UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Master Primo Livello in Cybersecurity

DEMIoT:
a secure microservices IoT environment

Relatore:

Prof: Pagano Michele

Candidato:

Samuele Padula

ANNO ACCADEMICO 2023/2024

Contents

Chapter 1

Project Goals

This report will outline the fundamental goals and implementation strategies of a project that focuses on the creation of a lab in the Docker environment. This lab is aimed at implementing the Mosquitto MQTT broker and managing client authentication using digital certificates. A key element of this process is the integration of an Open Source Public Key Infrastructure (PKI), specifically the use of EJBCA to issue the necessary certificates. The primary objective of this initiative is to provide a reliable and secure laboratory environment for the development, testing and practical demonstration of the integration of Mosquitto MQTT as a messaging broker, combined with a rigorous client authentication process based on digital certificates. This implementation provides an in-depth exploration of MQTT's capabilities in an advanced security context, as well as offering a replicable model for real-world operational environments. Throughout this report, key steps for configuring the Docker environment, installing and configuring the Mosquitto MQTT broker, and implementing client authentication using digital certificates issued by the EJBCA Open Source PKI system will be detailed. In addition, the benefits and security implications of this architecture will be examined, along with recommended best practices for managing and maintaining a secure and functional MQTT environment. Through this report, we aim to provide comprehensive and detailed guidance on the deployment process, highlighting the inherent benefits of a Mosquitto MQTT-based secure IoT messaging solution and emphasizing the crucial importance of authentication via digital certificates issued by a trusted PKI such as EJBCA.

Chapter 2

Components and architecture

2.1 Components

2.1.1 Docker

The adoption of Docker as a virtualization environment has proven crucial for creating an isolated and highly replicable environment for the IoT lab. Its ability to efficiently containerize applications and their dependencies, as well as its ease of deployment and management, makes Docker the ideal choice for creating a consistent and easily replicable environment. In addition, Docker simplifies resource management and offers a high degree of flexibility in deploying, testing and deploying the entire stack of applications and services required for the IoT lab. Finally, the handling of name resolution already provided by the Docker environment and the isolation of different networks was one of the key reasons for its selection.

2.1.2 Mosquitto

Selection of Mosquitto by Eclipse Foundation as an MQTT broker is based on its proven reliability in handling the MQTT protocol, which is widely used for communication between IoT devices. Its light weight and scalability make it an optimal choice for handling large volumes of messages with a low impact on system resources. In addition, its wide adoption in the IoT community provides a mature support ecosystem and continuous development of new features and enhancements.

2.1.3 PKI EJBCA

The adoption of the EJBCA PKI by PrimeKey has proven to be critical for the reliable management of digital certificates required for client and MQTT broker authentication. The robustness and wide range of features offered by EJBCA enable the secure generation, management and revocation of certificates, ensuring a high level of security in the IoT environment. EJBCA's flexibility and adaptability align perfectly with the needs of an infrastructure that requires dynamic deployment and

secure management of digital certificates for device authentication. In addition, the ability to interact with its REST API interface enabled integration with other architectural components.

2.2 Architecture

The PKI plays a central role within the developed architecture as it allows all other application components to communicate in an authenticated and encrypted manner. In fact, the PKI exposes two ports, 8080/tcp and 8443/tcp, HTTP and HTTPS, respectively, for simplified management of users and certificates via the Web interface and for communication with the REST API interface. The MariaDB database has been brought back into the architecture as part of the solution deployment, but it is a "hidden" component in the eyes of us end users. The various MQTT clients as well as the Mosquitto broker interact with the EJBCA API via HTTP protocol to request the generation of a digital certificate in X.509 format. All application communications, on the other hand, between the MQTT clients and the Mosquitto broker take place using MQTT over TLS. The broker exposes only port 8883/tcp through which clients communicate with the server. Finally, to conduct connection tests with the broker, a custom Ubuntu-based image with some of the tools needed to communicate with the broker was chosen as the MQTT clients. User-defined Bridge networks in Docker were chosen. These are custom networks created by the user to connect Docker containers. These networks provide isolation and efficient communication between containers in the same Docker environment. They work by allowing containers in the same network to communicate with each other using container names as hostnames and allowing developers to define custom network configurations, such as IP address and subnet, for containers within the network. User-defined bridge networks provide a controlled and isolated environment for communication between containers, enabling better management of network resources and providing greater security and flexibility in the Docker environment. In order for two containers to be able to communicate, they must belong to the same bridge network. In particular, it was decided to allow only client communication to the PKI via the network `access-ejbca-net` and the broker `broker-net` and broker communication with the PKI and clients via the networks `publisher-net` and `subscriber-net`. Any communication between MQTT clients turns out to be segregated at the network level.

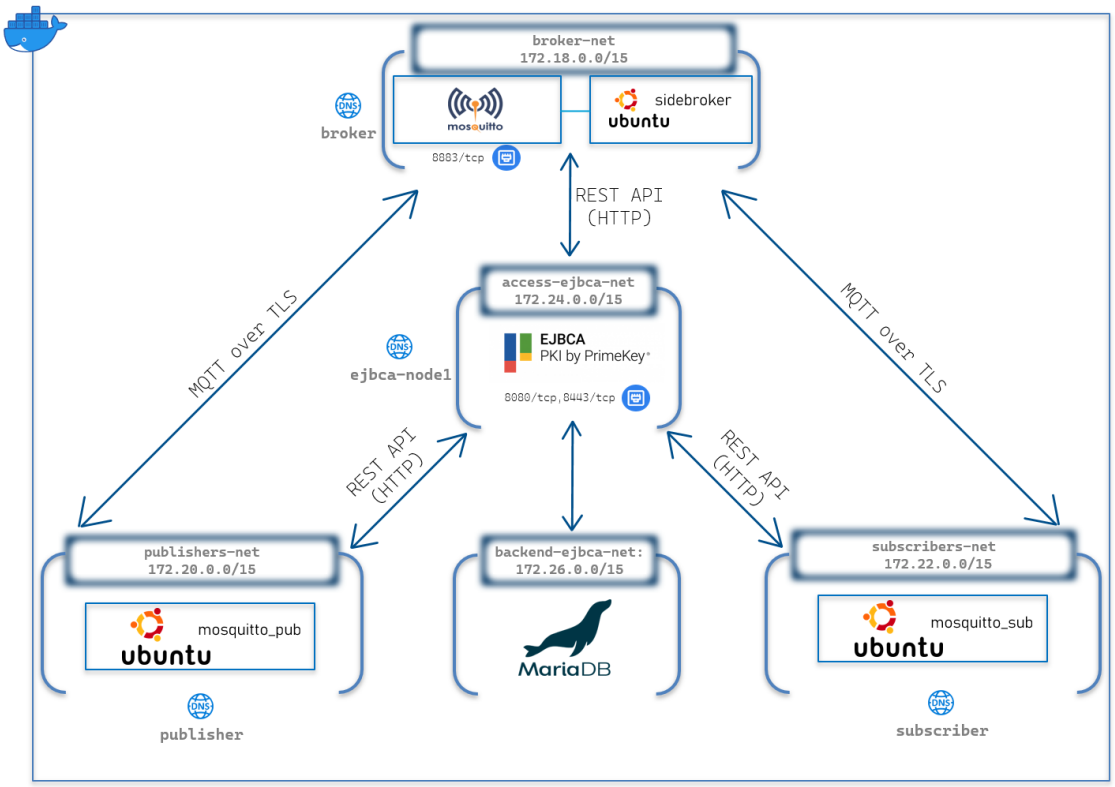


Figure 2.1: Docker architecture

Chapter 3

Deployment: structure and interactions

In this chapter we will address the various steps in order to properly install all the components of the presented solution. All the necessary code and configuration files can be found on the following GitHub repository https://github.com/padowla/DEMIoT.

3.1 Configuration of Docker environment

For the implementation of the project, Docker was chosen to be installed, specifically the installation was done on an Ubuntu 22.04.3 LTS machine, but by its nature, this lab is replicable on different platforms as well. In addition, an additional level of virtualization could be added by running Docker within a virtual machine running in a Type 1 or Type 2 Hypervisor. [3]. In addition, it was necessary to install the Docker Compose tool in order to more easily manage a multi-container environment. Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.[2] The docker compose YAML file used is as follows:

```
version: "3.9"
networks:
  broker-net:
    driver: bridge
    ipam:
      config:
        - subnet: 172.18.0.0/15
  publishers-net:
    driver: bridge
    ipam:
```

```
      config:
        - subnet: 172.20.0.0/15
  subscribers-net:
    driver: bridge
    ipam:
      config:
        - subnet: 172.22.0.0/15
  access-ejbca-net:
    driver: bridge
    ipam:
      config:
        - subnet: 172.24.0.0/15
  backend-ejbca-net:
    driver: bridge
    ipam:
      config:
        - subnet: 172.26.0.0/15
volumes: #here we define global volumes used by multiple services
  ca-certs: #the volume containing Certification Authority certificate file
    driver: local
services:
  ejbca-database:
    hostname: ejbca-database
    container_name: ejbca-database
    image: "library/mariadb:latest"
    networks:
      - backend-ejbca-net
    environment:
      - MYSQL_ROOT_PASSWORD=foo123
      - MYSQL_DATABASE=ejbca
      - MYSQL_USER=ejbca
      - MYSQL_PASSWORD=ejbca
    volumes:
      - ./ejbca/datadbdir:/var/lib/mysql:rw
  ejbca-node1:
    hostname: ejbca-node1
    container_name: ejbca
    image: keyfactor/ejbca-ce:latest
    depends_on:
      - ejbca-database
    networks:
      - access-ejbca-net
      - backend-ejbca-net
    environment:
      - DATABASE_JDBC_URL=jdbc:mariadb://ejbca-database
```

```
   :3306/ejbca?characterEncoding=UTF-8
      - LOG_LEVEL_APP=INFO
      - LOG_LEVEL_SERVER=INFO
      - TLS_SETUP_ENABLED=simple
    ports:
      - "80:8080"
      - "443:8443"
  sidebroker-service:
    build:
      context: .
      dockerfile: Dockerfile-sidebroker
    container_name: sidebroker
    hostname: sidebroker
    tty: true
    volumes:
      - ./mosquitto/certs/:/mosquitto/certs/:rw
      - ./mosquitto/keys/:/mosquitto/keys/:rw
    networks:
      - access-ejbca-net
  broker-service:
    build:
      context: .
      dockerfile: Dockerfile-mosquitto
    container_name: broker
    depends_on:
      sidebroker-service:
        condition: service_completed_successfully
    hostname: broker
    tty: true
    ports:
      - "8883:8883"
    volumes:
      - ./mosquitto/:/mosquitto/:rw
      - ca-certs:/mosquitto/ca-certs/:rw
    networks:
      - broker-net
      - subscribers-net
      - publishers-net
      - access-ejbca-net
  publisher-service:
    build:
      context: .
      dockerfile: Dockerfile-publisher
    container_name: publisher
    hostname: publisher
    tty: true
    volumes:
```

```
102       - ./publisher/:/publisher/:rw
103       - ca-certs:/publisher/ca-certs/:rw
104     networks:
105       - publishers-net
106       - access-ejbca-net
107   subscriber-service:
108     build:
109       context: .
110       dockerfile: Dockerfile-subscriber
111     container_name: subscriber
112     hostname: subscriber
113     tty: true
114     volumes:
115       - ./subscriber/:/subscriber:rw
116       - ca-certs:/subscriber/ca-certs/:rw
117     networks:
118       - subscribers-net
119       - access-ejbca-net
```

Listing 3.1: compose.yaml

To create the environment in Docker and run the applications specified in docker compose file, it is only necessary to run the command `docker compose up -d`.

3.2 Configuration of PKI EJBCA

In order to access the platform using the web console, the following URL must be copied into the search bar of the browser: http://localhost/ejbca/.

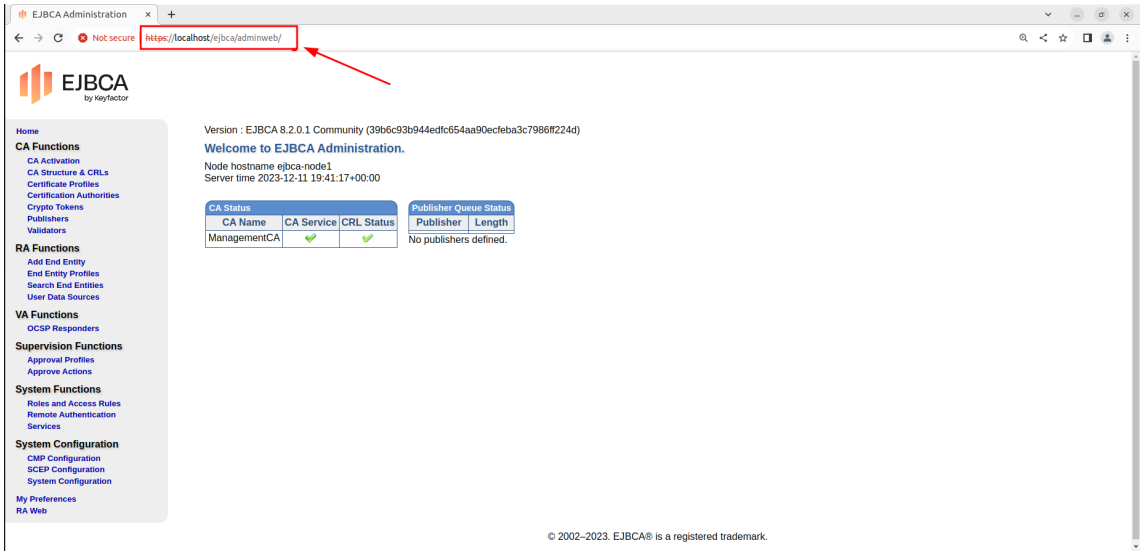
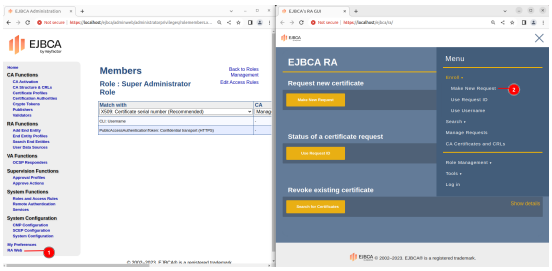
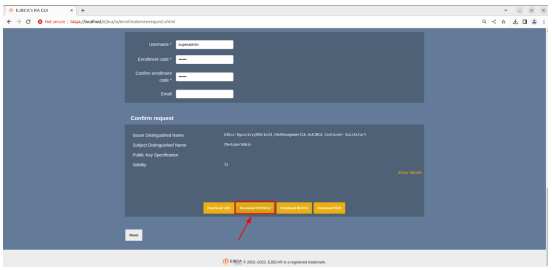


Figure 3.1: EJBCA Web Homepage

In order to be able to issue digital certificates, it is necessary to create a user in EJBCA that has permissions to generate new digital certificates. EJBCA offers the possibility of creating users with restricted roles while respecting the principle of least privilege. In this example, however, a user with an Administrator role was created as indicated in the documentation. This user has been associated with a certificate and a private key. The private key and certificate bundle are downloadable in a .p12 file that will be used later to make authenticated requests to the EJBCA REST API.[4].

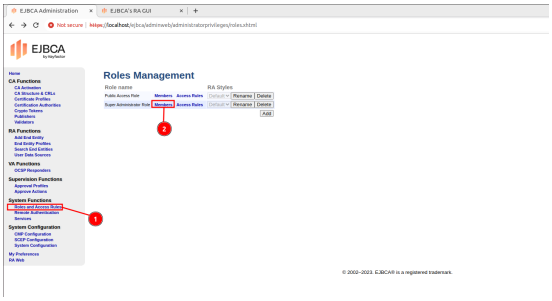


(a) Generate certificate with WEB UI

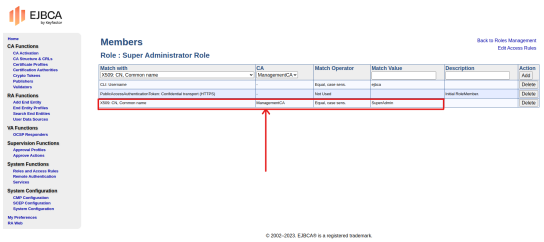


(b) Generate certificate with WEB UI

It is necessary to add the administrator user to the Super Administrator Role via the WEB interface by going to System Functions > Role and Access Rules > SuperAdministrator Role > Members:



(a) Add role to a user



(b) Add role to a user

Now we can copy the .p12 file we just downloaded inside the publisher, subscriber, and sidebroker folders.

In EJBCA we can generate our own Certification Authority hierarchy by configuring every last detail [5]. Since the purpose of this project is not to analyze this product but to offer a starting point for integration between the various software components used, ManagementCA for certificate generation will be used below. The ManagementCA in EJBCA is essentially a high-level CA responsible for overseeing and controlling the other CAs within the infrastructure, ensuring proper hierarchy, security and governance of the PKI. A Certificate Profile is a set of rules and configurations that determine how a digital certificate will be structured and formed when it is issued. The default certificate profiles will be used, respectively the SERVER one for broker certificate generation and the ENDUSER one for client certificate generation (as having the Extended Key Usage of type Client Authentication) [8].

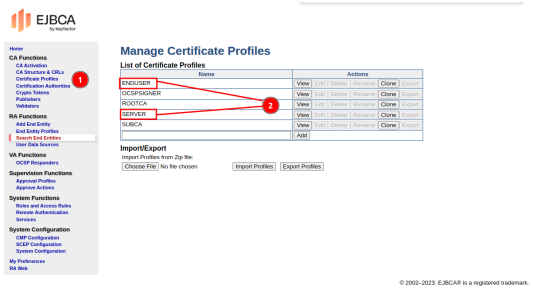


Figure 3.4: Default certificate profiles

Finally, it is necessary to download the file `ManagementCA.pem` from EJBCA WEB interface:

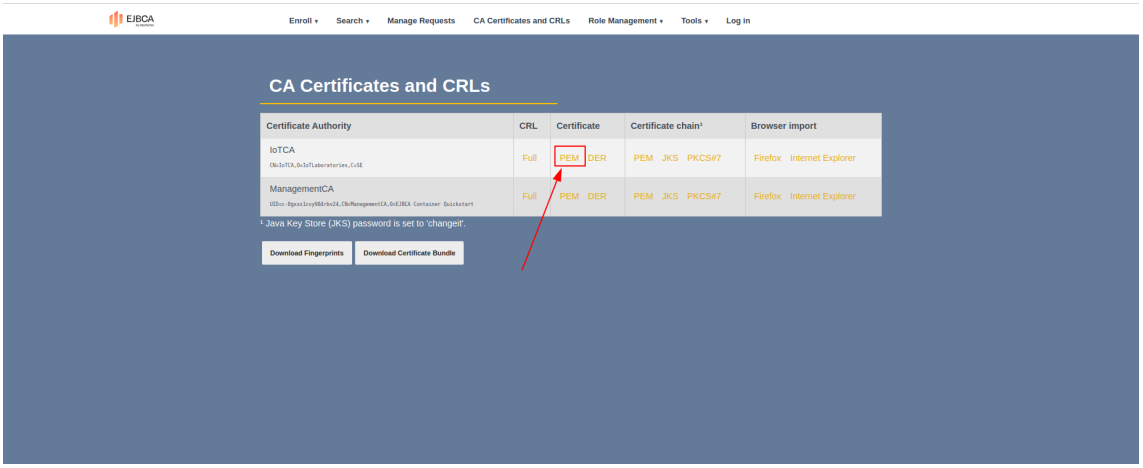


Figure 3.5: Download Certification Authority PEM file

and copy it to the root project folder named `ca-certs`.

Enabling the rest API

EJBCA Certificate Management REST API contains endpoints intended for integration with EJBCA, using CA Management, Crypto token management, Certificate management and ConfigDump RESTful Web Services. This allows for easy integration and lightweight HTTP interaction for the most crucial parts of EJBCA. The EJBCA Certificate Management REST API is disabled by default. To enable the service, go to CA UI → System Configuration → Protocol Configuration and select Enable for the REST end point you want to use. The REST API requires access to an active (non-external) CA [7].

Protocol	Resource Default URL	Status	Actions
ACME	/ejbca/acme	Enabled	Disable
Certstore	/ejbca/publicweb/certificates	Enabled	Disable
CMP	/ejbca/publicweb/cmp	Enabled	Disable
CRLstore	/ejbca/publicweb/crls	Enabled	Disable
EST	/.well-known/est	Enabled	Disable
MSAE	/ejbca/msae	Enabled	Disable
OCSP	/ejbca/publicweb/status/ocsp	Enabled	Disable
Public Web	/ejbca	Enabled	Disable
SCEP	/ejbca/publicweb/apply/scep	Enabled	Disable
RA Web	/ejbca/ra	Enabled	Disable
REST CA Management	/ejbca/ejbca-rest-api/v1/ca_management	Enabled	Disable
REST Certificate Management	/ejbca/ejbca-rest-api/v1/ca /ejbca/ejbca-rest-api/v1/certificate	Enabled	Disable
REST Crypto Token Management	/ejbca/ejbca-rest-api/v1/cryptotoken	Enabled	Disable
REST End Entity Management	/ejbca/ejbca-rest-api/v1/endentity	Enabled	Disable
REST End Entity Management V2	/ejbca/ejbca-rest-api/v2/endentity	Enabled	Disable
REST Configdump	/ejbca/ejbca-rest-api/v1/configdump	Disabled	Enable
REST Certificate Management V2	/ejbca/ejbca-rest-api/v2/certificate	Enabled	Disable
Webdist	/ejbca/publicweb/webdist	Enabled	Disable
Web Service	/ejbca/ejbcaws	Enabled	Disable
ITS Certificate Management	/ejbca/its	Disabled	Enable

Figure 3.6: Enable REST API EJBCA

3.3 Integration between EJBCA and application containers

3.3.1 sidebroker

The Docker principle that emphasizes the concept of having each application perform only the minimum necessary and required for its execution is often referred to as the "single responsibility principle" or "single responsibility principle." This principle is promoted through the practice of "single responsibility containerization" or "lightweight containers." This approach encourages the separation of functionality into separate containers, each of which performs a single responsibility or task. This promotes modularity, scalability and maintainability of the system as a whole. That is why it was decided to create a sidebroker container that would perform the tasks of generating a certificate-key pair for the broker. This was also made possible by the API exposed by EJBCA for requesting new certificates. In particular, the script `req_crt.sh`:

```
1 #!/bin/sh
2
3 CERT_FILE="broker.crt" # Define the path client certificate
4 CSR_FILE="broker.csr" # Define the path client Certificate
    Signing Request
5 PRIVATE_KEY="broker.key" # Define the path client private key
6 EJBCA_P12_AUTH_FILE="SuperAdmin.p12" # Define the path to P12
    authentication file EJBCA
7 EJBCA_PASSWORD_AUTH_FILE="foo123" # Define the password of P12
     authentication file EJBCA
8 EJBCA_HOST="ejbca-node1:8443" # Define the FQDN or IP address
    EJBCA
```

```
EJBCA_TRUST_CHAIN="ManagementCA.pem" # Define the path trust chain or single CA file EJBCA
EJBCA_USERNAME_END_ENTITY="superadmin" # Define the username of the entity created EJBCA
EJBCA_CERTIFICATE_PROFILE_NAME="SERVER" # Define the certificate profile name EJBCA
EJBCA_END_ENTITY_PROFILE_NAME="EMPTY" # Define the end entity profile name EJBCA
EJBCA_CA_NAME="ManagementCA" # Define the CA name EJBCA
RETRY_CHECK_EJBCA=5 # Define the time to wait before retry the EJBCA REST API availability
SERVICE_URL="http://ejbca-node1:8080/ejbca/publicweb/healthcheck/ejbcahealth" # Define the URL and port of the EJBCA REST API service



# Define a function that request a X.509 certificate
request_x509_certificate() {
  echo "[-] Certificate file not found, publisher requesting certificate..."
  # Create a CSR
  openssl req -new -out $CSR_FILE -newkey rsa:2048 -nodes -sha256 -keyout $PRIVATE_KEY -subj "/CN=broker"

  # Make the script runnable
  chmod a+x pkcs10Enroll.sh

  # Request the certificate
  ./pkcs10Enroll.sh \
      -c "$CSR_FILE" \
      -P "$EJBCA_P12_AUTH_FILE" \
      -s "$EJBCA_PASSWORD_AUTH_FILE" \
      -H "$EJBCA_HOST" \
      -t "$EJBCA_TRUST_CHAIN" \
      -u "$EJBCA_USERNAME_END_ENTITY" \
      -p "$EJBCA_CERTIFICATE_PROFILE_NAME" \
      -e "$EJBCA_END_ENTITY_PROFILE_NAME" \
      -n "$EJBCA_CA_NAME"


  # Rename the certificate
  mv "$EJBCA_USERNAME_END_ENTITY.crt" "$CERT_FILE"

  # Check the exit status of openssl
  if [ $? -eq 0 ]; then
    echo "Certificate requested done correctly!"
      # Copy the generated certificate and key inside the broker container volume
      cp $CERT_FILE /mosquitto/certs
        cp $PRIVATE_KEY /mosquitto/keys
        exit 0
```

```
51   else
52       echo "Error during certificate request! Check logs"
53       tail -f /dev/null #in case of error do not terminate to
     run
54   fi
55 }
56
57
58
59 until $(curl --output /dev/null --silent --head --fail
     $SERVICE_URL); do
60     echo "Waiting for the service to be available..."
61     sleep $RETRY_CHECK_EJBCA
62 done
63
64 echo "[+] EJBCA available"
65
66
67
68 # Check if the certificate file exists
69 if [ ! -f "$CERT_FILE" ]; then
70   request_x509_certificate
71 else
72
73   # Get the expiration date of the certificate
74   expiration_date=$(openssl x509 -in "$CERT_FILE" -noout -
     enddate | cut -d= -f2)
75
76   # Convert the expiration date to Unix timestamp
77   expiration_timestamp=$(date -d "$expiration_date" +%s)
78
79   # Get the current Unix timestamp
80   current_timestamp=$(date +%s)
81
82   # Compare expiration date with current date
83   if [ "$expiration_timestamp" -ge "$current_timestamp" ];
     then
84       echo "Certificate is valid. Expiration date:
     $expiration_date"
85       exit 0
86   else
87       echo "Certificate has expired. Expiration date:
     $expiration_date"
88       request_x509_certificate # Request a new certificate
89   fi
90 fi
```

Listing 3.2: req_crt.sh

is responsible for waiting for the PKI to be up and running and then checking whether or not a digital certificate exists [6]. If not, it will be requested by calling

the `pkcs10Enroll.sh` script with the correct parameters:

```
#!/bin/bash

INPUT_HOSTNAME=""
INPUT_CERT_PROFILE=""
INPUT_END_ENTITY_PROFILE=""
INPUT_CA_NAME=""
INPUT_USERNAME=""
enrollment_code=""

help () {
  echo "Usage: `basename $0` options"
  echo "-c : the csr file"
  echo "-P : the p12 file to authenticate with"
  echo "-s : the p12 file password"
  echo "-t : the Trust chain file for the TLS certificate"
  echo "-H : the EJBCA FQDN or IP Address"
  echo "-u : the username of the entity created in EJBCA"
  echo "-p : the certificate profile name"
  echo "-e : the end entity profile name"
  echo "-n : the CA name"
  echo "
This script will use the EJBCA REST API PKCS10Enroll endpoint
   to submit CSR's for a certificate
"
}

while getopts "c:P:H:s:t:u:p:e:n:xh" optname ; do
  case $optname in
    c )
      INPUT_CSR_FILE=$OPTARG ;;
    P )
      INPUT_P12_CREDENTIAL=$OPTARG ;;
    s )
      INPUT_P12_CREDENTIAL_PASSWD=$OPTARG ;;
    t )
      INPUT_TRUST_CHAIN=$OPTARG ;;
    H )
      INPUT_HOSTNAME=$OPTARG ;;
    u )
      INPUT_USERNAME=$OPTARG ;;
    p )
      INPUT_CERT_PROFILE=$OPTARG ;;
    e )
      INPUT_END_ENTITY_PROFILE=$OPTARG ;;
    n )
      INPUT_CA_NAME=$OPTARG ;;
    x )
      set -x ;;
    h )
```

```
      help ; exit 0 ;;
    ? )
      echo "Unknown option $OPTARG." ; help ; exit 1 ;;
    : )
      echo "No argument value for option $OPTARG." ; help ;
   exit 1 ;;
    * )
      echo "Unknown error while processing options." ;;
  esac
done

if [ ! -f "$INPUT_CSR_FILE" ]; then
    echo "Please try again with a csr"
    exit 1
fi

if [ ! -f "$INPUT_P12_CREDENTIAL" ]; then
    echo "Please specify a P12 file"
    exit 1
fi

csr="$(cat ${INPUT_CSR_FILE})"

template='{"certificate_request":$csr, "
   certificate_profile_name":$cp, "end_entity_profile_name":
   $eep, "certificate_authority_name":$ca, "username":$ee, "
   password":$pwd}'
json_payload=$(jq -n \
    --arg csr "$csr" \
    --arg cp "$INPUT_CERT_PROFILE" \
    --arg eep "$INPUT_END_ENTITY_PROFILE" \
    --arg ca "$INPUT_CA_NAME" \
    --arg ee "$INPUT_USERNAME" \
    --arg pwd "$enrollment_code" \
    "$template")
echo $json_payload
if [ -f "$INPUT_TRUST_CHAIN" ]; then
    curl -vv -k -X POST -s --cacert "$INPUT_TRUST_CHAIN" \
    --cert-type P12 \
    --cert "$INPUT_P12_CREDENTIAL:$INPUT_P12_CREDENTIAL_PASSWD
   " \
    -H 'Content-Type: application/json' \
    --data "$json_payload" \
    "https://${INPUT_HOSTNAME}/ejbca/ejbca-rest-api/v1/
   certificate/pkcs10enroll" \
    | jq -r .certificate | base64 -d > "${INPUT_USERNAME}-der.
   crt"
else
    curl -vv -k -X POST -s \
    --cert-type P12 \
    --cert "$INPUT_P12_CREDENTIAL:$INPUT_P12_CREDENTIAL_PASSWD
   " \
```

```
93      -H 'Content-Type: application/json' \
94      --data "$json_payload" \
95      "https://${INPUT_HOSTNAME}/ejbca/ejbca-rest-api/v1/
        certificate/pkcs10enroll" \
96      | jq -r .certificate | base64 -d > "${INPUT_USERNAME}-der.
        crt"
97  fi
98
99  openssl x509 -inform DER -in "${INPUT_USERNAME}-der.crt" -
        outform PEM -out "${INPUT_USERNAME}.crt"
```

Listing 3.3: pkcs10Enroll.sh

Finally, sidebroker container creation is handled through the Dockerfile-sidebroker as shown below:

```
1  FROM ubuntu:latest
2
3  # This is necessary because the mounting volumes comes
       after the building image in compose.yaml
4  COPY ./sidebroker/ /sidebroker
5
6  WORKDIR /sidebroker
7
8  # Download and install all necessary packages
9  RUN apt-get update && apt-get install -y jq openssl curl
       iputils-ping curl
10
11 RUN chmod a+x req_crt.sh
12
13 RUN chown root:root /sidebroker/req_crt.sh
14
15 ENTRYPOINT ["/bin/bash", "/sidebroker/req_crt.sh"]
```

Listing 3.4: Dockerfile-sidebroker

3.3.2 publisher and subscriber

The client images used for the purpose of application testing were deployed from the image base provided by Canonical [1]. Here are the Dockerfiles of the publisher and subscriber containers:

```
1  FROM ubuntu:latest
2
3  RUN apt-get update && apt-get install -y jq openssl curl
       iputils-ping curl mosquitto-clients
```

```
4
5  COPY ./publisher/ /publisher/
6
7  COPY ./ca-certs/ /publisher/ca-certs/
8
9  # Create a directory to store custom certificates
10 RUN mkdir -p /usr/local/share/ca-certificates
11
12 # Copy the custom certificate to the container
13 RUN cp /publisher/ca-certs/* /usr/local/share/ca-
      certificates/
14
15 RUN apt-get update && apt-get install -y ca-certificates
      && update-ca-certificates
16
17 WORKDIR /publisher
18
19 RUN chmod +x req_crt.sh
20
21 RUN chown root:root /publisher/req_crt.sh
22
23 ENTRYPOINT ["/bin/bash", "/publisher/req_crt.sh"]
```

Listing 3.5: Dockerfile-publisher

```
1  FROM ubuntu:latest
2
3  RUN apt-get update && apt-get install -y jq openssl curl
      iputils-ping curl mosquitto-clients
4
5  COPY ./subscriber/ /subscriber/
6
7  COPY ./ca-certs/ /subscriber/ca-certs/
8
9  # Create a directory to store custom certificates
10 RUN mkdir -p /usr/local/share/ca-certificates
11
12 # Copy the custom certificate to the container
13 RUN cp /subscriber/ca-certs/* /usr/local/share/ca-
      certificates/
14
15 RUN apt-get update && apt-get install -y ca-certificates
      && update-ca-certificates
16
17 WORKDIR /subscriber
18
```

```
RUN chmod +x req_crt.sh

RUN chown root:root /subscriber/req_crt.sh

ENTRYPOINT ["/bin/bash", "/subscriber/req_crt.sh"]
```

Listing 3.6: Dockerfile-subscriber

The creation of certificates for containers simulating MQTT clients (publishers and subscribers) is done using the same scripts seen earlier for the sidebroker making the necessary changes regarding the parameters and paths of the corresponding volumes. Two bash scripts, `subscribe.sh` and `publish.sh`, were also developed in order to speed up the distribution of MQTTS messages between publishers and subscribers:

```
#!/bin/bash

# Configuration
BROKER_ADDRESS="broker"
BROKER_PORT="8883"
CA_CERT="ca-certs/ManagementCA.pem"
CLIENT_CERT="publisher.crt"
CLIENT_KEY="publisher.key"
TOPIC="LAB"

# Infinite loop for publishing messages with timestamp every 10 seconds
while true; do
    TIMESTAMP=$(date +%s)  # Get current Unix timestamp
    MESSAGE="There is a new message! $(date)"  # Append formatted timestamp to your_message

    mosquitto_pub -h "$BROKER_ADDRESS" -p "$BROKER_PORT" --cafile "$CA_CERT" --cert "$CLIENT_CERT" --key "$CLIENT_KEY" -t "$TOPIC" -m "$MESSAGE" -d --insecure
    sleep 10  # Wait for 10 seconds before publishing again
done
```

Listing 3.7: publish.sh

```
# Configuration
BROKER_ADDRESS="broker"
BROKER_PORT="8883"
CA_CERT="ca-certs/ManagementCA.pem"
CLIENT_CERT="subscriber.crt"
CLIENT_KEY="subscriber.key"
TOPIC="LAB"
```

```
8
9
10 # Infinite loop for subscribing and receiving messages every
       10 seconds
11 while true; do
12     mosquitto_sub -h "$BROKER_ADDRESS" -p "$BROKER_PORT" --
       cafile "$CA_CERT" --cert "$CLIENT_CERT" --key "$CLIENT_KEY"
        -t "$TOPIC" -d
13     sleep 10  # Wait for 10 seconds before subscribing again
14 done
```

Listing 3.8: subscribe.sh

3.4 Configuration of Mosquitto broker

Mosquitto broker is deployed from the Eclipse Foundation base image [9]. A copy of ManagementCA's certificate is added in order to trust any certificate issued by ManagementCA and presented by clients during authentication:

```
1 FROM eclipse-mosquitto:latest
2
3 COPY ./mosquitto/ /mosquitto/
4
5 COPY ./ca-certs/ /mosquitto/ca-certs/
6
7 #to trust the self-signed certificate CA
8 RUN cp /mosquitto/ca-certs/* /usr/local/share/ca-
      certificates/
9
10 #update the trust store certificates
11 RUN update-ca-certificates
```

Listing 3.9: Dockerfile-mosquitto

Mosquitto broker at startup loads the certificate to be exposed in MQTTS communications, the private key and the certificate of the trusted CA for client authentication from the paths specified in the configuration file. The `require_certificate true` setting in the Mosquitto configuration file requires clients to present a valid certificate to authenticate to the MQTT server. Below is the `mosquitto.conf` configuration file:

```
1 listener 8883 0.0.0.0
2 protocol mqtt
3 certfile /mosquitto/certs/broker.crt
```

```
keyfile /mosquitto/keys/broker.key
tls_version tlsv1.3
require_certificate true
cafile /mosquitto/ca-certs/ManagementCA.pem
log_type all
allow_anonymous true
```

Listing 3.10: mosquitto.conf

Chapter 4

Proof of Concept

In this final chapter, significant moments and key procedures will be shown through detailed screenshots. The screenshots presented here serve as a visual guide to understanding the operation, interactions, and crucial communication steps between the various components of the implemented infrastructure. Via the command `docker compose logs -f` launched within the secureiot parent folder, it is possible to analyze the logs of the entire deployment process performed with the compose docker. In particular we note that both the sidebroker container and the MQTT clients check for the existence and validity of a previously issued digital certificate that is already present within the file system; if not, they request it via the scripts seen earlier. Here is an example for the sidebroker container:

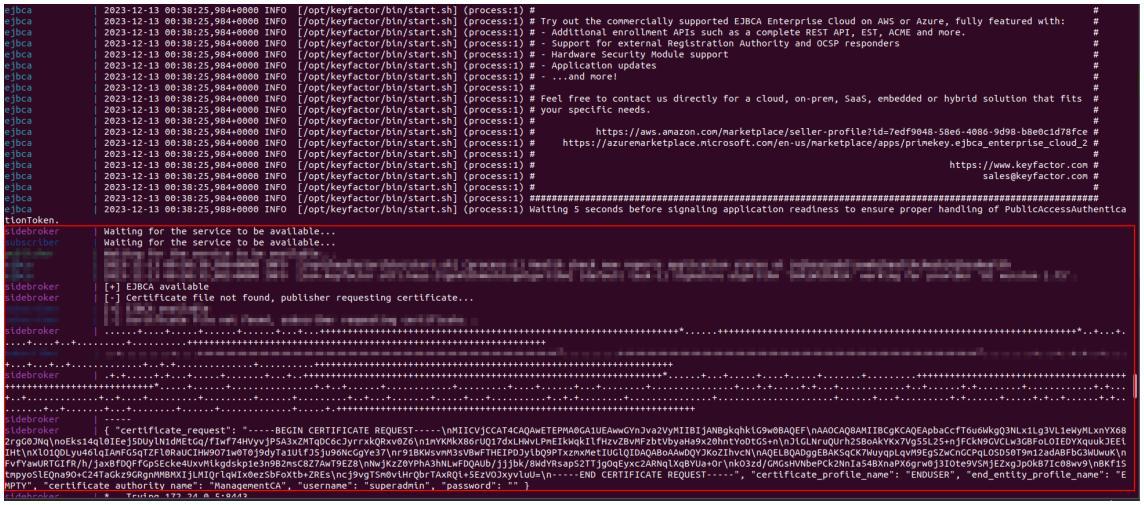
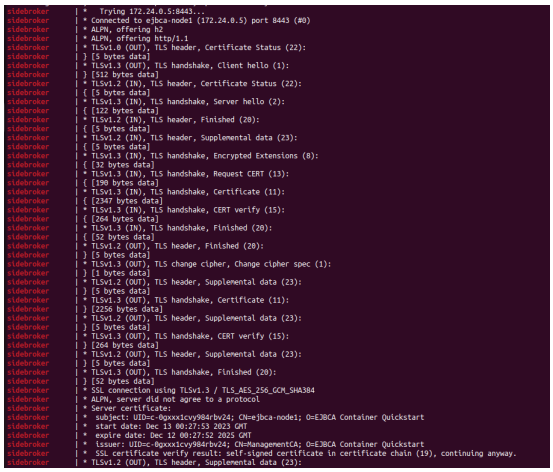
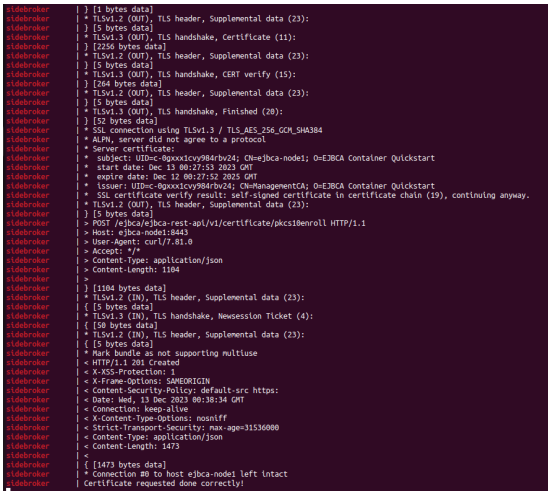


Figure 4.1: sidebroker requesting a certificate to EJBCA



(a) CURL command output



(b) CURL command output

After this phase of certificate request and generation, the Mosquitto broker, thanks to the specification in the `compose.yaml` of the `depends_on`, waits for the service sidebroker to be successfully completed so that it can be started with the certificates and keys needed for encrypted and authenticated communications:

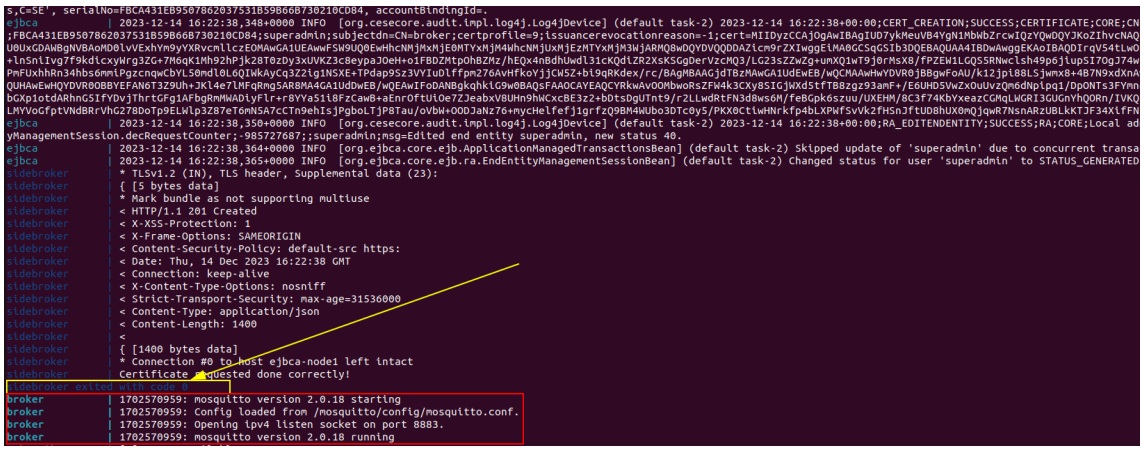


Figure 4.3: Mosquitto wait sidebroker to complete succesfully

We enter inside the application containers via the commands `docker exec -it subscriber /bin/bash` and `docker exec -it publisher /bin/bash` and run the scripts to send/receive the MQTT messages seen earlier. By launching the two scripts seen above, `subscribe.sh` and `publish.sh` respectively, we notice that connections with the broker are established without any error and messages are delivered correctly:

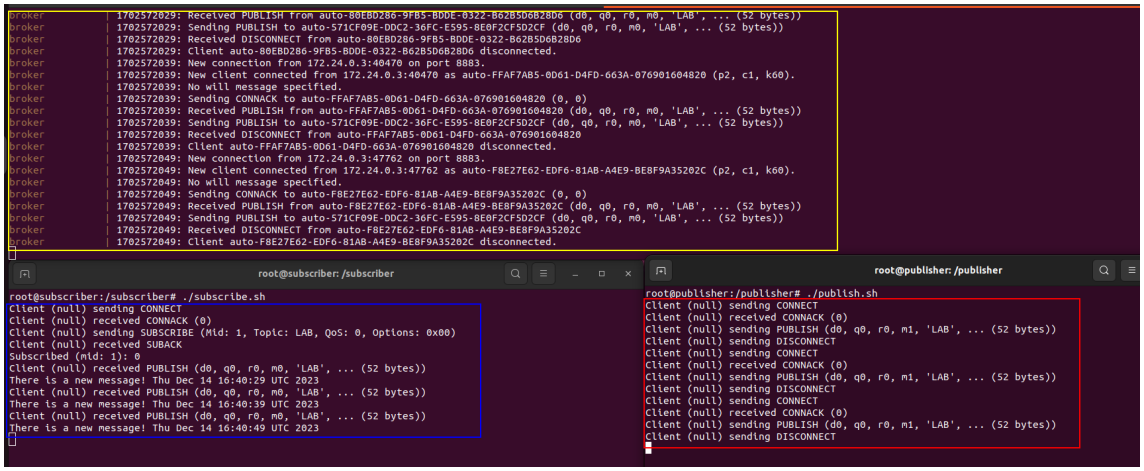


Figure 4.4: MQTTS Message Exchange

in red we have the messages sent by the publisher, in blue those received by the subscriber, and in yellow the logs of correct dispatching of messages by the broker.

Conclusion

The following project allowed the development of an advanced and comprehensive laboratory focused on the integration of several open-source components. This lab was designed to simulate a real IoT environment, with particular attention to the management of digital certificates and in reference especially to the use of MQTT and TLS protocols in order to ensure high security in the IoT environment, a mode now pervasive in our daily reality. I would like to thank Prof. Michele Pagano and Prof. Rosario Giuseppe Garroppo for their guidance and constant support during the implementation of this project. They played a key role in guiding the project, providing valuable advice, stimuli and insights that greatly enriched the learning journey and the realization of the lab.

Bibliography

[1] Canonical. Ubuntu Image on Docker Hub. https://hub.docker.com/_/ubuntu, 2023. [Online; accessed 11-December-2023].

[2] Docker Documentation. Install Docker Compose. https://docs.docker.com/compose/install/, 2023. [Online; accessed 11-December-2023].

[3] Docker Documentation. Install Docker Engine. https://docs.docker.com/engine/install/, 2023. [Online; accessed 11-December-2023].

[4] EJBCA PKI Documentation. Create Administrator Account. https://doc.primekey.com/ejbca/tutorials-and-guides/tutorial-start-out-with-ejbca-docker-container#TutorialStartoutwithEJBCADockercontainer-CreateAdministratorAccount, 2023. [Online; accessed 11-December-2023].

[5] EJBCA PKI Documentation. EJBCA Create a Root CA. https://doc.primekey.com/ejbca/tutorials-and-guides/tutorial-create-your-first-root-ca-using-ejbca, 2023. [Online; accessed 11-December-2023].

[6] EJBCA PKI Documentation. EJBCA Healtcheck. https://doc.primekey.com/ejbca/ejbca-operations/ejbca-operations-guide/ca-operations-guide/ejbca-maintenance/monitoring-and-healthcheck, 2023. [Online; accessed 11-December-2023].

[7] EJBCA PKI Documentation. EJBCA REST Interface. https://doc.primekey.com/ejbca/ejbca-operations/ejbca-ca-concept-guide/protocols/ejbca-rest-interface#:~:text=The%20EJBCA%20Certificate%20Management%20REST,(non%2Dexternal)%20CA., 2023. [Online; accessed 11-December-2023].

[8] EJBCA PKI Documentation. Start out with EJBCA Docker container. https://doc.primekey.com/ejbca/tutorials-and-guides/tutorial-start-out-with-ejbca-docker-container, 2023. [Online; accessed 11-December-2023].

[9] Eclipse Foundation. Mosquitto Image on Docker Hub. https://hub.docker.com/_/eclipse-mosquitto, 2023. [Online; accessed 11-December-2023].