Setting up the MCP Identity Registry

Table of Contents

Prerequisites

- Java JDK 8 (OpenJDK is preferred, but not a requirement)
- Maven 3+
- A running MySQL/MariaDB instance with databases and users created
- Keycloak 6.0.1
- NGINX
- Docker (if you want to run the identity registry using Docker containers)
- A SMTP server

Introduction

The MCP Identity Registry (MIR) is an identity solution targeted against actors of the maritime domain. It consists of three parts – an identity broker for logging in and federating users from other identity providers, a public key infrastructure for issuing and managing digital certificates and an API for management and access to the two first parts.

For the identity broker Keycloak has been chosen as it is open source and because it is based on OpenID Connect which is a widely used protocol for token based authentication.

This document will provide a generic guideline of how to setup and deploy the MIR on a local system.

The MCP is formerly known as the Maritime Cloud and therefore there will still be references to that in this document.

As the different components of the MIR rely on each other the configuration of one component may rely on the configuration of another component, but this will be documented as thoroughly as possible.

Complementary documentation of setup can be found at https://github.com/MaritimeConnectivityPlatform/IdentityRegistry, https://github.com/MaritimeConnectivityPlatform/MCP-PKI and https://github.com/MaritimeConnectivityPlatform/MCPKeycloakSpi

For easier management of the IR API it is advised to also setup the MCP management portal. The code for this can be found at https://github.com/MaritimeConnectivityPlatform/MCP-Portal. A running instance of this can be found at https://management.maritimecloud.net/.

Setting up the PKI

A part of the MIR is based on public key certificates. For this a custom PKI has been developed to issue and manage certificates for stakeholders.

To set up the PKI you must get the source code for it from https://github.com/MaritimeConnectivityPlatform/MCP-PKI/archive/v0.9.0.zip or 'git clone https://github.com/MaritimeConnectivityPlatform/MCP-PKI.git'. After you have done this you can build the project from the directory storing the code using the following command:

mvn clean install

After this has finished there should now be a new folder called **target**. This folder contains the resulting artifacts of the build process, including the jar file **mc-pki-0.9.0-SNAPSHOT-jar-with-dependencies.jar** which is the artifact that is going to be used to generate the root certificate authority (CA) and sub-CAs for the PKI.

The next step is to generate the root CA. This can be done using the following command where you should of course replace the different fields with information relevant to your setup:

```
java -jar mc-pki-0.9.0-SNAPSHOT-jar-with-dependencies.jar \
    --init \
    --truststore-path mcp-truststore.jks \
    --truststore-password changeit \
    --root-keystore-path root-ca-keystore.jks \
    --root-keystore-password changeit \
    --root-key-password changeit \
    --x500-name "C=DK, ST=Denmark, L=Copenhagen, O=MCP Test, OU=MCP Test, CN=MCP
Test Root Certificate, E=info@maritimeconnectivity.net" \
    --crl-endpoint
"https://api.example.com/x509/api/certificates/crl/urn:mrn:mcl:ca:maritimecloud"
```

You should now have two different keystore files called mcp-truststore.jks, which contains the root CA certificate, and root-ca-keystore.jks, which contains. The password for both of these is

‘changeit’, but can be set to something else by changing the values of truststore-password and root-keystore-password.

Now you will need to create a certificate revocation list (CRL) for the root CA. This can be done using the following command:

```
java -jar mc-pki-0.9.0-SNAPSHOT-jar-with-dependencies.jar \
    --generate-root-crl \
    --root-keystore-path root-ca-keystore.jks \
    --root-keystore-password changeit \
    --root-key-password changeit \
    --revoked-subca-file revoked-subca.csv \
    --root-crl-path root-ca.crl
```

The file **revoked-subca.csv** should either be empty or contain the sub CAs that have been revoked separated by commas and should have a format like this

```
<serial-number>;<revocation-reason>;<revocation-date>
```

The resulting file root-ca.crl contains the list of sub CAs that have been revoked.

The next step is then to generate the sub CA(s) that is going to be used for issuing client certificates. The command below shows an example of how to do this:

```
java -jar mc-pki-0.9.0-SNAPSHOT-jar-with-dependencies.jar \
    --create-subca \
    --root-keystore-path root-ca-keystore.jks \
    --root-keystore-password changeit \
    --root-key-password changeit \
    --truststore-path mc-truststore.jks \
    --truststore-password changeit \
    --subca-keystore subca-keystore.jks \
    --subca-keystore-password changeit \
    --subca-key-password changeit \
    --x500-name "UID=urn:mrn:mcl:ca:mcp-idreg, C=DK, ST=Denmark, L=Copenhagen,
O=MCP Test, OU=MCP Test, CN=MCP Test Identity Registry,
E=info@maritimeconnectivity.net" \
    --crl-endpoint
"https://api.example.com/x509/api/certificates/crl/urn:mrn:mcl:ca:mcp-idreg"
```

This creates a sub CA that contains the information given in x500-name and signed by the root CA. The sub CA certificate is added to mc-truststore.jks and the keys of the sub CA are stored in subca-keystore.jks. Note that in crl-endpoint you should change ‘api.example.com’ to whatever domain you are using and the MRN after ‘crl’ should be changed to whatever you wrote in ‘UID’.

All the files generated in this section should be stored for later usage.

Setting up NGINX

The next step is to set up NGINX. This is not required, but it is highly recommended to use NGINX as a reverse proxy because it makes handling of TLS and client certificate authentication a lot easier.
In the supplied file nginx.conf is an example of how NGINX can be used to proxy incoming requests to the Identity Registry API and Keycloak assuming that they are running on the same machine.
There are some requirements that need to be met to use the configuration.

The first one is that you need to have a domain registered with the sub-domains **api, api-x509, maritimeid, maritimeid-x509**. These can of course be changed to something else in the configuration is necessary.
The second requirement is that you need to have a TLS certificate for your domain. The path to the certificate should then be defined in the variable **ssl_certificate** and then the path to the corresponding private key should be defined in the variable **ssl_certificate_key**.
The third requirement is that the variable **ssl_crl** should point to an up-to-date file that contains the CRLs of the root CA and the sub CAs concatenated together. How to set this up will be described later in this document.
The fourth requirement is that the variable **ssl_client_certificate** must point to a file that contains the certificates of the root CA and the sub CAs in PEM format. These can be extracted from the mc-truststore.jks generated in the previous step using a tool like KeyStore Explorer.
NGINX can either be installed and run directly on the machine or in a Docker container. Note that if you choose to run it in a Docker container it is recommended to run it with the option --net=host so it binds directly to the network interface of the host instead of using the default bridge driver.

Setting up database

Prerequisite of installing both Keycloak and identity registry API is an installation of MariaDB. For Keycloak, it is important to set exactly same environment defined in the dockerfile of MCPKeycloakSPI (https://github.com/MaritimeConnectivityPlatform/MCPKeycloakSpi/blob/master/docker/Dockerfile). For example, if it contains ‘ENV DB_DATABASE keycloak’, you have to create a database ‘keycloak’ through *CREATE DATABASE keycloak*;. It is also required to set a dedicated user id in mysql.user and a password that perfectly corresponds to the values in the dockerfile. For the case using the prebuilt Docker container will be described later the database environment setup should be matched to the dockerfile of the repository of it.

Identity registry API provides bash files to setup the initial database, which are available on the repository (https://github.com/MaritimeConnectivityPlatform/IdentityRegistry/tree/master/setup). ‘setup-db.sh’ that should be executed first includes execution of a series of SQL statements in ‘create-database-and-user.sql’. After that it is able to create tables through ‘create-tables.sql’ and then a sample organization and role through ‘create-mc-org.sql’. For the case to remove the database and the user you can just execute ‘drop-db.sh’ or just ‘drop-db-and-user.sql’.

Setting up Keycloak

The next step is to setup Keycloak. This can be done using either a standalone installation of Keycloak or using the prebuilt Docker container. Using the Docker container is recommended, but if a standalone installation is required a guide on how to use this can be found at https://github.com/MaritimeConnectivityPlatform/MCPKeycloakSpi.

Using a Docker container

A prebuilt Docker container for Keycloak with MCP specific functionality can be found at https://cloud.docker.com/u/dmadk/repository/docker/dmadk/keycloak-mysql-mc-ha. At the time of writing the latest tag for this is **0.9.1**. The **latest** tag is built from the latest code from the main git branch and is therefore not guaranteed to be stable.

For creating the Docker container using the **0.9.1** tag you can use the following command:

```
docker create --name=mir-keycloak --restart=unless-stopped -p 8080:8080 \
-v <directory for configurations>:/mc-eventprovider-conf \
-e MC_IDREG_SERVER_ROOT=https://api-x509.example.com \
-e JGROUPS_DISCOVERY_EXTERNAL_IP=<valid IP address>
dmadk/keycloak-mysql-mc-ha:0.9.1
```

The directory that is mounted to '/mc-eventprovider-conf' in the container must contain the following files:

- mc-truststore.jks
- idbroker-updater.jks

The first one is the one that was generated earlier. The second one is only needed if user federation is used and will be described later on how to generate.

The URL given as value for the variable MC_IDREG_SERVER_ROOT must be set to the same value as the one set in the NGINX configuration.

If running Keycloak in clustered mode the value of JGROUPS_DISCOVERY_EXTERNAL_IP must be set to an IP address that Keycloak is reachable on. If it is not to be run in clustered mode it can just be set to 127.0.0.1 or another random IP.

As default the database and the user that is to be used for it are set to be **keycloak** and the password for the database user is set to be **password**. If you want to use other values than these you can set them as variables in the above command as described at https://hub.docker.com/r/jboss/keycloak.

To start Keycloak you can then use the command

```
docker run mir-keycloak
```

Setting up Realms in Keycloak

When you have gotten Keycloak up and running you will need to setup the needed Realms. To login to the admin interface of Keycloak go to http://localhost:8080 if you have Keycloak running on localhost or else use the domain set in your NGINX configuration. The first time you login you will need to setup an admin user and password.

After that you will need to import the three *-realm.json files from https://github.com/MaritimeConnectivityPlatform/IdentityRegistry/tree/master/setup.

This can be done in the admin interface of Keycloak by hovering the mouse over the dropdown 'Select realm' and then click 'Add realm'. Here you can then import the *-realm.json files one by one.

Note that the URLs to OIDC clients, identity providers, etc. are set to localhost in the *-realm.json files so these will need to be updated to the correct URLs after the files have been imported.

Setting up the Identity Registry API

The next step then is to set up the Identity Registry API (IDR API). There are two ways this can be done – either by setting it up directly on the system or by running it in a Docker container. Running in a Docker container is recommended, but both methods will be described in separate sections.

The source code for the Identity Registry API can be found at https://github.com/MaritimeConnectivityPlatform/IdentityRegistry.

Using a Docker container

A prebuilt Docker image for the Identity Registry API can be found at https://cloud.docker.com/u/dmadk/repository/docker/dmadk/mc-identity-registry-api. At the time of writing the latest tag for this is **0.9.1**. The **latest** tag is built from the latest code from the main git branch and is therefore not guaranteed to be stable.

For creating the Docker container using the **0.9.1** tag you can use the following command:

```
docker create --name=mir-api --restart=unless-stopped -p 8443:8443 \
-v <directory for configurations>:/conf dmadk/mc-identity-registry-api:0.9.1
```

The folder that has been mounted to 'conf' in the container needs to contain the following files:

- An application.yaml file
- A keycloak.json file
- mc-truststore.jks
- root-ca.crl
- subca-keystore.jks

How to make the two first files will be described later on. The three other files were the ones that were generated in the first step.

Directly on the system

For the standalone method you need to first download the newest version of the Identity Registry API from https://github.com/MaritimeConnectivityPlatform/IdentityRegistry. When you have done that you can compile the source code by going into the root directory of the project and issue the following command:

```
mvn clean install
```

After this has finished you will now have a new directory called 'target' that should contain a file called mc-identityregistry-core-latest.war. This can then be executed using the following command:

```
java -jar mc-identityregistry-core-latest.war \
--spring.config.location=<location for an application.yaml file> \
--keycloak.config.file=<location for a keycloak.json file>
```

Again how to make the two needed files will be described later.

Getting keycloak.json

For getting the keycloak.json file for IDR API you will need to login the admin interface of Keycloak. Here you should then choose the MaritimeCloud realm and click on 'Clients'. Here you then need to find and click on the client called 'mcidreg'. Here you should then click on the 'Installation' tab and in the dropdown that appears click 'Keycloak OIDC JSON' and then 'Download'. This file should be put in the configuration directory of the IDR API.

Making an application.yaml file

An example of an application.yaml file can be found at https://github.com/MaritimeConnectivityPlatform/IdentityRegistry/blob/master/src/main/resources/application.yaml.

It is important that this file is configured according to the correct URLs, database connections, name of the default sub CA, mail sending settings, etc.

When all of this has been configured the resulting file should then be put in the configuration directory of the IDR API.

Putting everything together

You can now also start up the IDR API. If you have set it up using the Docker container you can start it using the command

```
docker start mir-api
```

If you are not running it in a Docker container you can start it as described earlier.

If everything has been configured correctly all tables for the database should be generated automatically while the program starts.

After the program has finished starting up, indicated in the log by the entry 'Started McIdregApplication in X seconds (JVM running for Y)' , you now need to create an organization that is going to be used to bootstrap everything. To do this you need to insert the values from https://github.com/MaritimeConnectivityPlatform/IdentityRegistry/blob/master/setup/create-mc-org.sql into the database that is used for IDR API. This will create an organization called MaritimeCloud and a permission called MCADMIN for the organization that maps to the role ROLE_SITE_ADMIN which is a kind of 'superadmin' role that is able to administrate all entities in the IDR.

When this has been done we can move on to creating the organization that is going to be managing the MIR instance. A description on how to do this is given at https://github.com/MaritimeConnectivityPlatform/IdentityRegistry#insert-data. Note that this description assumes that both the IDR API and Keycloak are running on localhost so you might need to change the URL to fit your own setup. Also note that the organization that is going to be created is called 'Danish Maritime Authority' which you might also want to change. Finally the role mapping that is created from dma.json contains the role ROLE_ORG_ADMIN and the permission MCADMIN. This gives users from the DMA organization with the permission MCADMIN admin permissions for entities in the DMA organization. Like for the MaritimeCloud organization you might also want to create another role with a different name and the role ROLE_SITE_ADMIN for the organization if you want it to be able to have 'superadmin' permissions. Note that this cannot be created using the API and needs to be created directly in the database as was done earlier when creating the MaritimeCloud organization.

Finish setting up NGINX

Now that the IR API is up and running it is time to finish the setup of NGINX for it to be able to validate client certificates. This is done by setting up a CRON job that executes a script that gets the CRL for the root CA and for the sub CA(s) and combines them in a single file. If you want the CRON job to be run every 5 minutes and redirect the console output to a log file the CRON configuration could look like this:

```
*/5 * * * * update-crl.sh &>> update-crl.log
```

An example of this script can be seen in the supplied update-crl.sh.

In the NGNIX configuration you should then change the value of **ssl_crl** to the location of the file that contains the CRLs. After that you should then restart NGINX for the change to take effect.

Setting up synchronization between IR API and Keycloak

The next step now is to setup synchronization between the IR API Keycloak. This needs to be setup so that information about users, like names, permissions and so on, can be properly synchronized between the two components.

To do this you need to create a ‘device’ entity in the organization that you have setup to have the ‘superadmin’ permissions. This ‘device’ will then be used to facilitate the authentication of Keycloak against the IR API using a client certificate.

If you create the device directly using the IR API you can take inspiration from the given sync-device.json. The details of this device corresponds to the default configuration in the user-sync part of the IR API application.yaml. You can also consider setting up the MCP management portal so you don’t need to use the APIs directly. A manual on how to use this can be found here https://manual.maritimeconnectivity.net/.

If you are using the IR API directly you can reuse the authentication that you used previously to insert data. The device can then be created using the following command:

```
curl -k -H “Content-Type: application/json” -H “Authorization: Bearer $TOKEN” \
--data @sync-device.json
https://example.com/oidc/api/org/urn:mrn:mcl:org:dma/device
```

When you have then created this device you will need to issue a certificate for it. This can either be done using the management portal or again using the API directly. To do the latter the following command can be used:

```
curl -k -H “Authorization: Bearer $TOKEN”
https://example.com/oidc/api/org/urn:mrn:mcl:org:dma/device/urn:mrn:mcl:device:d
ma:sync/issue-new
```

The format of the returned JSON object is described at https://github.com/MaritimeConnectivityPlatform/IdentityRegistry#insert-data. The parts of the object that need to be used are the JKS keystore and the keystore password. The JKS keystore is BASE64 encoded so therefore it needs to be decoded and saved in a file called idbroker-updater.jks. This file should then be put in the directory that corresponds the /conf directory for the Keycloak Docker container or in the directory that corresponds to the ‘keystore-path’ variable if using a standalone installation of Keycloak. In the default configuration of Keycloak the password for the

keystore file is set to be ‘changeit’ so Keycloak to be able to open the keystore you might need to either change the password of it or set the ‘keystore-password’ variable to the correct password.