SAPIENZA
UNIVERSITÀ DI ROMA

Internship Report: MQTT over TLS Security Assessment

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di laurea triennale in Informatica erogato in modalità Teledidattica

Radek Patrick Di Luca
ID number 1803854

Responsabile
Prof. Angelo Spognardi

Academic Year 2023/2024

Internship Report: MQTT over TLS Security Assessment
Relazione di Tirocinio. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author’s email: diluca.1803854@studenti.uniroma1.it

Contents

Chapter 1

Problem Definition

The aim of this internship work was to assess the security of the TLS Protocol implementation of some of the main MQTT Broker Libraries that can be found in the IT community. Some faults at the Application layer (MQTT) of some of these libraries were found by my colleague Edoardo Di Paolo during his internship work [4], so the hypothesis was that these libraries might very well have some faults at the Transport layer (TLS) too. Therefore, through the generation of some fabricated TLS Certificates and the definition of a Suite of Automated Unit Tests, the goal was to expose vulnerabilities in these libraries, or validate their implementation as secure.

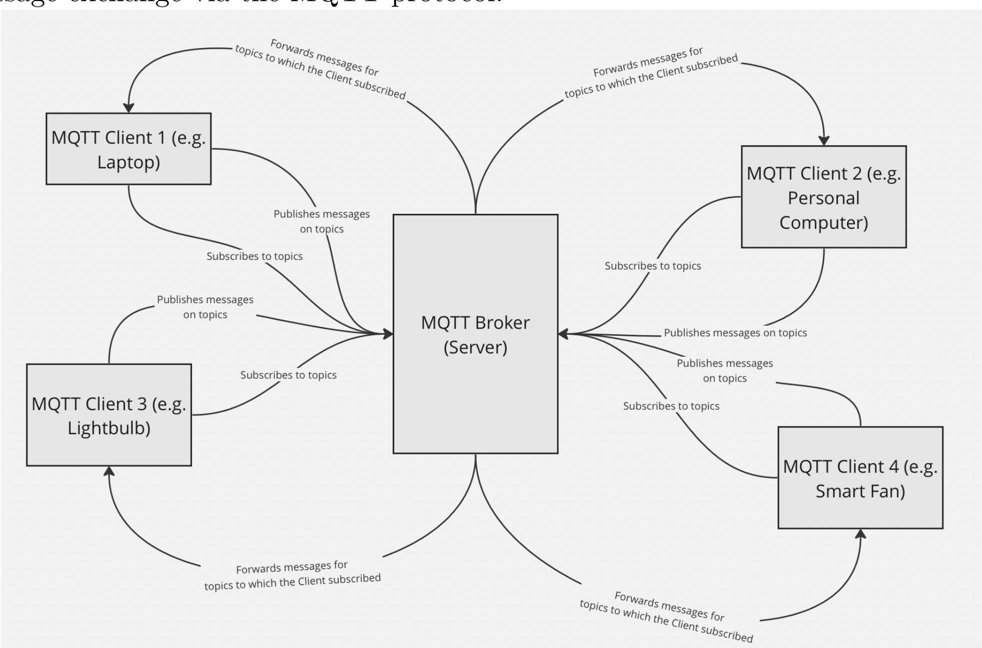
Chapter 2

Key Concepts

For the sake of this report, we will be using some core concepts that are critical to understanding the internship work.

2.0.1 MQTT

MQTT, also known as Message Queuing Telemetry Transport, is a lightweight protocol used on the Application Layer of the TCP/IP stack. MQTT is an alternative to the widely spread HTTP, and it's mainly used for connectivity to and from Internet of Things devices, due to the lightweight nature of the protocol and due to the low memory availability of the above mentioned IoT devices. Since the MQTT protocol is by nature a lightweight protocol, it does not feature many security capabilities, so it must rely on the security checks made by the layer immediately below MQTT, the Transport layer, via SSL/TLS. We can see here a representation of a typical message exchange via the MQTT protocol:



The exchange of information is mainly done through the publish/subscribe paradigm:

- **subscribe**: an MQTT Client subscribes to one or more topics. Each topic is identified by a unique string and after subscribing to the topic(s), the MQTT Client will be in a 'listening' state, receiving any new messages that will be published on the topic(s).
- **publish**: an MQTT Client publishes a message to a topic. Each topic is identified by a unique string, and by publishing the message, any Client that was subscribed to the topic will receive the published message.

2.0.2 SSL/TLS

SSL, also known as Secure Sockets Layer, is a protocol used on the Transport Layer of the TCP/IP stack, to provide security in the form of confidentiality, integrity and authenticity to one or both parties involved in the message exchange. In fact, SSL consists mainly of a Handshake phase, in which the client and server negotiate the parameters that will be used to establish the security of the following communication. During this Handshake phase, it is possible to negotiate whether the security is one-way (only the server is authenticated towards the client) or both ways (also known as mutual SSL, mutual TLS or abbreviated, mTLS).

2.0.3 Certificate Authority

A Certificate Authority, abbreviated CA, is a secure third party who is trusted by both TLS server and TLS client. In general, the client trusts the CA to certify that the server is who they claim to be. In mutual TLS, the CA is also used by the server, to certify that the client is who they claim to be.

Chapter 3

TLS Vulnerabilities

One of the main pieces of work done for this Internship was to define the ways in which an Attacker could possibly exploit a badly implemented TLS connection. Therefore, referencing the specifications RFC 2818 [1], RFC 8446 [3] and RFC 5280 [2], the following list of validations an MQTT Broker Library should implement was produced:

- **Chain of trust**: the certificate has to be either signed by a root Certificate Authority, or it has to have a linked list of references to various Certificate Authorities, up to a root node certificate self-signed by a root Certificate Authority. In this linked list, the certificate of every issuing Certificate Authority has to be signed by the Certificate Authority immediately above it.
- **Hostname**: the *Common Name* field should match the identifier of the entity to which we're connecting (server).
- **(Recursive) Expiration**: the *Not Valid Before* and *Not Valid After* fields contain information on the timestamps that delimit the interval in which the certificate should be accepted as valid. The Library should check that today's date is contained within that interval. This is valid also recursively for the Certificates of the issuing Certificate Authorities throughout the chain of trust.
- **Public key**: the *Public Key Info > Public Key* field should match the information provided by the Certificate Authority. This public key will be used to encrypt all the upcoming communication, so using the correct public key is essential.
- **Downgrade attack**: some attackers might try pretending that the latests versions of TLS are not supported by one of the two communicating parties, or they could also pretend that the set of supported ciphers is limited to only easily breakable ciphers, for example based on 512 bit-long cipher keys. This is why it's important to support only the versions of TLS that are still considered secure, and it's important to restrict the set of supported ciphers to only implementation-strong ciphers.

Chapter 4

Test Suite

To test the MQTT Broker Libraries, a Unit Test Suite was formally defined, with a series of descriptions and assertions made. The Unit Tests are defined following the Triangulation technique, which means that the Test Suite should assert both the valid scenarios in which the connection should be established and the illegal scenarios in which the connection should be rejected. Hence the Tests are defined as follows:

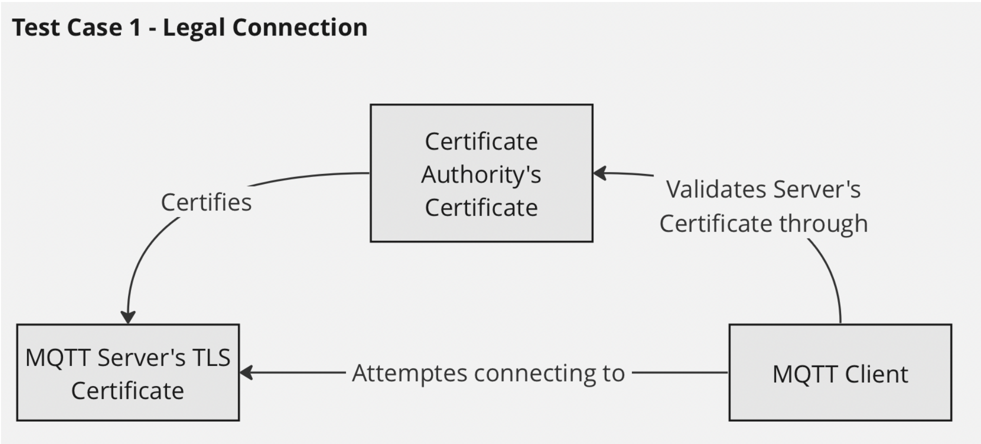
1. Test Case 1 - Legal Connection
2. Test Case 2 - Self Signed Attacker
3. Test Case 3 - Self Signed Attacker's Fake CA
4. Test Case 4 - Alteration 1 (Common Name)
5. Test Case 5 - Alteration 2 (Expiration Date)
6. Test Case 6 - Alteration 3 (Public Key)
7. Test Case 7 - Expired CA (lteration 4)
8. Test Case 8 - Certificate Extension
9. Test Case 9 - Longer Chain Of Trust Legal Connection
10. Test Case 10 - Altered Intermediate CA Common Name
11. Test Case 11 - Altered Intermediate CA Public Key

Note: The tested libraries are set up as MQTT Broker, or MQTT Server. The Client, which asserts the outcome of the test, always uses the Mosquitto command line tools to connect to the Server.

4.0.1 Test Case 1 - Legal Connection

This Test Case is set up by configuring the MQTT Broker Library with a valid TLS Certificate signed by the real Certificate Authority. The Tester Client connects to the server checking the Server TLS Certificate against the real Certificate Authority's Certificate. The table of the Unit Test is as follows:

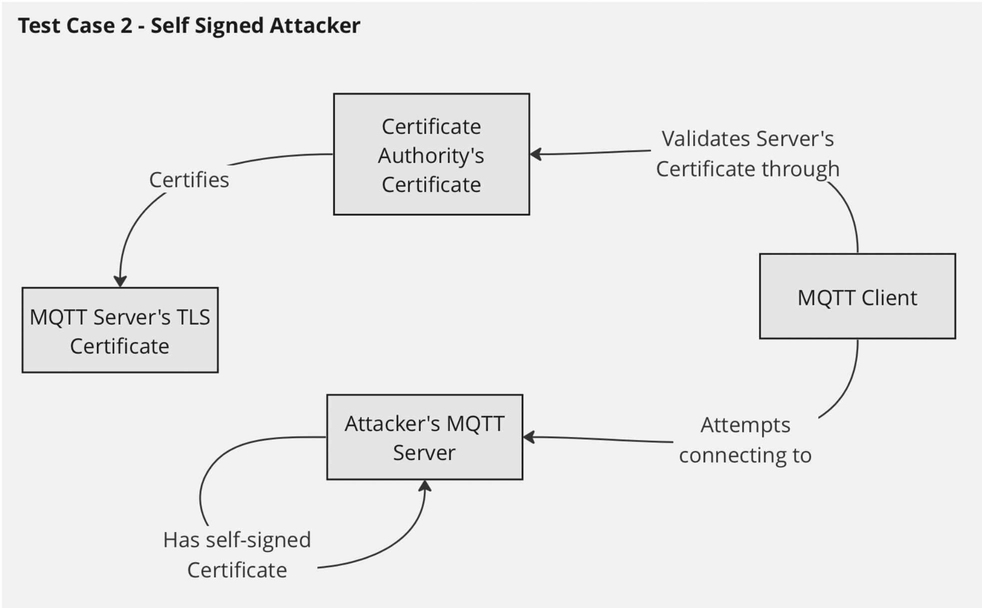
Intruder Access Capabilities	None
Intruder's Attack description	This test case represents the happy path with no intruder attack.
State of TLS Certificate	The TLS Certificate we use for this test is exactly the Server's Certificate.
State of Certificate's Signature	The signature is ***valid***
Assertion	The Library should ***accept*** the connection when a client tries connecting to the MQTT Library configured with this certificate.



4.0.2 Test Case 2 - Self Signed Attacker

This Test Case is set up by configuring the MQTT Broker Library with a forged self-signed TLS Certificate. The Tester Client connects to the server checking the Server TLS Certificate against the real Certificate Authority's Certificate. The table of the Unit Test is as follows:

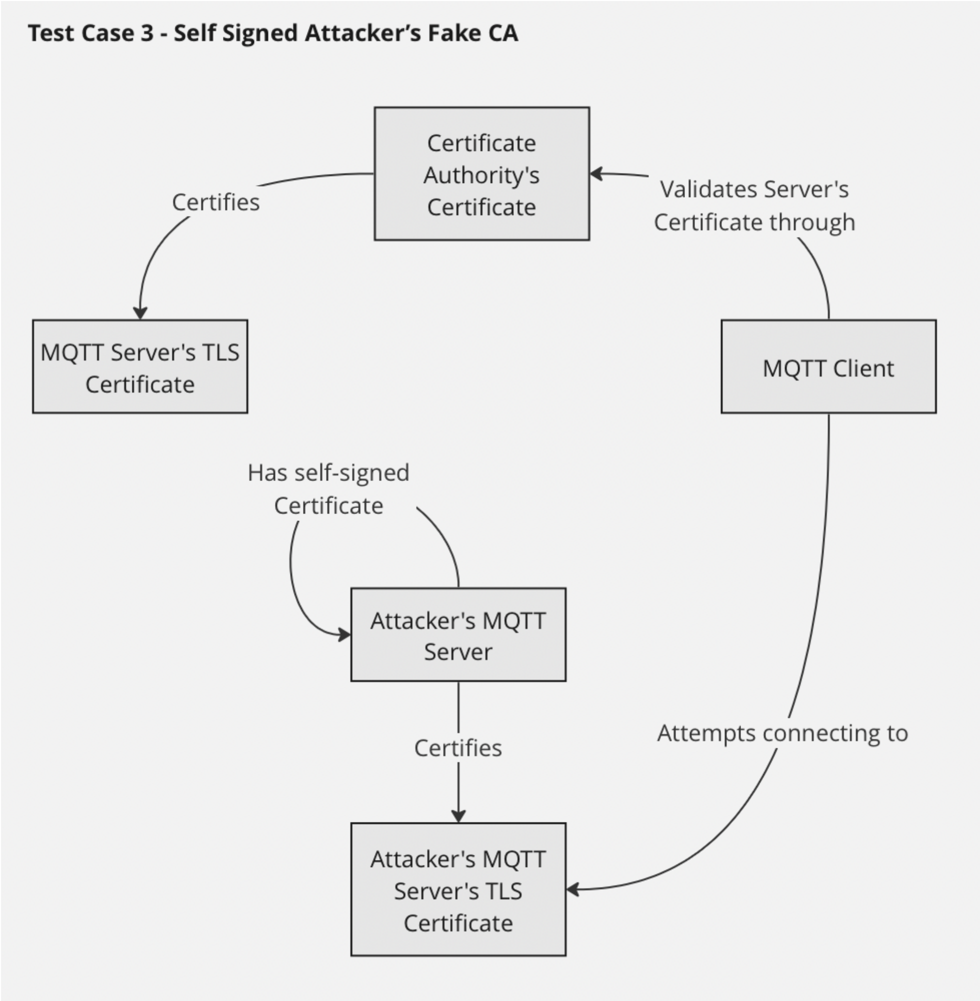
Intruder Access Capabilities	The Intruder impersonates an MQTT Server during the TLS Handshake phase.
Intruder's Attack description	The Intruder creates a self-signed certificate and uses it to configure the MQTT Library.
State of TLS Certificate	The TLS Certificate is self-signed by the attacker, so any field can be completely different from the Server's Certificate.
State of Certificate's Signature	The signature is ***valid***
Assertion	The Library should ***reject*** the connection when a client tries connecting to the MQTT Library configured with this certificate.



4.0.3 Test Case 3 - Self Signed Attacker's Fake CA

This Test Case is set up by configuring the MQTT Broker Library with a forged TLS Certificate signed by a forged Root Certificate Authority. The Tester Client connects to the server checking the Server TLS Certificate against the real Certificate Authority's Certificate. The table of the Unit Test is as follows:

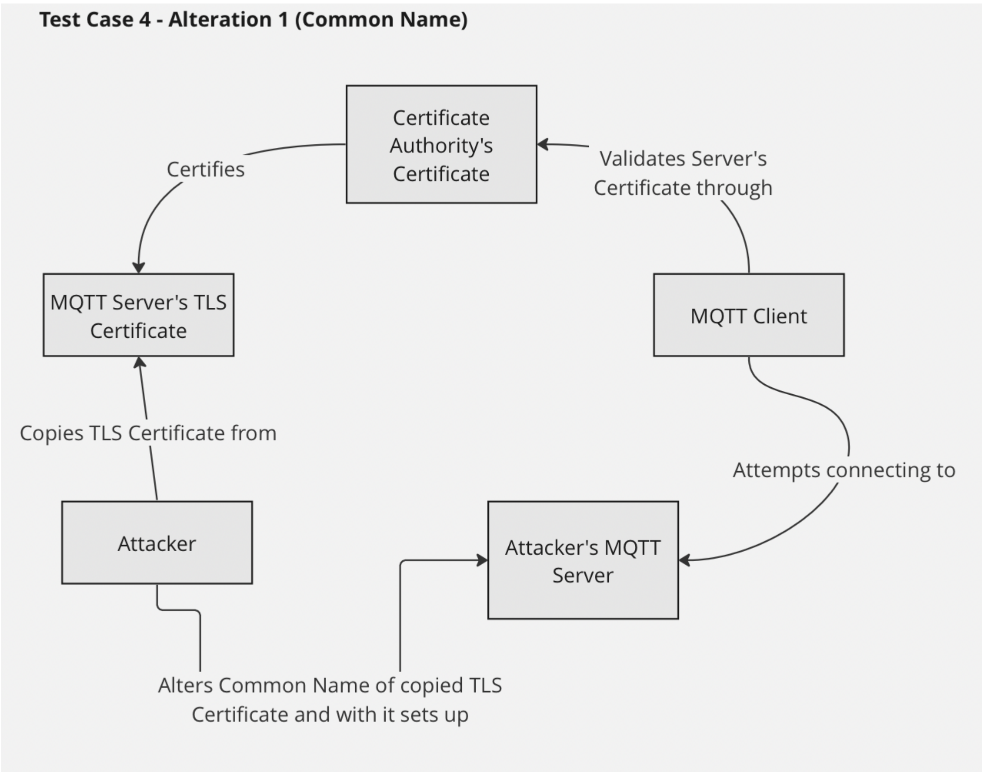
Intruder Access Capabilities	The Intruder impersonates an MQTT Server during the TLS Handshake phase.
Intruder's Attack description	The Intruder imitates the Server Certificate's chain of trust, creating their own root Certificate Authority and using it to sign their certificate. Then they use their certificate to configure the MQTT Library.
State of TLS Certificate	The TLS Certificate is imitating the Server Certificate, but it's signed by the Attacker's fake Certificate Authority.
State of Certificate's Signature	The signature is ***valid***
Assertion	The Library should ***reject*** the connection when a client tries connecting to the MQTT Library configured with this certificate.



4.0.4 Test Case 4 - Alteration 1 (Common Name)

This Test Case is set up by configuring the MQTT Broker Library with an altered TLS Certificate signed by the real Certificate Authority. The intruder alters the Common Name field, therefore the signature is compromised because the Server Certificate has been tampered with. The Tester Client connects to the server checking the Server TLS Certificate against the real Certificate Authority's Certificate. The table of the Unit Test is as follows:

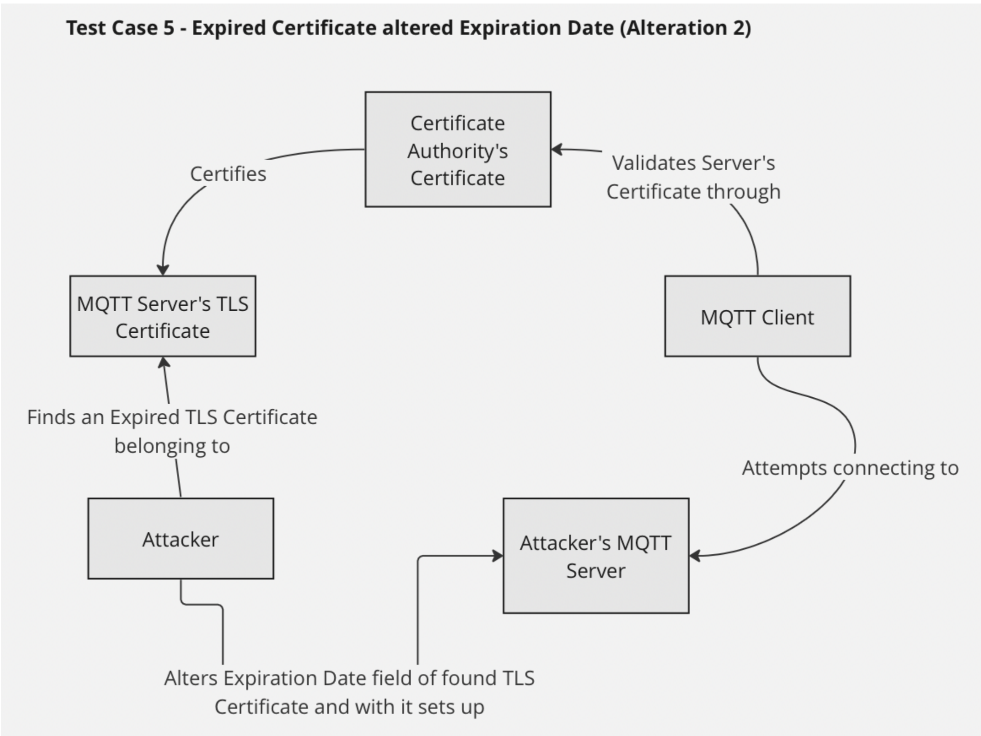
Intruder Access Capabilities	The Intruder impersonates an MQTT Server during the TLS Handshake phase.
Intruder's Attack description	The Intruder alters the Common Name field of the Server Certificate, replacing it with their own Common Name. Then they use the altered certificate to configure the MQTT Library.
State of TLS Certificate	The TLS Certificate is equal to the Server Certificate except for the Common Name field.
State of Certificate's Signature	The signature is ***not*** valid
Assertion	The Library should ***reject*** the connection when a client tries connecting to the MQTT Library configured with this certificate.



4.0.5 Test Case 5 - Alteration 2 (Expiration Date)

This Test Case is set up by configuring the MQTT Broker Library with an altered expired TLS Certificate signed by the real Certificate Authority. The intruder alters the Not Valid After field, therefore the signature is compromised because the Server Certificate has been tampered with. The Tester Client connects to the server checking the Server TLS Certificate against the real Certificate Authority's Certificate. The table of the Unit Test is as follows:

Intruder Access Capabilities	The Intruder has access to an old expired Server Certificate
Intruder's Attack description	The Intruder alters the expiration date of the expired Server Certificate, making it valid for the current date. Then the Intruder tries to configure the MQTT Library with the altered Certificate.
State of TLS Certificate	The TLS Certificate is the expired Server Certificate, but the Not Valid After field has been tampered with.
State of Certificate's Signature	The signature is ***not*** valid
Assertion	The Library should ***reject*** the connection when a client tries connecting to the MQTT Library configured with this certificate.

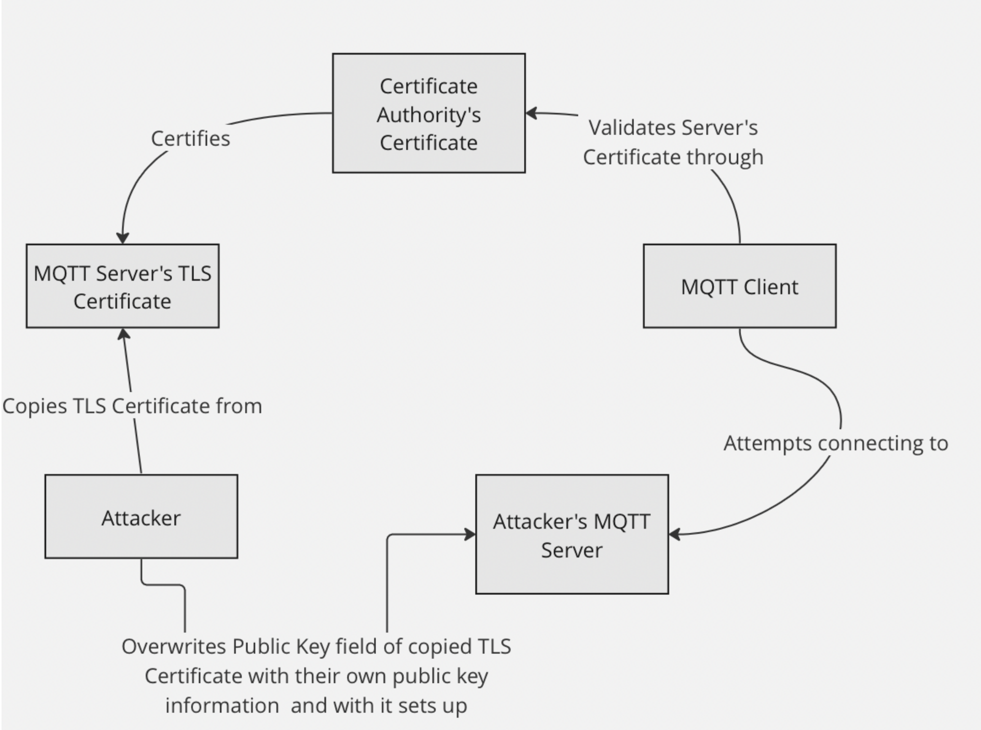


4.0.6 Test Case 6 - Alteration 3 (Public Key)

This Test Case is set up by configuring the MQTT Broker Library with an altered TLS Certificate signed by the real Certificate Authority. The intruder replaces the contents of the Public Key field with their own Public Key, therefore the signature is compromised because the Server Certificate has been tampered with. The Tester

Client connects to the server checking the Server TLS Certificate against the real Certificate Authority's Certificate. The table of the Unit Test is as follows:

Intruder Access Capabilities	The Intruder impersonates an MQTT Server during the TLS Handshake phase.
Intruder's Attack description	The Intruder alters the Public Key Info > Public Key field of the Server Certificate, replacing it with their own Public Key. Then they use the altered certificate to configure the MQTT Library.
State of TLS Certificate	The TLS Certificate is equal to the Server Certificate except for the Public Key field.
State of Certificate's Signature	The signature is ***not*** valid
Assertion	The Library should ***reject*** the connection when a client tries connecting to the MQTT Library configured with this certificate.

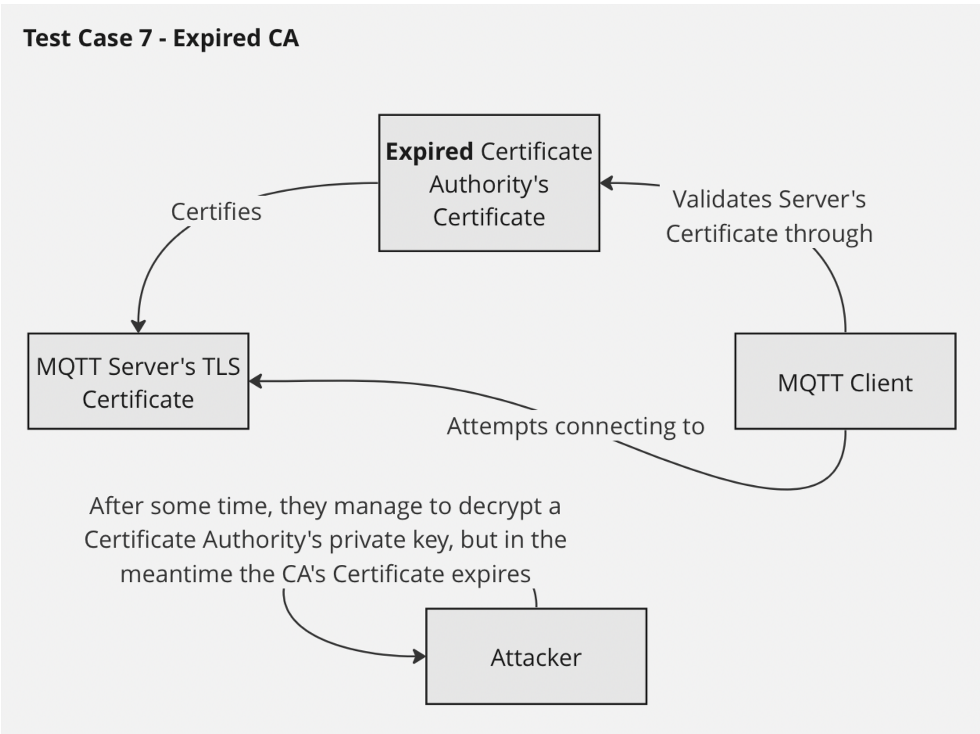


Test Case 6 - Alteration 3 (Public Key)

4.0.7 Test Case 7 - Expired CA (lteration 4)

This Test Case is set up by configuring the MQTT Broker Library with a forged TLS Certificate signed by an expired (real) Certificate Authority. This test represents a scenario in which the Intruder manages to decrypt the Certificate Authority's Public Key over a long period of time, during which the Client under attack is not updated with a new CA Certificate. Because of this, the Tester Client in this Test Case connects to the server checking the Server TLS Certificate against the expired Certificate Authority's Certificate. The table of the Unit Test is as follows:

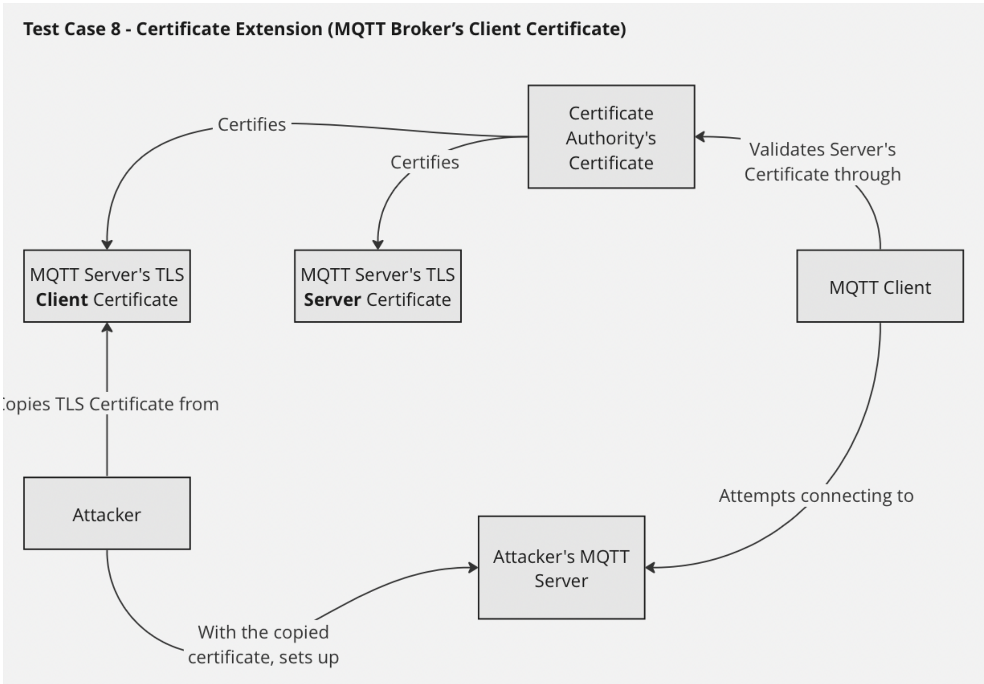
Intruder Access Capabilities	The Intruder has access to an old expired Certificate Authority Root or Intermediate Certificate
Intruder's Attack description	The Intruder tries using the formerly valid, but now expired, Certificate Authority Certificate, to sign their own certificate. Then they try using this certificate to configure the MQTT Library.
State of TLS Certificate	The TLS Certificate is a completely different certificate from the Server Certificate.
State of Certificate's Signature	The signature is ***valid***
Assertion	The Library should ***reject*** the connection when a client tries connecting to the MQTT Library configured with this certificate.



4.0.8 Test Case 8 - Certificate Extension

This Test Case is set up by configuring the MQTT Broker Library with a valid TLS Certificate signed by the real Certificate Authority, though this Certificate has been signed by the CA for the MQTT Broker to use only as a Client Certificate towards other Brokers (in mTLS). The Tester Client connects to the server checking the Server TLS Certificate against the real Certificate Authority's Certificate. The table of the Unit Test is as follows:

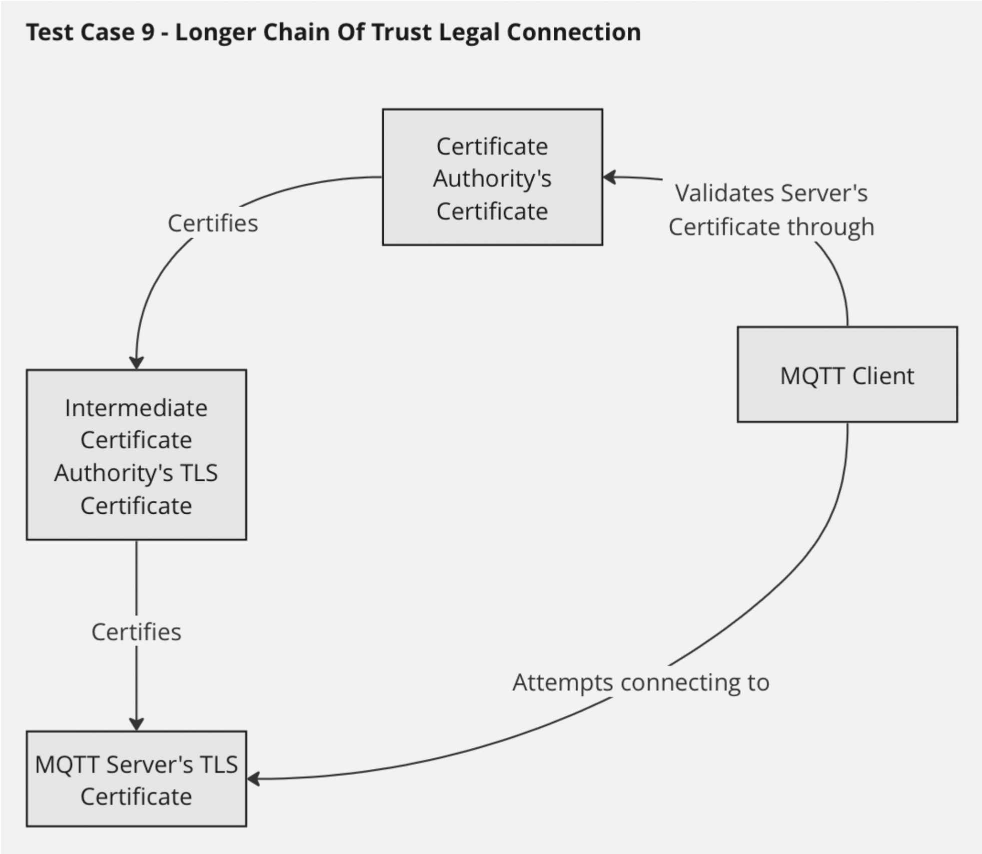
Intruder Access Capabilities	The Intruder has access to a certificate belonging to the Server's entity, but one that is used for TLS Client Authentication.
Intruder's Attack description	The Intruder tries using the TLS Client Certificate to configure the MQTT Library as a MQTT Server, hence using the certificate as a TLS Server Certificate.
State of TLS Certificate	The TLS Certificate is rightfully authenticating the MQTT Server entity, but this TLS Certificate is not intended to be used for Server Authentication.
State of Certificate's Signature	The signature is ***valid***
Assertion	The Library should ***reject*** the connection when a client tries connecting to the MQTT Library configured with this certificate.



4.0.9 Test Case 9 - Longer Chain Of Trust Legal Connection

This Test Case is set up by configuring the MQTT Broker Library with a valid TLS Certificate signed by the real Intermediate Certificate Authority, which in turn is signed by the real Root Certificate Authority. The Tester Client connects to the server checking the Server TLS Certificate against the real Root Certificate Authority's Certificate. The table of the Unit Test is as follows:

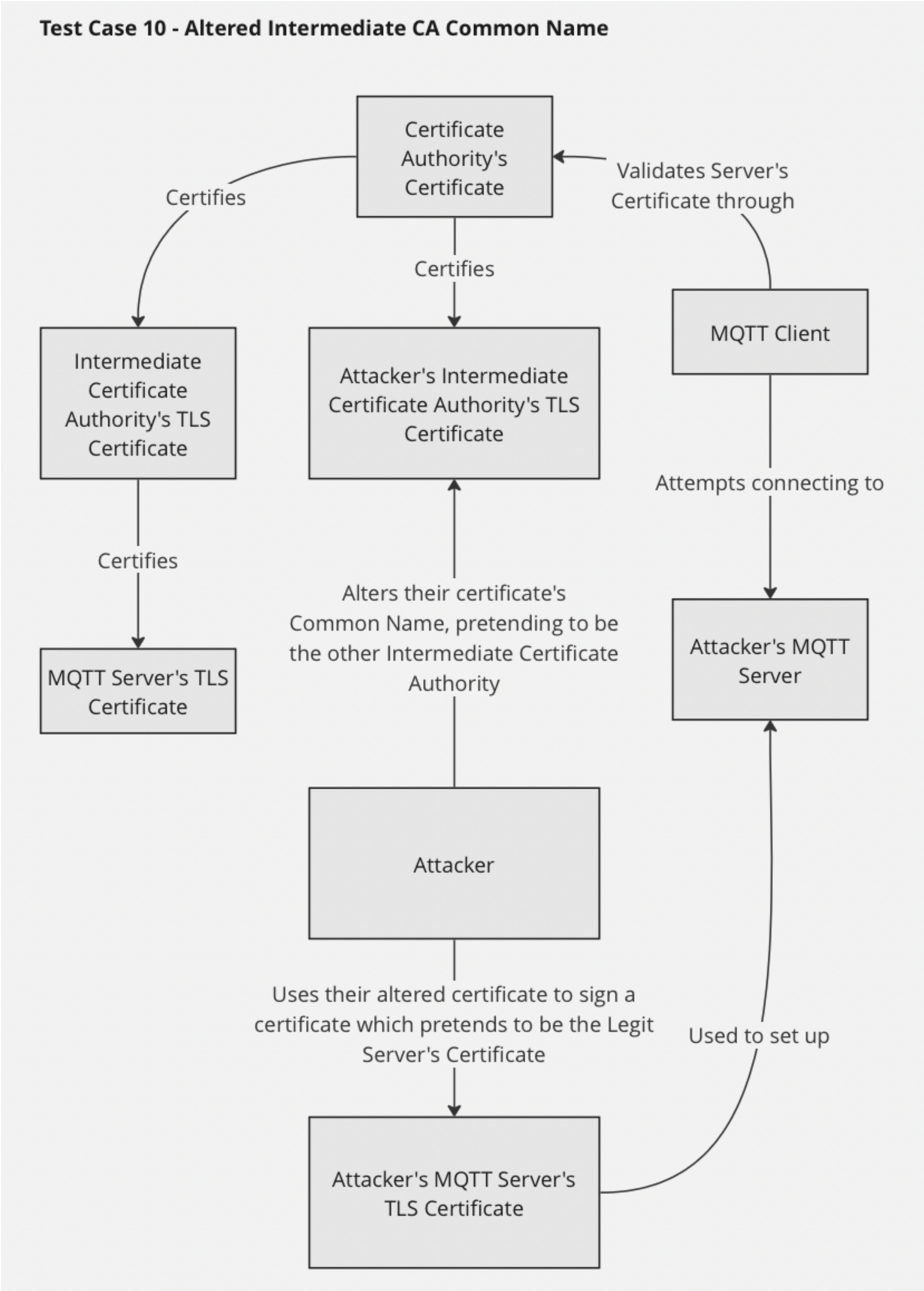
Intruder Access Capabilities	None
Intruder's Attack description	This test case represents a happy path with no intruder attack and with a longer chain of trust (Root CA + Intermediate CA).
State of TLS Certificate	The TLS Certificate we use for this test is exactly the Server's Certificate. (In this case, the Client connecting to the Server expects to receive a certificate signed by the Intermediate CA)
State of Certificate's Signature	The signature is ***valid***
Assertion	The Library should ***accept*** the connection when a client tries connecting to the MQTT Library configured with this certificate.



4.0.10 Test Case 10 - Altered Intermediate CA Common Name

This Test Case is set up by configuring the MQTT Broker Library with a forged TLS Certificate signed by an altered Intermediate Certificate Authority, which in turn is signed by the real Root Certificate Authority. The intruder alters their Intermediate CA's Common Name to pretend they are the real Intermediate CA, therefore the signature of the Intermediate CA is compromised. The Tester Client connects to the server checking the Server TLS Certificate against the real Root Certificate Authority's Certificate. The table of the Unit Test is as follows:

Intruder Access Capabilities	The Intruder owns an intermediate CA certificate signed by the Root CA.
Intruder's Attack description	The Intruder alters its certificate Common Name, trying to trick the client into believing the Intruder is signed by the real Intermediate CA.
State of TLS Certificate	The TLS Certificate is imitating the Server Certificate, but it's signed by the Attacker's fake Certificate Authority. (In this case, the Client connecting to the Server expects to receive a certificate signed by the Intermediate CA)
State of Certificate's Signature	The signature is ***not*** valid
Assertion	The Library should ***reject*** the connection when a client tries connecting to the MQTT Library configured with this certificate.

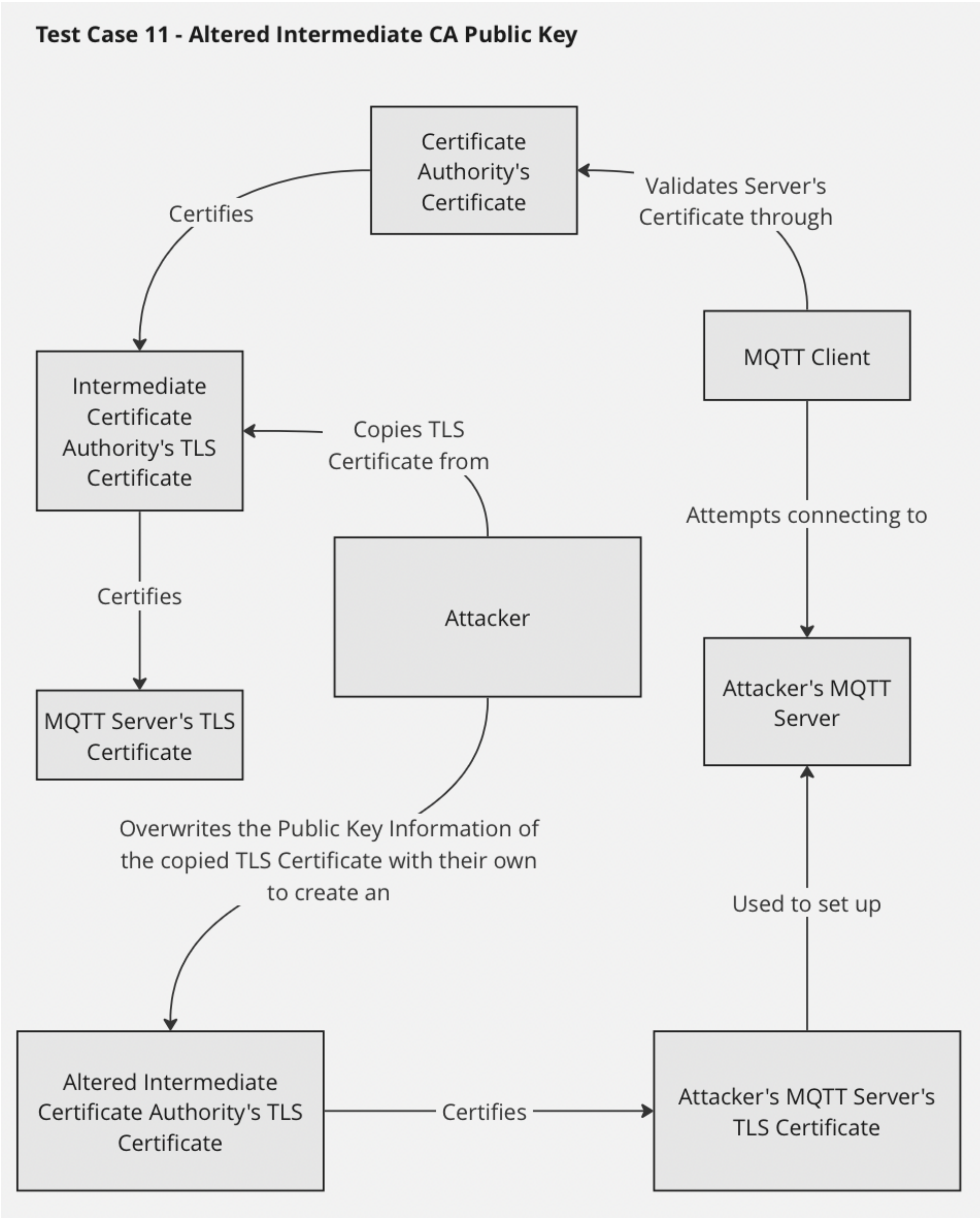


4.0.11 Test Case 11 - Altered Intermediate CA Public Key

This Test Case is set up by configuring the MQTT Broker Library with a forged TLS Certificate signed by an altered Intermediate Certificate Authority, which in turn is signed by the real Root Certificate Authority. The intruder replaces the real Intermediate CA's Public Key field contents with their own Public Key, to be able to decrypt the traffic easily, therefore the signature of the Intermediate CA is compromised. The Tester Client connects to the server checking the Server TLS

Certificate against the real Root Certificate Authority's Certificate. The table of the Unit Test is as follows:

Intruder Access Capabilities	The Intruder owns the Intermediate CA's Certificate.
Intruder's Attack description	The Intruder alters the Intermediate CA's Public Key field with their own Public Key, trying to trick the client into sending their traffic in a way that is easy to decrypt for the Intruder.
State of TLS Certificate	The TLS Certificate is imitating the Server Certificate, but it's signed by the Attacker's fake Certificate Authority. (In this case, the Client connecting to the Server expects to receive a certificate signed by the Intermediate CA)
State of Certificate's Signature	The signature is ***not*** valid
Assertion	The Library should ***reject*** the connection when a client tries connecting to the MQTT Library configured with this certificate.


Test Case 11 - Altered Intermediate CA Public Key
Certificate Authority's Certificate
Certifies
Validates Server's Certificate through
MQTT Client
Intermediate Certificate Authority's TLS Certificate
Copies TLS Certificate from
Attempts connecting to
Certifies
Attacker
MQTT Server's TLS Certificate
Attacker's MQTT Server
Overwrites the Public Key Information of the copied TLS Certificate with their own to create an
Used to set up
Altered Intermediate Certificate Authority's TLS Certificate
Certifies
Attacker's MQTT Server's TLS Certificate

Chapter 5

Developed Code

The code developed for this Internship is used to setup the laboratory environment and to execute the Unit Tests for each library. For each file there will be the code snippet followed by an explanation of the code.

5.0.1 TLS Certificates Generation Script

```
#!/bin/sh

CONTAINER_IP=$1

sh clean.sh

mkdir ca
cd ca
mkdir ca.db.certs
touch ca.db.index
echo "1234" > ca.db.serial
cd ../

mkdir second-level-ca
cd second-level-ca
mkdir ca.db.certs
touch ca.db.index
echo "1234" > ca.db.serial
cd ../

mkdir expired-ca
cd expired-ca
mkdir ca.db.certs
touch ca.db.index
echo "1234" > ca.db.serial
cd ../

mkdir fake-ca
cd fake-ca
mkdir ca.db.certs
touch ca.db.index
echo "1234" > ca.db.serial
cd ../

```

```
mkdir second-level-ca-2
cd second-level-ca-2
mkdir ca.db.certs
touch ca.db.index
echo "1234" > ca.db.serial
cd ../

mkdir second-level-ca-alt1-common-name
cd second-level-ca-alt1-common-name
mkdir ca.db.certs
touch ca.db.index
echo "1234" > ca.db.serial
cd ../

mkdir second-level-ca-alt2-public-key
cd second-level-ca-alt2-public-key
mkdir ca.db.certs
touch ca.db.index
echo "1234" > ca.db.serial
cd ../

mkdir server-certificate
mkdir attacker-certificate
mkdir alt1-common-name
mkdir alt2-expiration-date
mkdir alt3-public-key
mkdir alt4-expired-ca
mkdir fake-chain-of-trust
mkdir attacker-certificate-signed-by-altered-int-ca

# Root Certificate Authority's Certificate
openssl genrsa -out ca/ca.key 2048
openssl req -new -x509 -days 365 -key ca/ca.key -out ca/ca.pem \
-sha256 \
-subj "/C=it/ST=State/L=City/CN=Certificate Authority"

# Legit Server Certificate Request and CA Signing
openssl genrsa -out server-certificate/serverKey.pem 2048
openssl req -new -nodes -key server-certificate/serverKey.pem \
-sha256 \
-out server-certificate/serverCertificateRequest.pem \
-subj "/C=it/ST=State/L=City/CN=$CONTAINER_IP" \
-batch

openssl ca -config ca.conf -out server-certificate/serverCertificate.
    pem \
-in server-certificate/serverCertificateRequest.pem \
-batch

# Legit Server Certificate Request as Client and CA Signing
echo "unique_subject = no" > ca/ca.db.index.attr # Allow duplicate
    subjects to be signed by CA. In this case, the same subject wants
    to have a general SSL certificate and one for client
    authentication only.
openssl genrsa -out server-certificate/serverKeyAsClient.pem 2048
openssl req -new -nodes -key server-certificate/serverKeyAsClient.pem
    \
```

```
-sha256 \
-out server-certificate/serverCertificateRequestAsClient.pem \
-subj "/C=it/ST=State/L=City/CN=$CONTAINER_IP" \
-batch

openssl ca -config ca.conf -out server-certificate/
    serverCertificateAsClient.pem \
-in server-certificate/serverCertificateRequestAsClient.pem \
-extfile clientCertificateExtensions.conf \
-batch

# Intermediate Certificate Authority's Certificate Signing Request
    and Root CA Signing of it,
# then Signing the Certificate Signing Request of the Server with the
     Intermediate Certificate
openssl genrsa -out second-level-ca/ca.key 2048
openssl req -new -nodes -key second-level-ca/ca.key \
-sha256 \
-out second-level-ca/intermediateCACertificateRequest.pem \
-subj "/C=it/ST=State/L=City/CN=Intermediate Certificate Authority" \
-batch

openssl ca -config ca.conf -out second-level-ca/ca.pem \
-in second-level-ca/intermediateCACertificateRequest.pem \
-extfile intermediateCAExtensions.conf \
-batch

openssl ca -config second-level-ca.conf -out server-certificate/
    serverCertificateSignedByIntermediate.pem \
-in server-certificate/serverCertificateRequest.pem \
-batch

touch server-certificate/serverCertificateSignedByIntermediate-
    withRootCAIntegrated.pem
touch second-level-ca/ca-chain-of-trust.pem
cat second-level-ca/ca.pem ca/ca.pem > second-level-ca/ca-chain-of-
    trust.pem
cat server-certificate/serverCertificateSignedByIntermediate.pem
    second-level-ca/ca-chain-of-trust.pem > server-certificate/
    serverCertificateSignedByIntermediate-withRootCAIntegrated.pem

# Attacker's Self Signed Root Certificate
openssl genrsa -out attacker-certificate/attackerKey.pem 2048

openssl req -new -x509 -days 365 -key attacker-certificate/
    attackerKey.pem \
-sha256 \
-out attacker-certificate/attackerCertificate.der \
-outform DER \
-subj "/C=it/ST=State/L=City/CN=False Server" \
-batch

# Fake Chain of Trust (Attacker uses a self signed certificate as
    Root Certificate Authority)
openssl genrsa -out fake-ca/ca.key 2048
openssl req -new -x509 -days 365 -key fake-ca/ca.key -out fake-ca/ca.
    pem \
```

```
-sha256 \
-subj '/C=it/ST=State/L=City/CN=Certificate Authority'

openssl req -new -nodes -key attacker-certificate/attackerKey.pem \
-sha256 \
-out fake-chain-of-trust/attackerCertificateRequest.pem \
-subj "/C=it/ST=State/L=City/CN=$CONTAINER_IP" \
-batch

openssl ca -config fake-ca.conf -out fake-chain-of-trust/
    attackerCertificate.pem \
-in fake-chain-of-trust/attackerCertificateRequest.pem \
-batch

# Second Intermediate CA
openssl genrsa -out second-level-ca-2/ca.key 2048
openssl req -new -nodes -key second-level-ca-2/ca.key \
-sha256 \
-out second-level-ca-2/intermediateCACertificateRequest.pem \
-subj "/C=it/ST=State/L=City/CN=Second Intermediate Certificate
    Authority" \
-batch

openssl ca -config ca.conf -out second-level-ca-2/ca.pem \
-in second-level-ca-2/intermediateCACertificateRequest.pem \
-extfile intermediateCAExtensions.conf \
-batch

openssl req -new -nodes -key attacker-certificate/attackerKey.pem \
-sha256 \
-out attacker-certificate-signed-by-altered-int-ca/
    attackerCertificateRequest.pem \
-subj "/C=it/ST=State/L=City/CN=$CONTAINER_IP" \
-batch

# Intermediate CA Alt 1
cp second-level-ca-2/ca.key second-level-ca-alt1-common-name/ca.key
openssl x509 -in second-level-ca-2/ca.pem \
-outform DER \
-out second-level-ca-2/ca.der

openssl x509 -in second-level-ca/ca.pem \
-outform DER \
-out second-level-ca/ca.der

~/.venv/mqtt-over-tls/bin/python3 scriptsToAlterCertificate/
    alterCommonName.py \
'second-level-ca-2/ca.der' \
'second-level-ca-alt1-common-name/ca.der' \
'second-level-ca/ca.der'

openssl x509 -in second-level-ca-alt1-common-name/ca.der \
-inform DER \
-out second-level-ca-alt1-common-name/ca.pem

openssl ca -config second-level-ca-alt1-common-name.conf -out
    attacker-certificate-signed-by-altered-int-ca/attackerCertificate-
```

```
alt1.pem \
-in attacker-certificate-signed-by-altered-int-ca/attackerCertificateRequest.pem \
-batch

touch second-level-ca-alt1-common-name/ca-chain-of-trust.pem
cat second-level-ca-alt1-common-name/ca.pem ca/ca.pem > second-level-ca-alt1-common-name/ca-chain-of-trust.pem

# Intermediate CA Alt 2
cp second-level-ca-2/ca.key second-level-ca-alt2-public-key/ca.key

~/.venv/mqtt-over-tls/bin/python3 scriptsToAlterCertificate/alterPublicKey.py \
'second-level-ca/ca.der' \
'second-level-ca-alt2-public-key/ca.der' \
'second-level-ca-2/ca.der'

openssl x509 -in second-level-ca-alt2-public-key/ca.der \
-inform DER \
-out second-level-ca-alt2-public-key/ca.pem

openssl ca -config second-level-ca-alt2-public-key.conf -out attacker-certificate-signed-by-altered-int-ca/attackerCertificate-alt2.pem \
-in attacker-certificate-signed-by-altered-int-ca/attackerCertificateRequest.pem \
-batch

touch second-level-ca-alt2-public-key/ca-chain-of-trust.pem
cat second-level-ca-alt2-public-key/ca.pem ca/ca.pem > second-level-ca-alt2-public-key/ca-chain-of-trust.pem

# Convert Signed Server Certificate to .der (ASN.1 encoding) for alteration purposes
openssl x509 -in server-certificate/serverCertificate.pem \
-outform DER \
-out server-certificate/serverCertificate.der

# Alteration 1 - Changing the Common Name
~/.venv/mqtt-over-tls/bin/python3 scriptsToAlterCertificate/alterCommonName.py \
'server-certificate/serverCertificate.der' \
'alt1-common-name/attackerCertificate.der' \
'attacker-certificate/attackerCertificate.der'

# Alteration 2 - Expired Certificate
~/.venv/mqtt-over-tls/bin/python3 scriptsToAlterCertificate/alterExpirationDate.py

# Alteration 3 - Replacing the Public Key
~/.venv/mqtt-over-tls/bin/python3 scriptsToAlterCertificate/alterPublicKey.py \
'server-certificate/serverCertificate.der' \
'alt3-public-key/attackerCertificate.der' \
'attacker-certificate/attackerCertificate.der'
```

```
# Alteration 4 - Certificate signed by an Expired Certificate
    Authority Certificate
openssl x509 -in ca/ca.pem -out expired-ca/caCopy.der -outform DER
cp ca/ca.key expired-ca/ca.key
~/.venv/mqtt-over-tls/bin/python3 scriptsToAlterCertificate/
    alterCertificateAuthorityExpirationDate.py
openssl x509 -in expired-ca/ca.der -out expired-ca/ca.pem -inform DER

openssl req -new -nodes -key attacker-certificate/attackerKey.pem \
-sha256 \
-out alt4-expired-ca/attackerCertificateRequest.pem \
-subj "/C=it/ST=State/L=City/CN=$CONTAINER_IP" \
-batch

openssl ca -config expired-ca.conf -out alt4-expired-ca/
    attackerCertificate.pem \
-in alt4-expired-ca/attackerCertificateRequest.pem \
-batch

# For each Attacker Certificate, convert from .der to .pem for MQTT
    Library
openssl x509 -inform DER -in attacker-certificate/attackerCertificate
    .der -out attacker-certificate/attackerCertificate.pem
openssl x509 -inform DER -in alt1-common-name/attackerCertificate.der
     -out alt1-common-name/attackerCertificate.pem
openssl x509 -inform DER -in alt2-expiration-date/attackerCertificate
    .der -out alt2-expiration-date/attackerCertificate.pem
openssl x509 -inform DER -in alt3-public-key/attackerCertificate.der
    -out alt3-public-key/attackerCertificate.pem
```

This script, '**setupCertificates.sh**', is the main piece of code which creates the certificates for all the actors involved in the above defined Unit Tests, using the OpenSSL library to do so. When executed, we need to pass as argument the IP address of the MQTT Library Docker Container which will act as our MQTT Server. The reason we need this argument is that, when generating the certificates, we will need to specify this IP as the Server Certificate's Common Name. To start off, the script calls the subscript '**clean.sh**' to remove existing certificates and folders that we are going to create later. This is done in order to allow the script to be called multiple times if needed. The script then proceeds creating the following Certificate Authority folders:

- ca: this is the Root Certificate Authority used to sign the MQTT Server's TLS Certificate.

- second-level-ca: this is the Intermediate Certificate Authority, signed by the Root CA and used to sign the MQTT Server's TLS Certificate in the Test Case 9.

- expired-ca: this is a Root Certificate Authority which has been used to sign the MQTT Server's TLS Certificate but is now expired.

- fake-ca: this is a Root Certificate Authority forged by the Attacker to look like the real Root CA.

- second-level-ca-2: this is an Intermediate Certificate Authority owned by the

Attacker and signed by the Root CA. It is used by the Attacker in Test Case 10 to pretend to be the real Intermediate Certificate Authority.

- second-level-ca-alt1-common-name: this is the destination folder of the Attacker's Intermediate Certificate Authority after they tampered with its Common Name field.
- second-level-ca-alt2-public-key: this is the destination folder of the altered Intermediate Certificate Authority after the attacker tampered with its Public Key field. Used in Test Case 11.

The script then proceeds creating the following TLS Certificate folders:

- server-certificate: this folder will contain the TLS Certificate belonging to the MQTT Server.
- attacker-certificate: this folder will contain the self-signed TLS Certificate belonging to the Attacker.
- alt1-common-name: this folder will contain the MQTT Server's Altered TLS Certificate, after the attacker tampered with its Common Name field.
- alt2-expiration-date: this folder will contain the MQTT Server's Altered TLS Certificate, after the attacker tampered with its Not Valid After field.
- alt3-public-key: this folder will contain the MQTT Server's Altered TLS Certificate, after the attacker tampered with its Public Key field.
- alt4-expired-ca: this folder will contain the Attacker's TLS Certificate signed by the expired Root CA.
- fake-chain-of-trust: this folder will contain the Attacker's TLS Certificate signed by their Fake Root Certificate Authority.
- attacker-certificate-signed-by-altered-int-ca: this folder will contain the Attacker's Intermediate CA Altered TLS Certificates, for Test Case 10 (Altered CA's Common Name) and Test Case 11 (Altered CA's Public Key)

The script then generates, in order:

- the Root Certificate Authority's Private Key and TLS self-signed Certificate:
  - ca/ca.key
  - ca/ca.pem
- the Legit MQTT Server's Private Key:
  - server-certificate/serverKey.pem
- the Legit MQTT Server's TLS Certificate signed by the Root CA:
  - server-certificate/serverCertificate.pem

- the Legit MQTT Server's Private Key for usage as a Client:
  - server-certificate/serverKeyAsClient.pem
- the Legit MQTT Server's TLS Certificate for usage as a Client, signed by the Root CA:
  - server-certificate/serverCertificateAsClient.pem
- the Intermediate Certificate Authority's Private Key:
  - second-level-ca/ca.key
- the Intermediate Certificate Authority TLS Certificate signed by the Root CA:
  - second-level-ca/ca.pem
- the Legit MQTT Server's TLS Certificate signed by the Intermediate CA:
  - server-certificate/serverCertificateSignedByIntermediate.pem
- the concatenation of Root CA's and Intermediate CA's TLS Certificates:
  - second-level-ca/ca-chain-of-trust.pem
- the Attacker's Private Key and TLS self-signed Certificate:
  - attacker-certificate/attackerKey.pem
  - attacker-certificate/attackerCertificate.pem
- the Attacker's Fake Root Certificate Authority's Private Key and TLS self-signed Certificate:
  - fake-ca/ca.key
  - fake-ca/ca.pem
- the Attacker's TLS Certificate signed by the Fake Root CA:
  - fake-chain-of-trust/attackerCertificate.pem
- a second (different) Intermediate Certificate Authority's Private Key:
  - second-level-ca-2/ca.key
- the TLS Certificate of the second Intermediate CA, signed by the Root CA:
  - second-level-ca-2/ca.pem
- the Alteration 1 (Common Name) of the second Intermediate CA's TLS Certificate:
  - second-level-ca-alt1-common-name/ca.der
- the Attacker's TLS Certificate signed by the Altered Intermediate CA (Alteration 1):

  - attacker-certificate-signed-by-altered-int-ca/attackerCertificate-alt1.pem

- the concatenation of Root CA's and Altered Intermediate CA (Alteration 1)'s TLS Certificates:
  - second-level-ca-alt1-common-name/ca-chain-of-trust.pem
- the Alteration 2 (Public Key) of the second Intermediate CA's TLS Certificate:
  - second-level-ca-alt2-public-key/ca.der
- the Attacker's TLS Certificate signed by the Altered Intermediate CA (Alteration 2):
  - attacker-certificate-signed-by-altered-int-ca/attackerCertificate-alt2.pem
- the concatenation of Root CA's and Altered Intermediate CA (Alteration 2)'s TLS Certificates:
  - second-level-ca-alt2-public-key/ca-chain-of-trust.pem
- the Alteration 1 (Common Name) of the Legit MQTT Server's TLS Certificate:
  - alt1-common-name/attackerCertificate.der
- the Alteration 2 (Expiration Date) of the Legit MQTT Server's TLS Certificate:
  - alt2-expiration-date/attackerCertificate.der
- the Alteration 3 (Public Key) of the Legit MQTT Server's TLS Certificate:
  - alt3-public-key/attackerCertificate.der
- the Expired Root CA's TLS Certificate:
  - expired-ca/ca.pem
- the Attacker's TLS Certificate signed by the Expired CA:
  - alt4-expired-ca/attackerCertificate.der

Lastly, the script converts back to '**.pem**' all the certificates that were saved in '**.der**' extension by the alteration scripts.

5.0.2 TLS Certificate Common Name Alteration Script

```python
from pyasn1.codec.der.decoder import decode
from pyasn1.codec.der.encoder import encode
from pyasn1_modules import rfc2459
import sys

# Usage: this script takes 3 arguments:
#  1 - Certificate to be altered
#  2 - Destination path where to save the altered certificate
#  3 - Certificate to use as reference for altering the common name
```

```
with open(sys.argv[1], 'rb') as fileInput, \
open(sys.argv[2], 'wb') as fileOutput, \
open(sys.argv[3], 'rb') as alterationReferenceFileInput:
  certificateToAlter, restOfCertificate = decode(fileInput.read(),
    asn1Spec=rfc2459.Certificate())
  assert not restOfCertificate
  referenceCertificate, _ = decode(alterationReferenceFileInput.read
    (), asn1Spec=rfc2459.Certificate())
  certificateToAlter['tbsCertificate']['subject'] =
    referenceCertificate['tbsCertificate']['subject']
  outputSubstrate = encode(certificateToAlter)
  fileOutput.write(outputSubstrate)
  print("Finished saving Alteration 1 - Common Name in " + sys.argv
    [2])
```

Because this script is used to alter both the Legit MQTT Server's TLS Certificate and the second Intermediate CA's TLS Certificate, the script is designed to have 3 inputs:

- Path of the certificate to be altered.
- Path of the destination where the altered certificate will be saved.
- Path of the reference certificate that will be used to copy and paste the Common Name from.

The script uses the '**pyasn1**' , '**pyasn1_modules**' and '**sys**' libraries to:

1. Read the certificate to be altered and the reference certificate from disk.
2. Decode the certificate to be altered and the reference certificate, from ASN1-base64-encoded data to a Dictionary data structure.
3. Overwrite the **Common Name** of the certificate to be altered with the **Common Name** of the reference Certificate.
4. Encode the resulting Altered Certificate from a Dictionary data structure to a ASN1-base64-encoded data.
5. Save the Altered Certificate on disk.

5.0.3 TLS Certificate Expiration Date Alteration Script

```
from pyasn1.codec.der.decoder import decode
from pyasn1.codec.der.encoder import encode
from pyasn1_modules import rfc2459

with open('server-certificate/serverCertificate.der', 'rb') as
    fileInput, \
open('alt2-expiration-date/attackerCertificate.der', 'wb') as
    fileOutput:
  certificate, restOfCertificate = decode(fileInput.read(), asn1Spec=
    rfc2459.Certificate())
  assert not restOfCertificate
```

```
  certificate['tbsCertificate']['validity']['notBefore']['utcTime'] =
     "010530070422Z"
  certificate['tbsCertificate']['validity']['notAfter']['utcTime'] =
    "400530070422Z"
  outputSubstrate = encode(certificate)
  fileOutput.write(outputSubstrate)
  print("Finished saving Alteration 2 - Expired Certificate")
```

The script uses the '**pyasn1**' , '**pyasn1_modules**' and '**sys**' libraries to:

1. Read the certificate to be altered ('**server-certificate/serverCertificate.der**') from disk.

2. Decode the certificate to be altered from ASN1-base64-encoded data to a Dictionary data structure.

3. Change the validity range ('**Not Valid Before**' and '**Not Valid After**' fields) to a range that contains the current date, for example in the script it's changed to years 2001 until 2040.

4. Encode the resulting Altered Certificate from a Dictionary data structure to a ASN1-base64-encoded data.

5. Save the Altered Certificate on disk.

5.0.4 TLS Certificate Public Key Alteration Script

```
from pyasn1.codec.der.decoder import decode
from pyasn1.codec.der.encoder import encode
from pyasn1_modules import rfc2459
import sys

# Usage: this script takes 3 arguments:
#  1 - Certificate to be altered
#  2 - Destination path where to save the altered certificate
#  3 - Certificate to use as reference for altering the public key

with open(sys.argv[1], 'rb') as fileInput, \
open(sys.argv[2], 'wb') as fileOutput, \
open(sys.argv[3], 'rb') as alterationReferenceFileInput:
  certificateToAlter, restOfCertificate = decode(fileInput.read(),
    asn1Spec=rfc2459.Certificate())
  assert not restOfCertificate
  referenceCertificate, _ = decode(alterationReferenceFileInput.read
    (), asn1Spec=rfc2459.Certificate())
  certificateToAlter['tbsCertificate']['subjectPublicKeyInfo'] =
    referenceCertificate['tbsCertificate']['subjectPublicKeyInfo']
  outputSubstrate = encode(certificateToAlter)
  fileOutput.write(outputSubstrate)
  print("Finished saving Alteration 3 - Public Key in " + sys.argv
    [2])
```

Because this script is used to alter both the Legit MQTT Server's TLS Certificate and the Intermediate CA's TLS Certificate, the script is designed to have 3 inputs:

- Path of the certificate to be altered.

- Path of the destination where the altered certificate will be saved.
- Path of the reference certificate that will be used to copy and paste the Common Name from.

The script uses the '**pyasn1**' , '**pyasn1__modules**' and '**sys**' libraries to:

1. Read the certificate to be altered and the reference certificate from disk.
2. Decode the certificate to be altered and the reference certificate, from ASN1-base64-encoded data to a Dictionary data structure.
3. Overwrite the **Public Key** of the certificate to be altered with the **Public Key** of the reference Certificate.
4. Encode the resulting Altered Certificate from a Dictionary data structure to a ASN1-base64-encoded data.
5. Save the Altered Certificate on disk.

5.0.5 TLS Certificate Keystores Generation Script

```
#!bin/sh

CERTIFICATES='certificates'

rm -rf keystores
mkdir keystores

# Generate TLS Certificates for Test Cases, in case it was not already done for other libraries tests
cd $CERTIFICATES
sh setupCertificates.sh $1
cd -

# Convert all test-case certificates into .p12 stores (PKCS12)
openssl pkcs12 -export \
-in $CERTIFICATES/server-certificate/serverCertificate.pem \
-inkey $CERTIFICATES/server-certificate/serverKey.pem \
-passout pass:hivemqStorePassword \
-name hivemq \
-out keystores/serverCertificate.p12

openssl pkcs12 -export \
-in $CERTIFICATES/server-certificate/serverCertificateSignedByIntermediate-withRootCAIntegrated.pem \
-inkey $CERTIFICATES/server-certificate/serverKey.pem \
-passout pass:hivemqStorePassword \
-name hivemq \
-out keystores/serverCertificateSignedByIntermediate.p12

openssl pkcs12 -export \
-in $CERTIFICATES/server-certificate/serverCertificateAsClient.pem \
-inkey $CERTIFICATES/server-certificate/serverKeyAsClient.pem \
-passout pass:hivemqStorePassword \
-name hivemq \
```

```
-out keystores/serverCertificateAsClient.p12

openssl pkcs12 -export \
-in $CERTIFICATES/attacker-certificate/attackerCertificate.pem \
-inkey $CERTIFICATES/attacker-certificate/attackerKey.pem \
-passout pass:hivemqStorePassword \
-name hivemq \
-out keystores/attackerCertificate.p12

openssl pkcs12 -export \
-in $CERTIFICATES/fake-chain-of-trust/attackerCertificate.pem \
-inkey $CERTIFICATES/attacker-certificate/attackerKey.pem \
-passout pass:hivemqStorePassword \
-name hivemq \
-out keystores/attackerCertificateFakeChainOfTrust.p12

openssl pkcs12 -export \
-in $CERTIFICATES/attacker-certificate-signed-by-altered-int-ca/attackerCertificate-alt1.pem \
-inkey $CERTIFICATES/attacker-certificate/attackerKey.pem \
-passout pass:hivemqStorePassword \
-name hivemq \
-out keystores/attackerCertificateAlteredIntCACommonName.p12

openssl pkcs12 -export \
-in $CERTIFICATES/attacker-certificate-signed-by-altered-int-ca/attackerCertificate-alt2.pem \
-inkey $CERTIFICATES/attacker-certificate/attackerKey.pem \
-passout pass:hivemqStorePassword \
-name hivemq \
-out keystores/attackerCertificateAlteredIntCAPublicKey.p12

openssl pkcs12 -export \
-in $CERTIFICATES/alt1-common-name/attackerCertificate.pem \
-inkey $CERTIFICATES/server-certificate/serverKey.pem \
-passout pass:hivemqStorePassword \
-name hivemq \
-out keystores/alteration1CommonName.p12

openssl pkcs12 -export \
-in $CERTIFICATES/alt2-expiration-date/attackerCertificate.pem \
-inkey $CERTIFICATES/server-certificate/serverKey.pem \
-passout pass:hivemqStorePassword \
-name hivemq \
-out keystores/alteration2ExpirationDate.p12

openssl pkcs12 -export \
-in $CERTIFICATES/alt3-public-key/attackerCertificate.pem \
-inkey $CERTIFICATES/attacker-certificate/attackerKey.pem \
-passout pass:hivemqStorePassword \
-name hivemq \
-out keystores/alteration3PublicKey.p12

openssl pkcs12 -export \
-in $CERTIFICATES/alt4-expired-ca/attackerCertificate.pem \
-inkey $CERTIFICATES/attacker-certificate/attackerKey.pem \
-passout pass:hivemqStorePassword \
```

```
-name hivemq \
-out keystores/alteration4ExpiredCA.p12

# For each PKCS12, convert to JKS for HiveMQ
cd keystores

keytool -importkeystore \
-srckeystore serverCertificate.p12 -srcstoretype PKCS12 \
-destkeystore serverCertificate.jks -deststoretype JKS \
-srcstorepass hivemqStorePassword \
-deststorepass hivemqStorePassword

keytool -importkeystore \
-srckeystore serverCertificateSignedByIntermediate.p12 -srcstoretype PKCS12 \
-destkeystore serverCertificateSignedByIntermediate.jks -deststoretype JKS \
-srcstorepass hivemqStorePassword \
-deststorepass hivemqStorePassword

keytool -importkeystore \
-srckeystore serverCertificateAsClient.p12 -srcstoretype PKCS12 \
-destkeystore serverCertificateAsClient.jks -deststoretype JKS \
-srcstorepass hivemqStorePassword \
-deststorepass hivemqStorePassword

keytool -importkeystore \
-srckeystore attackerCertificate.p12 -srcstoretype PKCS12 \
-destkeystore attackerCertificate.jks -deststoretype JKS \
-srcstorepass hivemqStorePassword \
-deststorepass hivemqStorePassword

keytool -importkeystore \
-srckeystore attackerCertificateFakeChainOfTrust.p12 -srcstoretype PKCS12 \
-destkeystore attackerCertificateFakeChainOfTrust.jks -deststoretype JKS \
-srcstorepass hivemqStorePassword \
-deststorepass hivemqStorePassword

keytool -importkeystore \
-srckeystore attackerCertificateAlteredIntCACommonName.p12 -srcstoretype PKCS12 \
-destkeystore attackerCertificateAlteredIntCACommonName.jks -deststoretype JKS \
-srcstorepass hivemqStorePassword \
-deststorepass hivemqStorePassword

keytool -importkeystore \
-srckeystore attackerCertificateAlteredIntCAPublicKey.p12 -srcstoretype PKCS12 \
-destkeystore attackerCertificateAlteredIntCAPublicKey.jks -deststoretype JKS \
-srcstorepass hivemqStorePassword \
-deststorepass hivemqStorePassword

keytool -importkeystore \
```

```
-srckeystore alteration1CommonName.p12 -srcstoretype PKCS12 \
-destkeystore alteration1CommonName.jks -deststoretype JKS \
-srcstorepass hivemqStorePassword \
-deststorepass hivemqStorePassword

keytool -importkeystore \
-srckeystore alteration2ExpirationDate.p12 -srcstoretype PKCS12 \
-destkeystore alteration2ExpirationDate.jks -deststoretype JKS \
-srcstorepass hivemqStorePassword \
-deststorepass hivemqStorePassword

keytool -importkeystore \
-srckeystore alteration3PublicKey.p12 -srcstoretype PKCS12 \
-destkeystore alteration3PublicKey.jks -deststoretype JKS \
-srcstorepass hivemqStorePassword \
-deststorepass hivemqStorePassword

keytool -importkeystore \
-srckeystore alteration4ExpiredCA.p12 -srcstoretype PKCS12 \
-destkeystore alteration4ExpiredCA.jks -deststoretype JKS \
-srcstorepass hivemqStorePassword \
-deststorepass hivemqStorePassword
```

Some of the tested MQTT Libraries work with Java Keystores (JKS) instead of retrieving the TLS Certificates from absolute paths. Therefore this script is designed to save the TLS Certificates generated by the '**setupCertificates.sh**' script into Java Keystores. More specifically, the script applies these steps to the certificates and private keys of each Test Case:

1. Save a '**.p12**' keystore containing the Server's Certificate and Private Key, using the '**openssl pkcs12**' tool.

2. Convert the '**.p12**' keystore to a '**.jks**' keystore using the **default-jre**'s '**keytool**' command.

All the derived keystores are saved in the 'keystores' folder.

5.0.6 MQTT Client Tester Script

```
#!/bin/sh

NC='\033[0;0m'
RED='\033[0;31m'
GREEN='\033[0;32m'

echo "Setting up MQTT client subscription..."

if [ -z $2 ];
then
    caPath=certificates/ca/ca.pem
else
    caPath=$2
fi

echo "Using validation ca certificate on tester client: $caPath"
mosquitto_sub -h $1 -p 8883 -i 'subid' -t 'test' --cafile $caPath &
```

```
subPID=$!

sleep 1.5

echo "Killing $subPID"
kill $subPID

if [ $? = "0" ]
then
    echo "${GREEN}The MQTT client subscription was working correctly${NC}"
else
    echo "${RED}The MQTT client subcription was not active${NC}"
fi
```

The Tester (MQTT Client) uses a Tester Script named '**testMQTTBroker.sh**', invoking a Mosquitto Command-Line Interface distribution subscription command to connect to the MQTT Server set up by the Test Environment. The Tester Script then checks if the subscription was active (if exit code is 0) and prints a line on **stdout** that represents the Unit Test Result. The Tester Script also accepts an optional argument to specify a Certificate Authority TLS Certificate path to use to connect to the MQTT Server. This is used mainly for **Test Case 7**, where the reference CA TLS Certificate to use must be the expired one.

5.0.7 Library Tester Script

```
#!/bin/bash
bold=$(tput bold)
normal=$(tput sgr0)

MOSQUITTO_CONTAINER_IP=$(docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' mosquittoContainer)

testConfigurations=(
    "mosquitto.conf"
    "mosquitto-self-signed.conf"
    "mosquitto-fake-ca.conf"
    "mosquitto-alt1.conf"
    "mosquitto-alt2.conf"
    "mosquitto-alt3.conf"
    "mosquitto-alt4.conf"
    "mosquitto-using-client-cert.conf"
    "mosquitto-longer-chain-of-trust.conf"
    "mosquitto-altered-common-name-longer-chain-of-trust.conf"
    "mosquitto-altered-public-key-longer-chain-of-trust.conf"
)

testTitles=(
    "Test Case 1 - Legal Connection"
    "Test Case 2 - Self Signed Attacker"
    "Test Case 3 - Self Signed Attacker Fake CA"
    "Test Case 4 - Alteration 1 (Common Name)"
    "Test Case 5 - Expired Certificate altered Expiration Date (Alteration 2)"
    "Test Case 6 - Alteration 3 (Public Key)"
    "Test Case 7 - Expired CA (for laboratory purposes, Alteration 4)"
```

```
    "Test Case 8 - Certificate Extension (MQTT Broker Client Certificate)"
    "Test Case 9 - Longer Chain Of Trust Legal Connection"
    "Test Case 10 - Altered Intermediate CA Common Name"
    "Test Case 11 - Altered Intermediate CA Public Key"
)

# Test Case 7 has to be validated against expired ca on client side, not default legal ca.
caPathOverride=(
    ""
    ""
    ""
    ""
    ""
    ""
    "certificates/expired-ca/ca.pem"
    ""
    ""
    ""
    ""
)

for i in ${!testConfigurations[@]}; do
    echo "${bold}Running ${testTitles[$i]}${normal}"
    CONFIGURATION_FOR_CURRENT_TEST=${testConfigurations[$i]}
    echo "Configuring mosquittoContainer with new certificates..."
    docker exec mosquittoContainer cp /mosquitto/config/test-configurations/$CONFIGURATION_FOR_CURRENT_TEST /mosquitto/config/mosquitto.conf
    sleep 1
    echo "Restarting mosquittoContainer..."
    docker restart mosquittoContainer
    sleep 3
    docker exec testerContainer bash -c "cd /app/src && sh testMQTTBroker.sh $MOSQUITTO_CONTAINER_IP ${caPathOverride[$i]}"
done
```

For each library an automated Tester Script has been developed. The script is designed to loop through the Test Cases, and for each test case the script will:

1. Configure the MQTT Server Docker Container with the Broker Configuration of the current Test Case.
2. Restart the MQTT Server Docker Container
3. Command the MQTT Client Docker Container to try connecting to the MQTT Server using the MQTT Client Tester Script ('**testMQTTBroker.sh**')
4. Proceed to the next Test Case

The automated script works only if, before executing it, both the MQTT Server and MQTT Client Docker Containers are running.

Chapter 6

Tested MQTT Libraries

The MQTT Libraries that were subjected to Unit Testing are some Libraries that are widely spread:

Library Name	**Tested Version**
Mosquitto	2.0.18
HiveMQ Community Edition	2023.9
Aedes	0.8.0
Moquette	0.18
EMQX	5.3.0

Chapter 7

Test Results

The Test Results for the tested MQTT Libraries can be found in the following tables. A green 'ok' represents a succesful connection, a red 'error' represents a failed connection. The first row of the table represents the expected result for each Unit Test to succeed.

Library Name	**Test Case 1**	**Test Case 2**	**Test Case 3**	**Test Case 4**	**Test Case 5**	**Test Case 6**	**Test Case 7**
No Library (Expected Results)	ok	error	error	error	error	error	error
Mosquitto	ok	error	error	error	error	error	error
HiveMQ Community Edition	ok	error	error	error	error	error	error
Aedes	ok	error	error	error	error	error	error
Moquette	ok	error	error	error	error	error	error
EMQX	ok	error	error	error	error	error	error

Library Name	**Test Case 8**	**Test Case 9**	**Test Case 10**	**Test Case 11**
No Library (Expected Results)	error	ok	error	error
Mosquitto	error	ok	error	error
HiveMQ Community Edition	error	ok	error	error
Aedes	error	ok	error	error
Moquette	error	ok	error	error
EMQX	error	ok	error	error

Chapter 8

Docker Test Environment

8.0.1 MQTT Client Tester Image

For the Docker Test Environment, the main piece of work was to setup the MQTT Client Tester Docker Container Image. The code used for the generation of this Image can be found in the following Dockerfile:

```
# Based off the docker/welcome-to-docker Image, for more info check
    MAINTAINERS.md
FROM debian:stable

WORKDIR /app

COPY ./src ./src

SHELL ["/bin/bash", "-c"]

RUN apt-get update -yq \
    && apt-get install -yq python3 python3-pip python3.11-venv git
    openssl mosquitto mosquitto-clients default-jre

RUN python3 -m venv ~/.venv/mqtt-over-tls
RUN ~/.venv/mqtt-over-tls/bin/pip3 install -r ./src/requirements.txt
```

The Dockerfile is based on the Linux Debian operating system image. It stores all its files in the folders '**/app**' and '**/app/src**'. The main files that are copied in these folders are:

- setupCertificates.sh
- setupKeystores.sh
- testMQTTBroker.sh
- requirements.txt (configuration file to install python package dependencies)
- Test configurations for: **EMQX**, **HiveMQ** and **Mosquitto**
- Automated scripts to run all Test Cases for: **EMQX**, **HiveMQ**, **Mosquitto**, **Aedes** and **Moquette**

The Dockerfile also installs all the tools and libraries needed to run the tests:

- python
- pip
- python virtual environment
- git
- openssl
- mosquitto
- mosquitto-clients
- default-jre

8.0.2 Moquette MQTT Server Image

There was no official Moquette Docker Container Image on Docker Hub, so our work included also the creation of a MQTT Server Docker Container Image which runs Moquette on startup. The code used for the generation of this Image can be found in the following Dockerfile:

```
# Based off the docker/welcome-to-docker Image, for more info check
    MAINTAINERS.md
FROM debian:stable

WORKDIR /app

COPY ./src ./src

SHELL ["/bin/bash", "-c"]

RUN apt-get update -yq \
    && apt-get install -yq openssl default-jre

ENTRYPOINT ["src/moquette/bin/moquette.sh"]
```

This Dockerfile is also based on the Linux Debian operating system image. It stores all its files in the folder '**/app/src/moquette**'. This folder contains a pre-compiled version of the Moquette MQTT Broker library. The Dockerfile also installs all the dependencies needed by Moquette to run the MQTT Server:

- openssl
- default-jre

Lastly, the Dockerfile runs on startup the script '**/app/src/moquette/bin/moquette.sh**'

8.0.3 Aedes MQTT Server Image

The official Aedes Docker Container Image found on Docker Hub had a very complex Container setup that used Docker Volumes to setup the configuration of the library. Because this procedure was not fitting our means of testing, we decided to create a custom MQTT Server Docker Container Image which runs Aedes on startup.

The code used for the generation of this Image can be found in the following Dockerfile:

```
# Based off the docker/welcome-to-docker Image, for more info check
    MAINTAINERS.md
FROM debian:stable

WORKDIR /app

COPY ./src ./src

SHELL ["/bin/bash", "-c"]

RUN apt-get update -yq \
    && apt-get install -yq openssl default-jre npm

RUN npm install aedes-cli -g

ENTRYPOINT ["src/run.sh"]
```

This Dockerfile is also based on the Linux Debian operating system image. It stores all its files in the folder '**/app/src**'. This folder contains:

- the Aedes test configurations needed to run each Test Case
- a '**run.sh**' script which simply runs aedes-cli getting the configuration from the path '**/app/src/config/conf.js**'

The Dockerfile also installs all the dependencies needed to run Aedes' MQTT Server, and it installs the latest version of Aedes itself. The dependencies are:

- openssl
- default-jre
- npm

Lastly, the Dockerfile runs on startup the above mentioned script '**run.sh**'.

Chapter 9

RouterOS CHR Tests

As part of this Internship research, we also validated the TLS implementation of the MQTT features offered by the RouterOS Cloud Hosted Router (CHR) distribution. RouterOS CHR enables a machine to act as a MQTT Client, optionally encrypting data via TLS. The tests have been done manually, and as MQTT Broker to connect to, we used the Mosquitto Library. Just like in Chapter 7, the results are summarised in the following table:

Library Name	**Test Case 1**	**Test Case 2**	**Test Case 3**	**Test Case 4**	**Test Case 5**	**Test Case 6**	**Test Case 7**
No Library (Expected Results)	ok	error	error	error	error	error	error
RouterOS CHR	ok	error	error	error	error	error	error

Library Name	**Test Case 8**	**Test Case 9**	**Test Case 10**	**Test Case 11**
No Library (Expected Results)	error	ok	error	error
RouterOS CHR	error	ok	error	error

Chapter 10

Conclusion

The development of the Unit Tests presented in this Internship Report allowed us to assert the security of the tested MQTT Broker Libraries and RouterOS CHR Operating System, which all passed the entirety of the developed Test Suite. Therefore, the results presented in Chapters 7 and 9 represent a validation of the TLS security requirements of the tested software. Furthermore, the developed Test Suite and Laboratory Environment enables us for future monitoring of the TLS security requirements of new versions of the tested libraries, or of additional MQTT Broker Libraries altogether.

Bibliography

[1] RFC 2818. *HTPP over TLS.* URL: `http://www.ietf.org/rfc/rfc2818.txt`.

[2] RFC 5280. *Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile.* URL: `http://tools.ietf.org/html/rfc5280`.

[3] RFC 8446. *TLS v1.3.* URL: `https://datatracker.ietf.org/doc/html/rfc8446`.

[4] Angelo Spognardi Edoardo Di Paolo Enrico Bassetti. "Security assessment of common open source MQTT brokers and clients". In: (2021). DOI: `https://doi.org/10.48550/arXiv.2309.03547`.