM configuration:

- idmtool configuration tool
- RESTful Web Service configuration

Idmtool

A utility script located at "etc/idmtool" is used to manipulate the TokenAuthRealm IdM policy. idmtool assumes a single domain, the default one (sdn), since multiple domains are not supported in the Boron release. General usage information for idmtool is derived through issuing the following command:

```
$ python etc/idmtool -h
usage: idmtool [-h] [--target-host TARGET_HOST]
               user
               {list-users,add-user,change-password,delete-user,list-domains,list-
↪roles,add-role,delete-role,add-grant,get-grants,delete-grant}
               ...

positional arguments:
  user                  username for BSC node
  {list-users,add-user,change-password,delete-user,list-domains,list-roles,add-role,
↪delete-role,add-grant,get-grants,delete-grant}
                        sub-command help
    list-users          list all users
    add-user            add a user
    change-password     change a password
    delete-user         delete a user
    list-domains        list all domains
    list-roles          list all roles
    add-role            add a role
    delete-role         delete a role
    add-grant           add a grant
    get-grants          get grants for userid on sdn
    delete-grant        delete a grant
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  --target-host TARGET_HOST
                        target host node
```

Add a user

```
python etc/idmtool admin add-user newUser
Password:
Enter new password:
Re-enter password:
add_user(admin)

command succeeded!

json:
{
    "description": "",
    "domainid": "sdn",
    "email": "",
    "enabled": true,
    "name": "newUser",
    "password": "**********",
    "salt": "**********",
    "userid": "newUser@sdn"
}
```

Note: AAA redacts the password and salt fields for security purposes.

Delete a user

```
$ python etc/idmtool admin delete-user newUser@sdn
Password:
delete_user(newUser@sdn)

command succeeded!
```

List all users

```
$ python etc/idmtool admin list-users
Password:
list_users

command succeeded!

json:
{
```

(continues on next page)

(continued from previous page)

```
    "users": [
        {
            "description": "user user",
            "domainid": "sdn",
            "email": "",
            "enabled": true,
            "name": "user",
            "password": "**********",
            "salt": "**********",
            "userid": "user@sdn"
        },
        {
            "description": "admin user",
            "domainid": "sdn",
            "email": "",
            "enabled": true,
            "name": "admin",
            "password": "**********",
            "salt": "**********",
            "userid": "admin@sdn"
        }
    ]
}
```

Change a user's password

```
$ python etc/idmtool admin change-password admin@sdn
Password:
Enter new password:
Re-enter password:
change_password(admin)

command succeeded!

json:
{
    "description": "admin user",
    "domainid": "sdn",
    "email": "",
    "enabled": true,
    "name": "admin",
    "password": "**********",
    "salt": "**********",
    "userid": "admin@sdn"
}
```

Add a role

```
$ python etc/idmtool admin add-role network-admin
Password:
add_role(network-admin)
```

(continues on next page)

(continued from previous page)

```
command succeeded!

json:
{
    "description": "",
    "domainid": "sdn",
    "name": "network-admin",
    "roleid": "network-admin@sdn"
}
```

Delete a role

```
$ python etc/idmtool admin delete-role network-admin@sdn
Password:
delete_role(network-admin@sdn)

command succeeded!
```

List all roles

```
$ python etc/idmtool admin list-roles
Password:
list_roles

command succeeded!

json:
{
    "roles": [
        {
            "description": "a role for admins",
            "domainid": "sdn",
            "name": "admin",
            "roleid": "admin@sdn"
        },
        {
            "description": "a role for users",
            "domainid": "sdn",
            "name": "user",
            "roleid": "user@sdn"
        }
    ]
}
```

List all domains

```
$ python etc/idmtool admin list-domains
Password:
list_domains
```

(continues on next page)

(continued from previous page)

```
command succeeded!

json:
{
    "domains": [
        {
            "description": "default odl sdn domain",
            "domainid": "sdn",
            "enabled": true,
            "name": "sdn"
        }
    ]
}
```

Add a grant

```
$ python etc/idmtool admin add-grant user@sdn admin@sdn
Password:
add_grant(userid=user@sdn,roleid=admin@sdn)

command succeeded!

json:
{
    "domainid": "sdn",
    "grantid": "user@sdn@admin@sdn@sdn",
    "roleid": "admin@sdn",
    "userid": "user@sdn"
}
```

Delete a grant

```
$ python etc/idmtool admin delete-grant user@sdn admin@sdn
Password:
http://localhost:8181/auth/v1/domains/sdn/users/user@sdn/roles/admin@sdn
delete_grant(userid=user@sdn,roleid=admin@sdn)

command succeeded!
```

Get grants for a user

```
python etc/idmtool admin get-grants admin@sdn
Password:
get_grants(admin@sdn)

command succeeded!

json:
{
    "roles": [
```

(continues on next page)

(continued from previous page)

```
        {
            "description": "a role for users",
            "domainid": "sdn",
            "name": "user",
            "roleid": "user@sdn"
        },
        {
            "description": "a role for admins",
            "domainid": "sdn",
            "name": "admin",
            "roleid": "admin@sdn"
        }
    ]
}
```

Configuration using the RESTful Web Service

The TokenAuthRealm IdM policy is fully configurable through a RESTful web service. Full documentation for manipulating AAA IdM data is located online (https://wiki.opendaylight.org/images/0/00/AAA_Test_Plan.docx), and a few examples are included in this guide:

Get All Users

```
curl -u admin:admin http://localhost:8181/auth/v1/users
OUTPUT:
{
    "users": [
        {
            "description": "user user",
            "domainid": "sdn",
            "email": "",
            "enabled": true,
            "name": "user",
            "password": "**********",
            "salt": "**********",
            "userid": "user@sdn"
        },
        {
            "description": "admin user",
            "domainid": "sdn",
            "email": "",
            "enabled": true,
            "name": "admin",
            "password": "**********",
            "salt": "**********",
            "userid": "admin@sdn"
        }
    ]
}
```

Create a User

```
curl -u admin:admin -X POST -H "Content-Type: application/json" --data-binary @./user.
↪json http://localhost:8181/auth/v1/users
PAYLOAD:
{
    "name": "ryan",
    "userid": "ryan@sdn",
    "password": "ryan",
    "domainid": "sdn",
    "description": "Ryan's User Account",
    "email": "ryandgoulding@gmail.com"
}

OUTPUT:
{
    "userid":"ryan@sdn",
    "name":"ryan",
    "description":"Ryan's User Account",
    "enabled":true,
    "email":"ryandgoulding@gmail.com",
    "password":"**********",
    "salt":"**********",
    "domainid":"sdn"
}
```

Create an OAuth2 Token For Admin Scoped to SDN

```
curl -d 'grant_type=password&username=admin&password=a&scope=sdn' http://
↪localhost:8181/oauth2/token

OUTPUT:
{
    "expires_in":3600,
    "token_type":"Bearer",
    "access_token":"5a615fbc-bcad-3759-95f4-ad97e831c730"
}
```

Use an OAuth2 Token

```
curl -H "Authorization: Bearer 5a615fbc-bcad-3759-95f4-ad97e831c730" http://
↪localhost:8181/auth/v1/domains
{
    "domains":
    [
        {
            "domainid":"sdn",
            "name":"sdn”,
            "description":"default odl sdn domain",
            "enabled":true
        }
    ]
}
```

Token Store Configuration Parameters

Edit the file "etc/opendaylight/karaf/08-authn-config.xml" and edit the following: .**timeToLive**: Configure the maximum time, in milliseconds, that tokens are to be cached. Default is 360000. Save the file.

2.5.2 ODLJndiLdapRealm

How it works

LDAP integration is provided in order to externalize identity management. This configuration allows federation with an external LDAP server. The user's OpenDaylight role parameters are mapped to corresponding LDAP attributes as specified by the groupRolesMap. Thus, an LDAP operator can provision attributes for LDAP users that support different OpenDaylight role structures.

Configuring ODLJndiLdapRealm

To configure LDAP parameters, modify "etc/shiro.ini" parameters to include the ODLJndiLdapRealm:

```
# OpenDaylight provides a few LDAP implementations, which are disabled out of the box.
# ODLJndiLdapRealm includes authorization functionality based on LDAP elements
# extracted through and LDAP search.  This requires a bit of knowledge about
# how your LDAP system is setup.  An example is provided below:
ldapRealm = org.opendaylight.aaa.shiro.realm.ODLJndiLdapRealm
ldapRealm.userDnTemplate = uid={0},ou=People,dc=DOMAIN,dc=TLD
ldapRealm.contextFactory.url = ldap://<URL>:389
ldapRealm.searchBase = dc=DOMAIN,dc=TLD
ldapRealm.ldapAttributeForComparison = objectClass
ldapRealm.groupRolesMap = "Person":"admin"
# ...
# further down in the file...
# Stacked realm configuration;  realms are round-robbined until authentication␣
↪succeeds or realm sources are exhausted.
securityManager.realms = $tokenAuthRealm, $ldapRealm
```

2.5.3 ODLJndiLdapRealmAuthNOnly

How it works

This is useful for setups where all LDAP users are allowed equal access.

Configuring ODLJndiLdapRealmAuthNOnly

Edit the "etc/shiro.ini" file and modify the following:

```
ldapRealm = org.opendaylight.aaa.shiro.realm.ODLJndiLdapRealm
ldapRealm.userDnTemplate = uid={0},ou=People,dc=DOMAIN,dc=TLD
ldapRealm.contextFactory.url = ldap://<URL>:389
# ...
# further down in the file...
# Stacked realm configuration;  realms are round-robbined until authentication␣
↪succeeds or realm sources are exhausted.
securityManager.realms = $tokenAuthRealm, $ldapRealm
```

2.5.4 KeystoneAuthRealm

How it works

This realm authenticates OpenDaylight users against the OpenStack's Keystone server. This realm uses the Keystone's Identity API v3 or later.

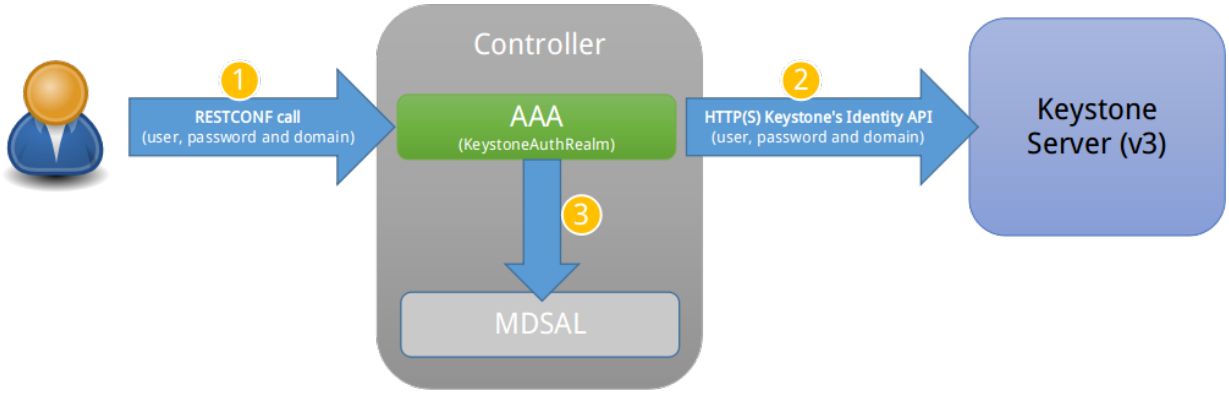

Fig. 2: KeystoneAuthRealm authentication/authorization mechanism

As can shown on the above diagram, once configured, all the RESTCONF APIs calls will require sending **user**, **password** and optionally **domain** (1). Those credentials are used to authenticate the call against the Keystone server (2) and, if the authentication succeeds, the call will proceed to the MDSAL (3). The credentials must be provisioned in advance within the Keystone Server. The user and password are mandatory, while the domain is optional, in case it is not provided within the REST call, the realm will default to (**Default**), which is hard-coded. The default domain can be also configured through the *shiro.ini* file (see the *AAA User Guide*).

The protocol between the Controller and the Keystone Server (2) can be either HTTPS or HTTP. In order to use HTTPS the Keystone Server's certificate must be exported and imported on the Controller (see the *Certificate Management* section).

Configuring KeystoneAuthRealm

Edit the "etc/shiro.ini" file and modify the following:

```
# The KeystoneAuthRealm allows for authentication/authorization against an
# OpenStack's Keystone server. It uses the Identity's API v3 or later.
keystoneAuthRealm = org.opendaylight.aaa.shiro.realm.KeystoneAuthRealm
# The URL where the Keystone server exposes the Identity's API v3 the URL
# can be either HTTP or HTTPS and it is mandatory for this realm.
keystoneAuthRealm.url = https://<host>:<port>
# Optional parameter to make the realm verify the certificates in case of HTTPS
#keystoneAuthRealm.sslVerification = true
# Optional parameter to set up a default domain for requests using credentials
# without domain, uncomment in case you want a different value from the hard-coded
# one "Default"
#keystoneAuthRealm.defaultDomain = Default
```

Once configured the realm, the mandatory fields are the fully quallified name of the class implementing the realm *keystoneAuthRealm* and the endpoint where the Keystone Server is listening *keystoneAuthRealm.url*.

The optional parameter *keystoneAuthRealm.sslVerification* specifies whether the realm has to verify the SSL certificate or not. The optional parameter *keystoneAuthRealm.defaultDomain* allows to use a different default domain from the hard-coded one *"Default"*.

2.6 Authorization Configuration

OpenDaylight supports two authorization engines at present, both of which are roughly similar in behavior:

- Shiro-Based Authorization
- MDSAL-Based Dynamic Authorization

Note: The preferred mechanism for configuring AAA Authentication is the MDSAL-Based Dynamic Authorization. Read the following section.

2.6.1 Shiro-Based Static Authorization

OpenDaylight AAA has support for Role Based Access Control (RBAC) based on the Apache Shiro permissions system. Configuration of the authorization system is done off-line; authorization currently cannot be configured after the controller is started. The Authorization provided by this mechanism is aimed towards supporting coarse-grained security policies, the MDSAL-Based mechanism allows for a more robust configuration capabilities. Shiro-based Authorization describes how to configure the Authentication feature in detail.

Note: The Shiro-Based Authorization that uses the *shiro.ini* URLs section to define roles requirements is **deprecated** and **discouraged** since the changes made to the file are only honored on a controller restart.

Shiro-Based Authorization is not **cluster-aware**, so the changes made on the *shiro.ini* file have to be replicated on every controller instance belonging to the cluster.

The URL patterns are matched relative to the Servlet context leaving room for ambiguity, since many endpoints may match (i.e., "/restconf/modules" and "/auth/modules" would both match a "/modules/**" rule).

Enable "admin" Role Based Access to the IdMLight RESTful web service

Edit the "etc/shiro.ini" configuration file and add "/auth/v1/**= authcBasic, roles[admin]" above the line "/** = authcBasic" within the "urls" section.

```
/auth/v1/** = authcBasic, roles[admin]
/** = authcBasic
```

This will restrict the idmlight rest endpoints so that a grant for admin role must be present for the requesting user.

Note: The ordering of the authorization rules above is important!

2.6.2 MDSAL-Based Dynamic Authorization

The MDSAL-Based Dynamic authorization uses the MDSALDynamicAuthorizationFilter engine to restrict access to particular URL endpoint patterns. Users may define a list of policies that are insertion-ordered. Order matters for that list of policies, since the first matching policy is applied. This choice was made to emulate behavior of the Shiro-Based Authorization mechanism.

A **policy** is a key/value pair, where the key is a **resource** (i.e., a "URL pattern") and the value is a list of **permissions** for the resource. The following describes the various elements of a policy:

- **Resource**: the resource is a string URL pattern as outlined by Apache Shiro. For more information, see http://shiro.apache.org/web.html.
- **Description**: an optional description of the URL endpoint and why it is being secured.
- **Permissions list**: a list of permissions for a particular policy. If more than one permission exists in the permissions list they are evaluated using logical "OR". A permission describes the prerequisites to perform HTTP operations on a particular endpoint. The following describes the various elements of a permission:
  - **Role**: the role required to access the target URL endpoint.
  - **Actions list**: a leaf-list of HTTP permissions that are allowed for a Subject possessing the required role.

This an example on how to limit access to the modules endpoint:

```
HTTP Operation:
put URL: /restconf/config/aaa:http-authorization/policies

headers: Content-Type: application/json Accept: application/json

body:
  { "aaa:policies":
    { "aaa:policies":
      [ { "aaa:resource": "/restconf/modules/**",
        "aaa:permissions": [ { "aaa:role": "admin",
                               "aaa:actions": [ "get",
                                                "post",
                                                "put",
                                                "patch",
                                                "delete"
                                              ]
                             }
                           ]
      }
    ]
    }
  }
```

The above example locks down access to the modules endpoint (and any URLS available past modules) to the "admin" role. Thus, an attempt from the OOB *admin* user will succeed with 2XX HTTP status code, while an attempt from the OOB *user* user will fail with HTTP status code 401, as the user *user* is not granted the "admin" role.

2.7 Accounting Configuration

Accounting is handled through the standard slf4j logging mechanisms used by the rest of OpenDaylight. Thus, one can control logging verbosity through manipulating the log levels for individual packages and classes directly through the Karaf console, JMX, or etc/org.ops4j.pax.logging.cfg. In normal operations, the default levels exposed do not provide much information about AAA services; this is due to the fact that logging can severely degrade performance.

All AAA logging is output to the standard karaf.log file. For debugging purposes (i.e., to enable maximum verbosity), issue the following command:

```
log:set TRACE org.opendaylight.aaa
```

2.7.1 Enable Successful/Unsuccessful Authentication Attempts Logging

By default, successful/unsuccessful authentication attempts are NOT logged. This is due to the fact that logging can severely decrease REST performance. To enable logging of successful/unsuccessful REST attempts, issue the following command in Karaf's console:

```
log:set DEBUG org.opendaylight.aaa.shiro.filters.AuthenticationListener
```

It is possible to add custom AuthenticationListener(s) to the Shiro-based configuration, allowing different ways to listen for successful/unsuccessful authentication attempts. Custom AuthenticationListener(s) must implement the org.apache.shiro.authc.AuthenticationListener interface.

2.8 Certificate Management

The **Certificate Management Service** is used to manage the keystores and certificates at the OpenDaylight distribution to easily provides the TLS communication.

The Certificate Management Service managing two keystores:

1. **OpenDaylight Keystore** which holds the OpenDaylight distribution certificate self sign certificate or signed certificate from a root CA based on generated certificate request.
2. **Trust Keystore** which holds all the network nodes certificates that shall to communicate with the OpenDaylight distribution through TLS communication.

The Certificate Management Service stores the keystores (OpenDaylight & Trust) as *.jks* files under configuration/ssl/ directory. Also the keystores could be stored at the MD-SAL datastore in case OpenDaylight distribution running at cluster environment. When the keystores are stored at MD-SAL, the Certificate Management Service rely on the **Encryption-Service** to encrypt the keystore data before storing it to MD-SAL and decrypted at runtime.

2.8.1 How to use the Certificate Management Service to manage the TLS communication

The following are the steps to configure the TLS communication:

1. After starting the distribution, the *odl-aaa-cert* feature has to get installed. Use the following command at Karaf CLI to check.

```
opendaylight-user@root>feature:list -i | grep aaa-cert
odl-aaa-cert | 0.5.0-SNAPSHOT | x | odl-aaa-0.5.0-SNAPSHOT | OpenDaylight :: AAA ::␣
↪aaa certificate Service
```

2. The initial configuration of the Certificate Manager Service exists under the distribution directory etc/opendaylight/datastore/initial/config/aaa-cert-config.xml.

```
<aaa-cert-service-config xmlns="urn:opendaylight:yang:aaa:cert">
  <use-config>false</use-config>
  <use-mdsal>false</use-mdsal>
  <bundle-name>opendaylight</bundle-name>
  <ctlKeystore>
    <name>ctl.jks</name>
    <alias>controller</alias>
    <store-password/>
    <dname>CN=ODL, OU=Dev, O=LinuxFoundation, L=QC Montreal, C=CA</dname>
    <validity>365</validity>
```

(continues on next page)

(continued from previous page)

```
    <key-alg>RSA</key-alg>
    <sign-alg>SHA1WithRSAEncryption</sign-alg>
     <keysize>1024</keysize>
     <cipher-suites>
       <suite-name />
     </cipher-suites>
  </ctlKeystore>
  <trustKeystore>
    <name>truststore.jks</name>
    <store-password/>
  </trustKeystore>
</aaa-cert-service-config>
```

Now as it is explained above, the Certificate Manager Service support two mode of operations; cluster mode and single mode. To use the single mode change the use-config to true and it is recommended as long as there is no need for cluster environment. To use the cluster mode change the use-config and use-mdsal configurations to true and the keystores will be stored and shard across the cluster nodes within the MD-SAL datastore.

The initial password become randomly generated when the *aaa-cert* feature is installed.

The cipher suites can be restricted by changing the **<cipher-suites>** configuration, however, the JDK has to be upgraded by installing the Java Cryptography Extension policy.

```
<cipher-suites>
  <suite-name>TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384</suite-name>
</cipher-suites>
  <cipher-suites>
<suite-name>TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384</suite-name>
  </cipher-suites>
<cipher-suites>
  <suite-name>TLS_DHE_RSA_WITH_AES_256_GCM_SHA384</suite-name>
</cipher-suites>
```

3. The new configurations will take affect after restarting the distribution.

4. Now to add or get certificate to the OpenDaylight and Trust keystores, the Certificate Manager Service provides the following RPCs.

```
a) Set the node certificate that will communicate with OpeDaylight through TLS
connection.
POST /operations/aaa-cert-rpc:setNodeCertifcate
{
  "input": {
    "node-cert": "string",
    "node-alias": "string"
  }
}
```

```
b) Get the node certificate based on node alias.
POST /operations/aaa-cert-rpc:getNodeCertifcate
{
  "input": {
    "node-alias": "string"
  }
}
```

```
c) Get the OpeDaylight keystore certificate.
POST /operations/aaa-cert-rpc:getODLCertificate
{
  output {
    odl-cert "string"
  }
}
```

```
d) Generate a certificate request from the OpeDaylight keystore to be signed
by a CA.
POST /operations/aaa-cert-rpc:getODLCertificateReq
{
  output {
    odl-cert-req "string"
  }
}
```

```
e) Set the OpeDaylight certificate, the certificate should be generated
based on a certificate request generated from the ODL keystore otherwise the
certificated will not be added.
POST /operations/aaa-cert-rpc:setODLCertificate
{
  "input": {
    "odl-cert-alias": "string",
    "odl-cert": "string"
  }
}
```

Note: The Certificate Manager Service RPCs are allowed only to the Role Admin Users and it could be completely disabled through the shiro.ini config file. Check the URL section at the shiro.ini.

2.9 Encryption Service

The **AAA Encryption Service** is used to encrypt the OpenDaylight's users' passwords and TLS communication certificates. This section shows how to use the AAA Encryption Service with an OpenDaylight distribution project to encrypt data.

The following are the steps to configure the Encryption Service:

1. After starting the distribution, the *aaa-encryption-service* feature has to get installed. Use the following command at Karaf CLI to check.

```
opendaylight-user@root>feature:list -i | grep aaa-encryption-service
odl-aaa-encryption-service | 0.5.0-SNAPSHOT | x | odl-aaa-0.5.0-SNAPSHOT |␣
↪OpenDaylight :: AAA :: Encryption Service
```

2. The initial configuration of the Encryption Service exists under the distribution directory etc/opendaylight/datastore/initial/config/aaa-encrypt-service-config.xml

```
<aaa-encrypt-service-config xmlns="config:aaa:authn:encrypt:service:config">
  <encrypt-key/>
  <encrypt-salt/>
```

(continues on next page)

(continued from previous page)

```
  <encrypt-method>PBKDF2WithHmacSHA1</encrypt-method>
  <encrypt-type>AES</encrypt-type>
  <encrypt-iteration-count>32768</encrypt-iteration-count>
  <encrypt-key-length>128</encrypt-key-length>
  <cipher-transforms>AES/CBC/PKCS5Padding</cipher-transforms>
</aaa-encrypt-service-config>
```

Note: Both the initial encryption key and encryption salt become randomly generated when the *aaa-encryption-service* feature is installed.

3. Finally the new configurations will take affect after restarting the distribution.

2.10 Using the AAA Command Line Interface (CLI)

The AAA offers a CLI through the Karaf's console. This CLI allows the user to configure and use some of the functionalities provided by AAA.

The AAA CLI exists under the **odl-aaa-cli** feature. This feature can be installed by executing the following command.

```
feature:install odl-aaa-cli
```

To check that the installation of the feature succeeded type "aaa" and press *tab* to see the list of available commands under the *aaa* scope.

```
opendaylight-user@root>aaa:
aaa:add-domain           aaa:add-grant            aaa:add-role             aaa:add-
↪user
aaa:change-user-pwd      aaa:export-keystores     aaa:gen-cert-req         aaa:get-
↪cipher-suites
aaa:get-domains          aaa:get-node-cert        aaa:get-odl-cert         aaa:get-
↪roles
aaa:get-tls-protocols    aaa:get-users            aaa:import-keystores     aaa:remove-
↪domain
aaa:remove-grant         aaa:remove-role          aaa:remove-user
```

2.10.1 Add a User

The *add-user* command allows for adding an OpenDaylight user. The following user parameters can be specified.

```
aaa:add-user --name <user name>
             --roleName <role>
             --userDescription <user description>
             --email <user email>
             --domainName <domain name>
```

2.10.2 List available Users

The *get-users* command list all the available users within the Controller.

```
aaa:get-users

  user
  admin
```

2.10.3 Remove a User

The *remove-user* command allows for removing an OpenDaylight user. The command needs the user name as parameter.

```
aaa:remove-user --name <user name>
```

2.10.4 Change the OpenDaylight user password

The *change-user-pwd* command allows for changing the OpenDaylight user's password. It takes the user name as argument then will ask for the given user current password.

```
aaa:change-user-pwd -user admin
  Enter current password:
  Enter new password:
  admin's password has been changed
```

2.10.5 Add a Role

The *add-role* command allows for adding a role to the Controller.

```
aaa:add-role --name <role name>
             --desc <role description>
             --domainName <domain name>
```

2.10.6 List available Roles

The *get-roles* command list all the available roles within the controller.

```
aaa:get-roles

  user
  admin
```

2.10.7 Remove a Role

The *remove-role* command allows for removing an OpenDaylight role. The command needs the role name as parameter. The role will be removed from those users who have it.

```
aaa:remove-role --name <role name>
```

2.10.8 Add a Domain

The *add-domain* command allows for adding a domain to the Controller.

```
aaa:add-domain --name <domain name>
               --desc <domain description>
```

2.10.9 List available Domains

The *get-domains* command list all the available domains within the controller. The system asks for the administrator credentials to execute this command.

```
aaa:get-domains

  sdn
```

2.10.10 Remove a Domain

The *remove-domain* command allows for removing an OpenDaylight role. The command needs the domain name as parameter.

```
aaa:remove-domain --name <domain name>
```

2.10.11 Add a Grant

The *add-grant* command allows for creating a grant for an existing user. The command returns a grant id for that user.

```
aaa:add-grant --userName <user name>
              --domainName <domain name>
              --roleName <role name>
```

2.10.12 Remove a Grant

The *remove-grant* command allows for removing an OpenDaylight grant. This command needs the user name, domain and and role as parameters.

```
aaa:remove-grant --userName <user name>
                 --domainName <domain name>
                 --roleName <role name>
```

2.10.13 Generate Certificate Request

Generate certificate request command will generate a certificate request based on the generated OpenDaylight keystore and print it on the Karaf CLI. The system asks for the keystore password.

```
aaa:gen-cert-req

-----BEGIN CERTIFICATE REQUEST-----
MIIBlzCCAQACAQAwWTELMAkGA1UEBhMCQ0ExFDASBgNVBAcMC1FDIE1vbnRyZWFsMRgwFgYDVQQKDA
```

(continues on next page)

(continued from previous page)

```
9MaW51eEZvdW5kYXRpb24xDDAKBgNVBAsMA0RldjEMMAoGA1UEAwwDT0RMMIGfMA0GCSqGSIb3DQEB
AQUAA4GNADCBiQKBgQCCmLW6j+JLYJM5yAMwscw/CHqPnp5elPa1YtQsHKEAvp1I+mLVtHKZeXeteA
kyp6ORxw6KQ515fcDyQVrRJiSM15jUd27UaFq5ku0+qJeG+Qh2btx+cvNSE7/+cgUWWosKz4Aff5F5
FqR62jLUTNzqCvoaTbZaOnLYVq+O2dYyZwIDAQABMA0GCSqGSIb3DQEBBQUAA4GBADhDr4Jm7gVm/o
p861/FShyw1ZZscxOEl2TprJZiTO6sn3sLptQZv8v52Z+Jm5dAgr7L46c97Xfa+0j6Y4LXNb0f88lL
RG8PxGbk6Tqbjqc0WS+U1Ibc/rcPK4HEN/bcYCn+Na1gLBaFXUPg08ozG6MwqFNeS5Z0jz1W0D9/oiao
-----END CERTIFICATE REQUEST-----
```

2.10.14 Get OpenDaylight Certificate

The *get-odl-certificate* command will print the OpenDaylight certificate at the Karaf CLI. The system asks for the keystore password.

```
aaa:get-odl-cert -storepass <store_password>

-----BEGIN CERTIFICATE-----
MIICKTCCAZKgAwIBAgIEI75RWDANBgkqhkiG9w0BAQUFADBZMQwwCgYDVQQDDANPREwxDDAKBgNVBA
sMA0RldjEYMBYGA1UECgwPTGludXhGb3VuZGF0aW9uMRQwEgYDVQQHDAtRQyBNb250cmVhbDELMAkG
A1UEBhMCQ0EwHhcNMTYxMTMwMTYyNDE3WhcNMTcxMTMwMTYyNDE3WjBZMQwwCgYDVQQDDANPREwxDD
AKBgNVBAsMA0RldjEYMBYGA1UECgwPTGludXhGb3VuZGF0aW9uMRQwEgYDVQQHDAtRQyBNb250cmVh
bDELMAkGA1UEBhMCQ0EwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAIKYtbqP4ktgkznIAzCxzD
8Ieo+enl6U9rVi1CwcoQC+nUj6YtW0cpl5d614CTKno5HHDopDnXl9wPJBWtEmJIzXmNR3btRoWrmS
7T6ol4b5CHZu3H5y81ITv/5yBRZaiwrPgB9/kXkWpHraMtRM3OoK+hpNtlo6cthWr47Z1jJnAgMBAA
EwDQYJKoZIhvcNAQEFBQADgYEAL9DK/P/yEBre3Mg3bICAUAvSvZic+ydDmigWLsY4J3UzKdV2f1jI
s+rQTEgtlHShBf/ed546D49cp3XEzYrcxgILhGXDziCrUK0K1TiYqPTp6FLijjdydGlPpwuMyyV5Y0
iDiRclWuPz2fHbs8WQOWNs6VQ+WaRExtEsEC4qgSo=
-----END CERTIFICATE-----
```

2.10.15 Get Cipher Suites

The *get-cipher-suites* command shows the cipher suites supported by the JVM used by the OpenDaylight controller in TLS communication. For example, here are the Default Ciphers Suites in JDK 8.

```
aaa:get-cipher-suites

  TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
  TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
  TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
```

2.10.16 Get TLS Protocols

The *get-tls-protocols* command shows the TLS protocols supported by the JVM used by the OpenDaylight controller. For example, the JDK 8 supports the following TLS protocols: TLSv1.2 (default), TLSv1.1, TLSv1 and SSLv3.

```
aaa:get-tls-protocols

  TLS_KRB5_WITH_RC4_128_SHA
  TLS_KRB5_WITH_RC4_128_MD5
  TLS_KRB5_WITH_3DES_EDE_CBC_SHA
  TLS_KRB5_WITH_3DES_EDE_CBC_MD5
  TLS_KRB5_WITH_DES_CBC_SHA
```

2.10.17 Get Node Certificate

The *get-node-cert* command prints a certificate for a given network node alias. This command is useful to check if the network node certificate has been added properly to the truest keystore. It takes the certificate alias as arguments.

```
aaa:get-node-cert -alias ovs1
-----BEGIN CERTIFICATE-----
MIICKTCCAZKgAwIBAgIEI75RWDANBgkqhkiG9w0BAQUFADBZMQwwCgYDVQQDDANPREwxDDAKBgNVBA
sMA0RldjEYMBYGA1UECgwPTGludXhGb3VuZGF0aW9uMRQwEgYDVQQHDAtRQyBNb250cmVhbDELMAkG
A1UEBhMCQ0EwHhcNMTYxMTMwMTYyNDE3WhcNMTcxMTMwMTYyNDE3WjBZMQwwCgYDVQQDDANPREwxDD
AKBgNVBAsMA0RldjEYMBYGA1UECgwPTGludXhGb3VuZGF0aW9uMRQwEgYDVQQHDAtRQyBNb250cmVh
bDELMAkGA1UEBhMCQ0EwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAIKYtbqP4ktgkznIAzCxzD
8Ieo+enl6U9rVi1CwcoQC+nUj6YtW0cpl5d614CTKno5HHDopDnXl9wPJBWtEmJIzXmNR3btRoWrmS
7T6ol4b5CHZu3H5y81ITv/5yBRZaiwrPgB9/kXkWpHraMtRM3OoK+hpNtlo6cthWr47Z1jJnAgMBAA
EwDQYJKoZIhvcNAQEFBQADgYEAL9DK/P/yEBre3Mg3bICAUAvSvZic+ydDmigWLsY4J3UzKdV2f1jI
s+rQTEgtlHShBf/ed546D49cp3XEzYrcxgILhGXDziCrUK0K1TiYqPTp6FLijjdydGlPpwuMyyV5Y0
iDiRclWuPz2fHbs8WQOWNs6VQ+WaRExtEsEC4qgSo=
-----END CERTIFICATE-----
```

2.10.18 Export Keystores

The *export-keystores* command exports the default MD-SAL Keystores to .jks files in the default directory for keystores (configuration/ssl/).

```
aaa:export-keystores

  Default directory for keystores is configuration/ssl/
```

2.10.19 Import Keystores

The *import-keystores* command imports the default MD-SAL Keystores. The keystores (odl and trust) should exist under default SSL directory (configuration/ssl/).

```
aaa:import-keystores --trustKeystoreName <name of the trust keystore>
                     --trustKeystorePwd <password for the trust keystore>
                     --odlKeystoreName <name of the ODL keystore>
                     --odlKeystorePwd <password for the ODL keystore>
                     --odlKeystoreAlias <alias of the ODL keystore>
                     --tlsProtocols <list of TLS protocols separated by ','>
                     --cipherSuites <list of Cipher suites separated by ','>
```

Warning: It is strongly recommended to run the history clear command after you execute all the AAA CLI commands so Karaf logs stay clean from any adversary.

```
history -c
```