UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

Securing MQTT using TLS

::.. Objective

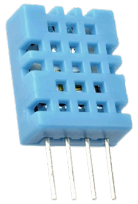
Today's lecture will focus on implementing a secure MQTT protocol for data exchange between Arduino and an MQTT Broker. Utilizing TLS security, commonly employed in HTTPS, we will see the application of MQTTS in Arduino UNO and examine Node-RED as a potential tool for building intricate systems based on sensor data. The main component involved are:

1) MQTT Broker (RabbitMQ)
2) Node-RED
3) Arduino Uno WiFi

We will explore each component in order to understand possible configuration and security mechanisms.

Architecture

::.. Architecture



RabbitMQ

Pub/Sub

1. The Arduino UNO WiFi collect data from DHT11 sensor
2. The IoT device publish the data on a RabbitMQ topic
3. Node-RED flow read from the topic

MQTT Broker (RabbitMQ)

::.. MQTT Broker (RabbitMQ)

RabbitMQ is an open-source message-broker software that originally implemented the Advanced Message Queuing Protocol (AMQP) and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol (STOMP), MQ Telemetry Transport (MQTT), and other protocols. Written in Erlang, the RabbitMQ server is built on the Open Telecom Platform framework for clustering and failover.

::.. MQTT Broker (RabbitMQ)

Some key features of RabbitMQ are:

Reliable message delivery: RabbitMQ ensures that messages are delivered to their intended recipients, even in the event of failures.

Scalability: RabbitMQ can be easily scaled up or down to handle changes in message volume.

High availability: RabbitMQ is highly available, meaning that it can withstand failures without losing messages.

Flexibility: RabbitMQ supports a variety of messaging protocols and can be used with a wide range of applications.

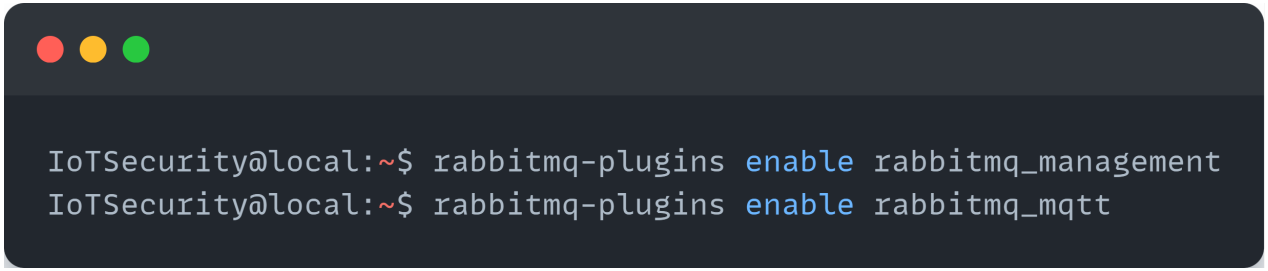
::.. MQTT Broker (RabbitMQ)

Two plug-in are needed for the aim of current project:

Management Plugin: The RabbitMQ management plugin provides an HTTP-based API for management and monitoring of RabbitMQ nodes and clusters, along with a browser-based UI and a command line tool, rabbitmqadmin.

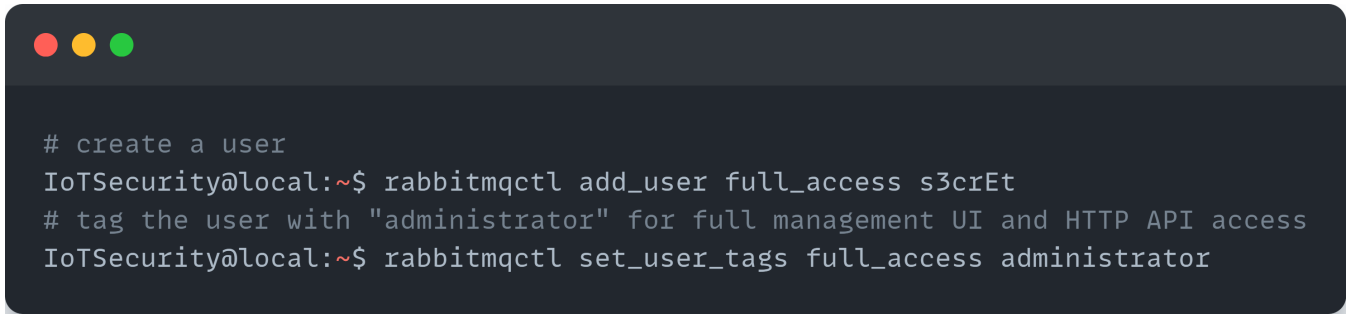
MQTT Plugin: RabbitMQ supports MQTT 3.1.1 via a plugin that ships in the core distribution.

It is possible to enable both using rabbitmq-plugins command.



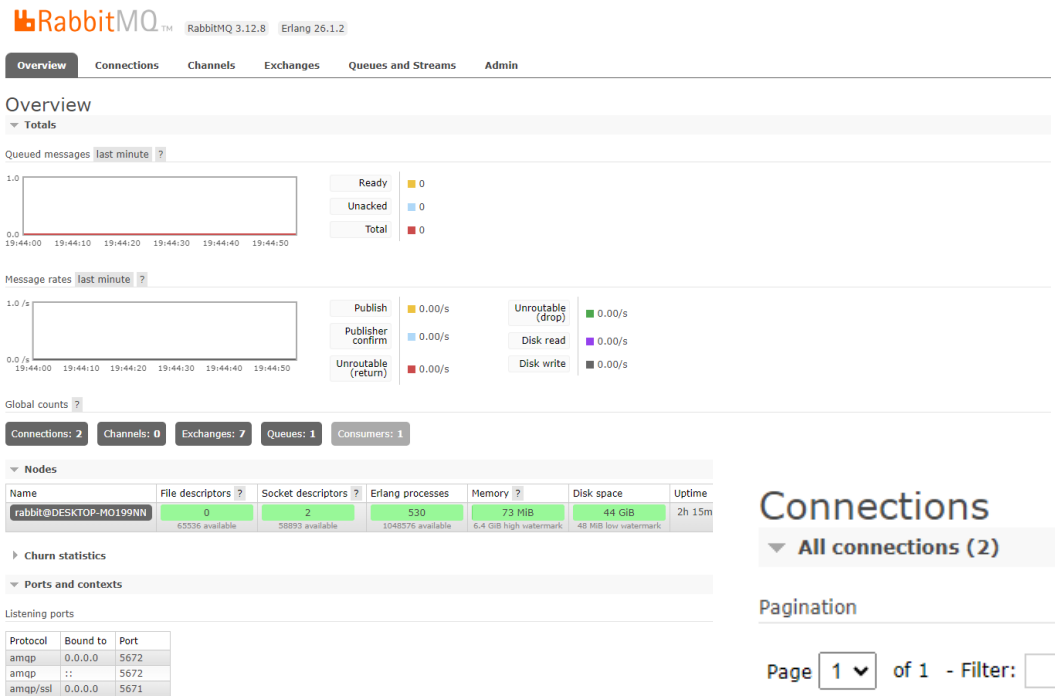
::.. Management Plugin

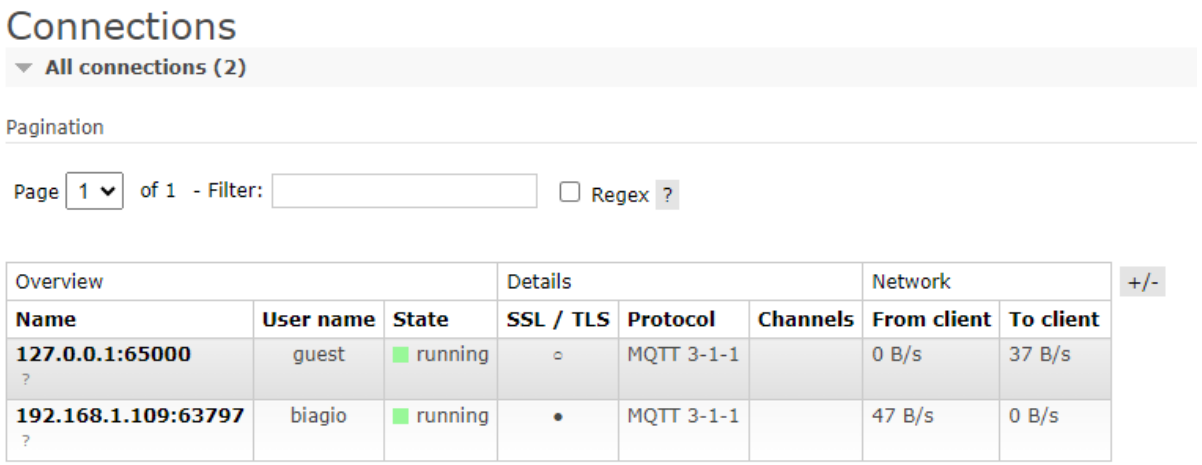
The management UI can be accessed using a Web browser at *http://{node-hostname}:*15672. The management UI requires authentication and authorization, much like RabbitMQ requires it from connecting clients. In addition to successful authentication, management UI access is controlled by user tags. The tags are managed using rabbitmqctl. Newly created users do not have any tags set on them by default.



::.. Management Plugin

Since RabbitMQ supports multiple protocols the console displays connections, channels, exchanges, queue and so on.





::.. MQTT Plugin

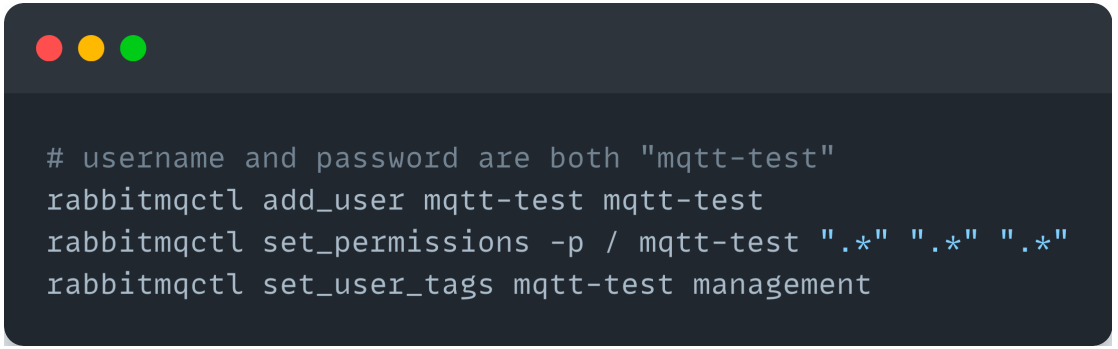
RabbitMQ supports by default AMQP. After the MQTT Plugin has been enabled, also the MQTT protocol will be available for communication with the broker.

MQTT runs by default on port 1883, it is possible to change the port on the config file.

The credentials used for the authentication are different from those used for the management pluigin. It is possible to define a new user using the following commands.

```
# username and password are both "mqtt-test"
rabbitmqctl add_user mqtt-test mqtt-test
rabbitmqctl set_permissions -p / mqtt-test ".*" ".*" ".*"
rabbitmqctl set_user_tags mqtt-test management
```

::.. MQTT Plugin

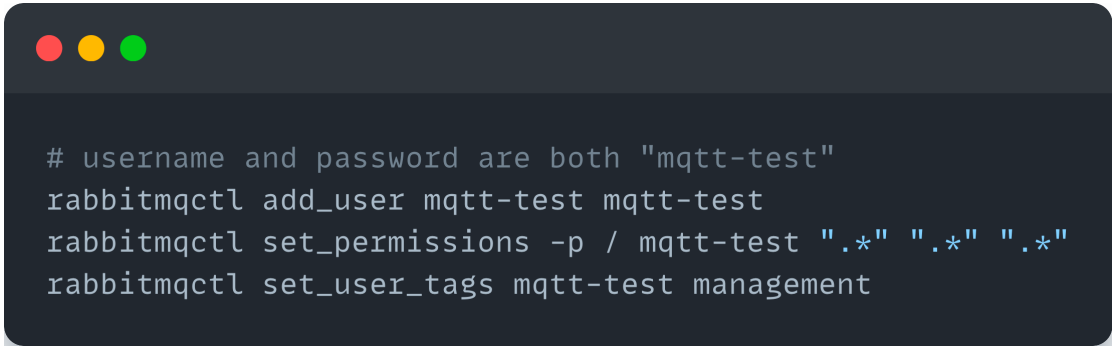


```
# username and password are both "mqtt-test"
rabbitmqctl add_user mqtt-test mqtt-test
rabbitmqctl set_permissions -p / mqtt-test ".*" ".*" ".*"
rabbitmqctl set_user_tags mqtt-test management
```

It is important to define the permissions and the tags, for enabling a good level of authorization.

In the example the user mqtt-test is able to read and write on all the topics and is tagged as a management user.

::.. MQTT Plugin Authorization



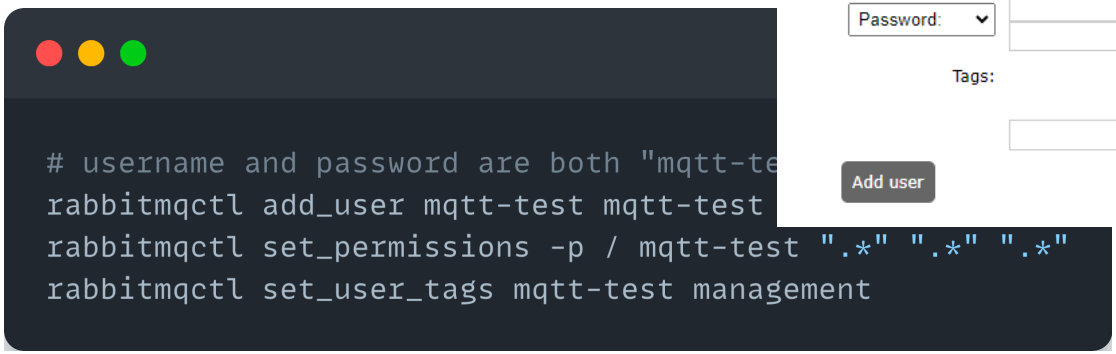
```
# username and password are both "mqtt-test"
rabbitmqctl add_user mqtt-test mqtt-test
rabbitmqctl set_permissions -p / mqtt-test ".*" ".*" ".*"
rabbitmqctl set_user_tags mqtt-test management
```

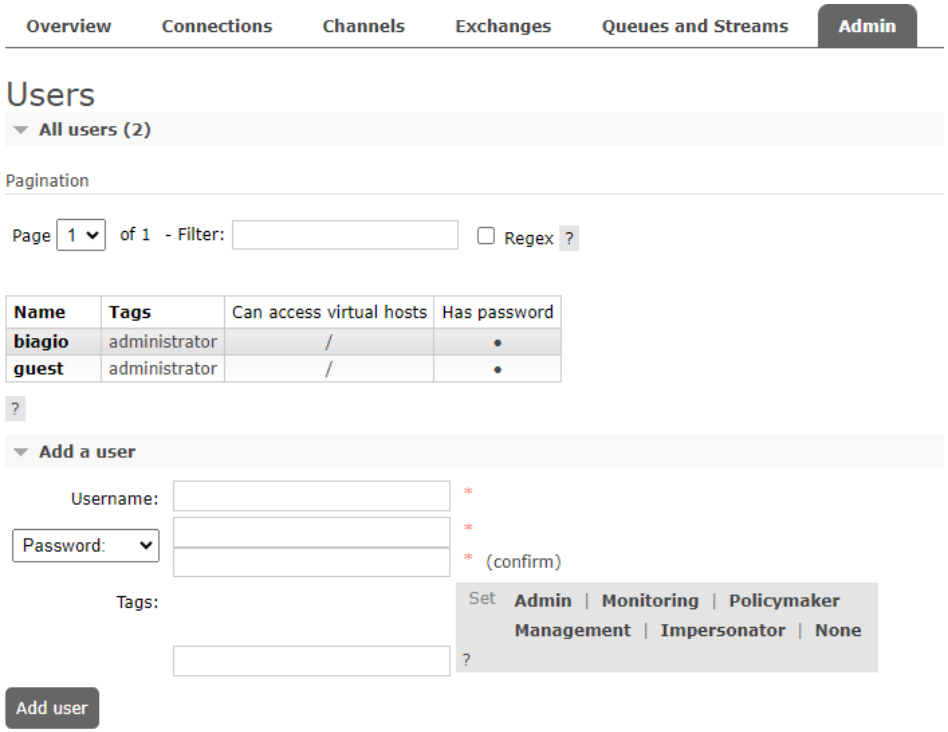
It is important to define the permissions and the tags, for enabling a good level of authorization.

In the example, the user mqtt-test can configure, read and write on all the topics and is tagged as a management user.

::.. MQTT Plugin Authorization

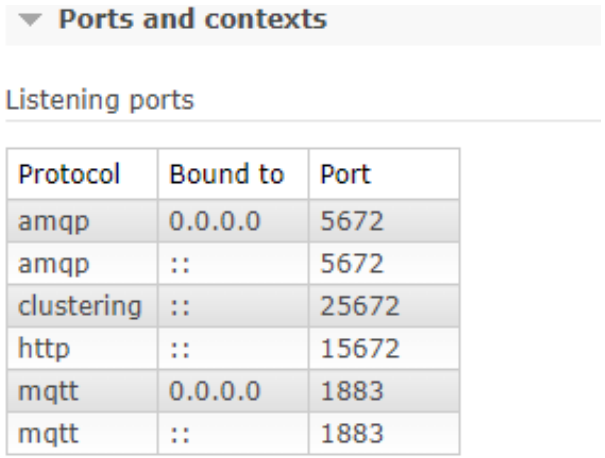
The same operation can be performed using the management plugin.





::.. Securing RabbitMQ Protocols

Once we configured management and MQTT, it is possible to notice all the protocols implemented over RabbitMQ Broker.



Ports and contexts

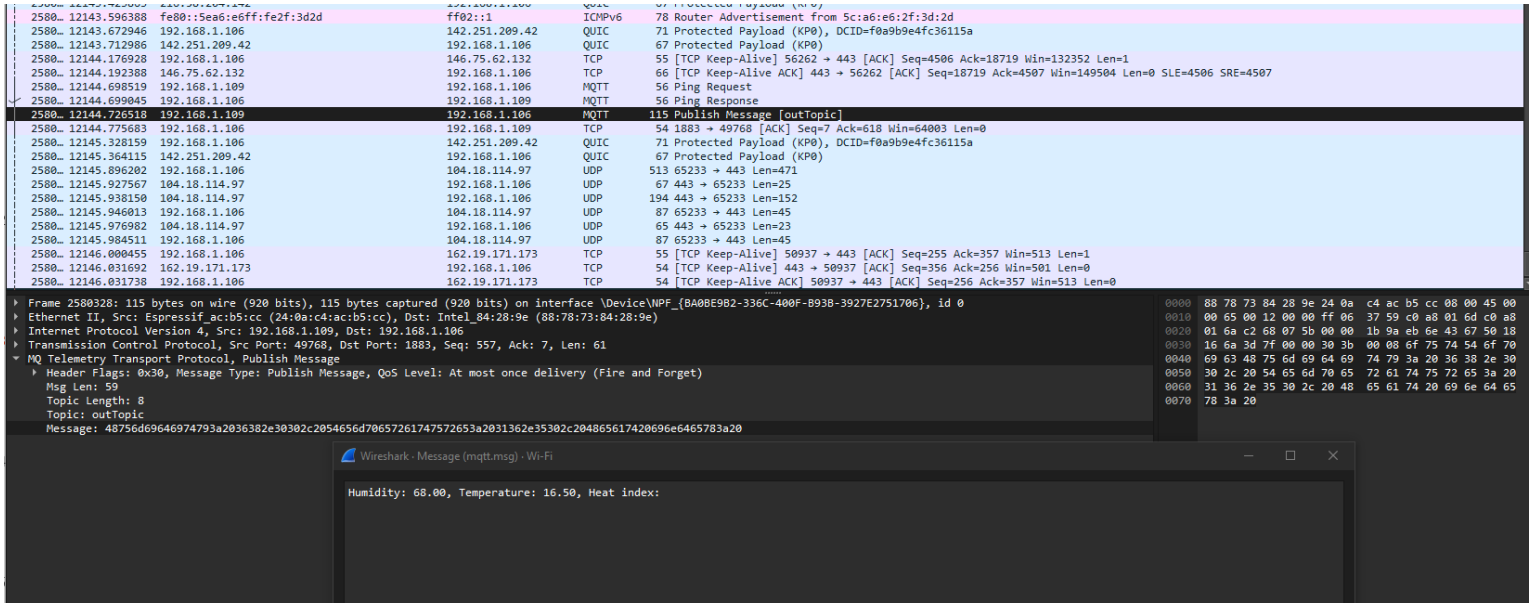
Listening ports

Protocol	Bound to	Port
amqp	0.0.0.0	5672
amqp	::	5672
clustering	::	25672
http	::	15672
mqtt	0.0.0.0	1883
mqtt	::	1883

None of the protocols manage security by default, it is needed to manually create SSL/TLS certificate before having a secure communication.

::.. Securing RabbitMQ Protocols

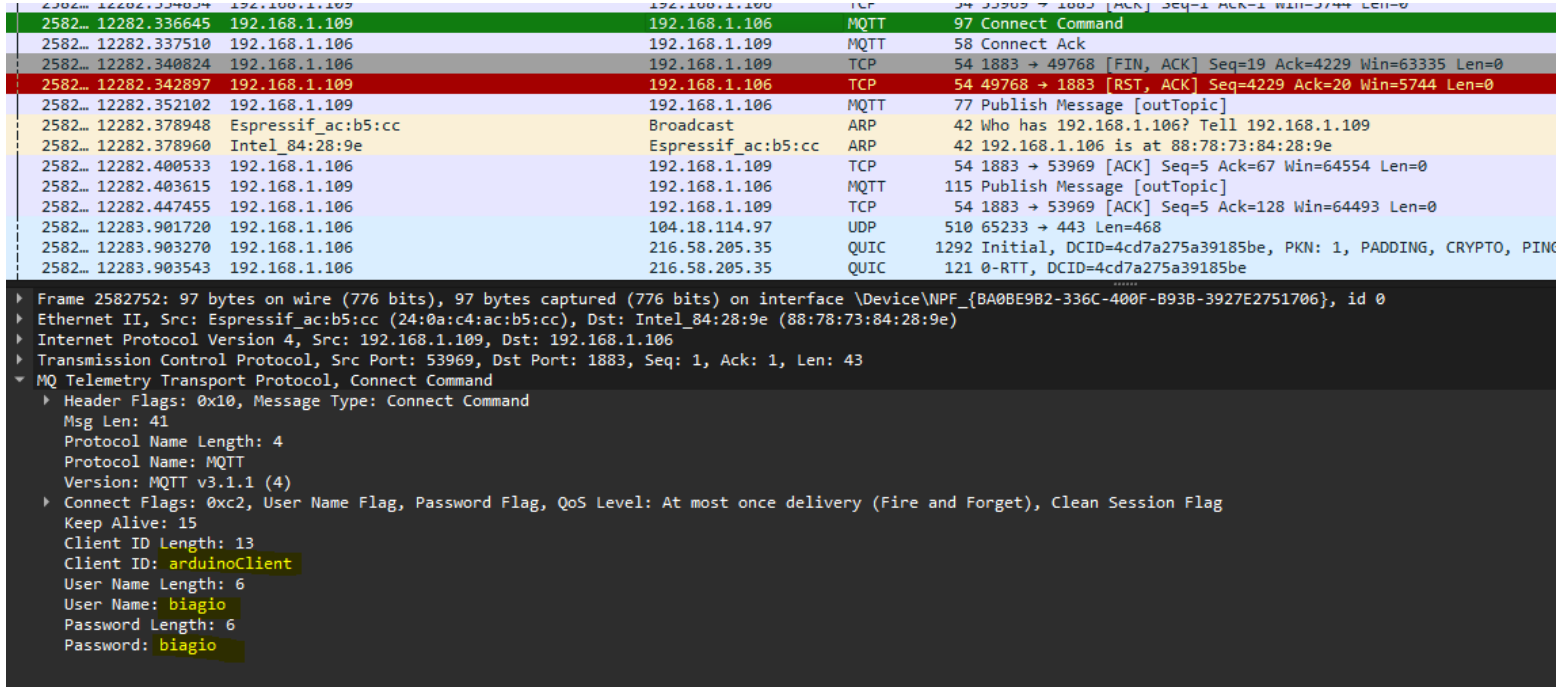
Unsecure protocols can lead to information disclosure, by applying a MITM attack, an attacker is able to sniff all the information between the device and the broker.



Wireshark is able to detect the MQTT protocol and also the topic on which the device is going to publish the information

::.. Securing RabbitMQ Protocols

The authentication mechanism employed by the MQTT Plugin does not encrypt transmitted data, making it vulnerable to eavesdropping. Moreover, the credentials used for broker connection are also susceptible to interception.

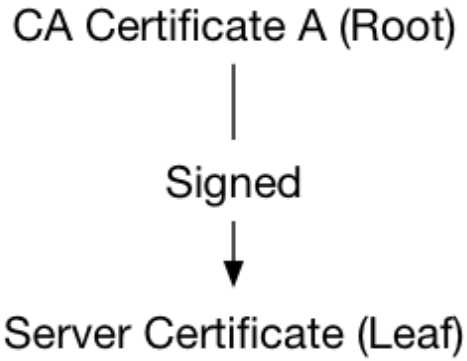


::.. Securing RabbitMQ Protocols

To secure a protocol, a SSL/TLS approach can be used, which require that a certificate must be released from a Certificate Authority (CA).

In this special case, considering that we're in academic project, a self-signed certificate can be released.

It is possible to create a new CA using OpenSSL, which offers method for creating asymmetric key pairs and certificates.



::.. Certificate Authority (CA)

The CA is responsible for creating new certificate, before to release a certificate that will be used from the Broker, it is necessary to issue a self-signed certificate.

```
# generate the private key to become a local CA
IoTSecurity@local:~$ openssl genrsa -des3 -out myCA.key 2048
# generate the root certificate
IoTSecurity@local:~$ openssl req -x509 -new -key myCA.key -days 1825 -out myCA.pem
```

The commands produce a root certificate that can be used to sign new certificate (in our case that one for the broker).

::.. Broker Certificate

Once generated the certificate for the CA, we can generate the certificate for the broker, making a request of sign to the CA, and then signing it.

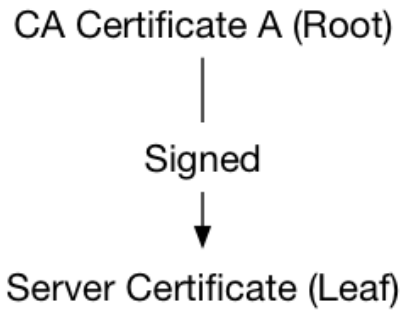
```
# generate the private key for the broker
IoTSecurity@local:~$ openssl genrsa -out broker.key 2048
# generate the sign request
IoTSecurity@local:~$ openssl req -new -key broker.key -out broker.csr -config openssl-s.cnf
# sign the request with the root CA
IoTSecurity@local:~$ openssl x509 -req -in broker.csr -CA rootCA.crt -CAkey rootCA.key
-CAcreateserial -out broker.crt -days 3650 -sha256
```

If all the steps have been done correctly, we should have the certificate in .crt format, one for the CA and one for the broker.

::.. Certificate Conversion

RabbitMQ does not accept the .crt; we need the certificates to .pem format.

```
# convert the CA certificate
IoTSecurity@local:~$ openssl x509 -in rootCA.crt -out rootCA.pem
# convert the broker certificate
IoTSecurity@local:~$ openssl x509 -in broker.crt -out broker.pem
```



At this point, 3 files are relevant for RabbitMQ:

- **rootCA.pem** -> the certificate for the CA, needed for assessing the validity of broker certificate
- **brokerCA.pem** -> the certificate for the broker
- **broker.key** -> the private key associated to the broker's certificate

::.. Securing RabbitMQ Protocols

The last step for securing the RabbitMQ protocols is to include these files in the configuration file of RabbitMQ.

```
listeners.ssl.default = 5671

ssl_options.cacertfile = /path/to/ca_certificate.pem
ssl_options.certfile   = /path/to/server_certificate.pem
ssl_options.keyfile    = /path/to/server_key.key
ssl_options.verify     = verify_peer
ssl_options.fail_if_no_peer_cert = true

mqtt.listeners.tcp.default = 1883
mqtt.listeners.ssl.default = 8883
```

We define the port on which the secure AMQP will be available, together with reference to files. Flags used:

Verify -> the certificate used by the client must be verified (mutual authentication).

Fail if no peer cert -> refuse if the certificate is not valid.

::.. Securing RabbitMQ Protocols

Now, the protocols are available in unsecure and secure version.

On the left the pre-secure available protocols and port, on the right the new ones.

Ports and contexts

Listening ports

Protocol	Bound to	Port
amqp	0.0.0.0	5672
amqp	::	5672
clustering	::	25672
http	::	15672
mqtt	0.0.0.0	1883
mqtt	::	1883

Listening ports

Protocol	Bound to	Port
amqp	0.0.0.0	5672
amqp	::	5672
amqp/ssl	0.0.0.0	5671
amqp/ssl	::	5671
clustering	::	25672
http	::	15672
mqtt	0.0.0.0	1883
mqtt	::	1883
mqtt/ssl	0.0.0.0	8883
mqtt/ssl	::	8883

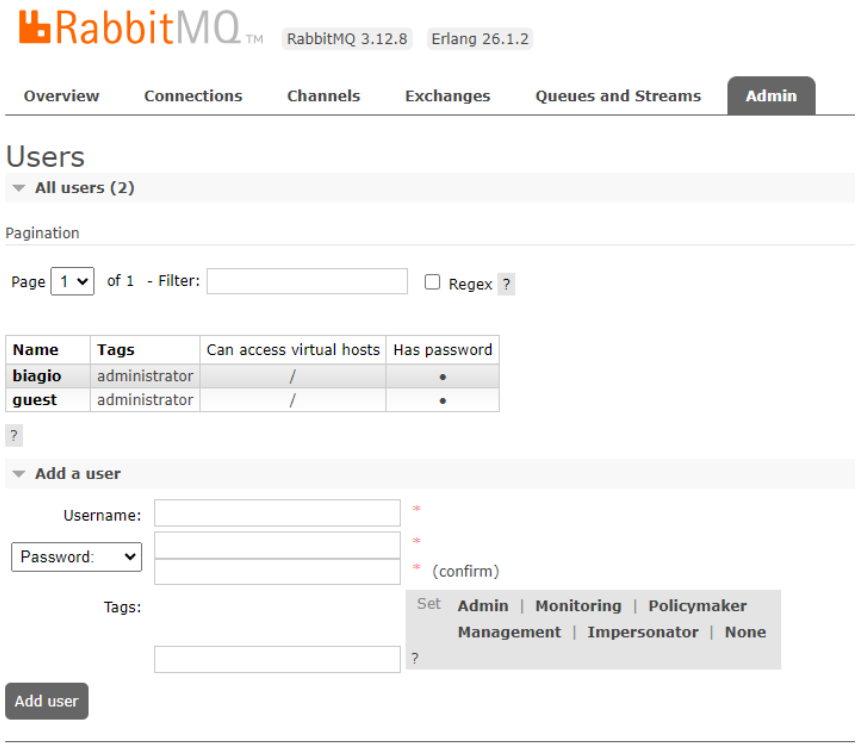
Web contexts

::.. MQTT Authentication

Once we secured the channel, the user authentication must be implemented in order to prevent malicious access.

From the management console we can add new users and select the role.

Each user will be authenticated by user/password.

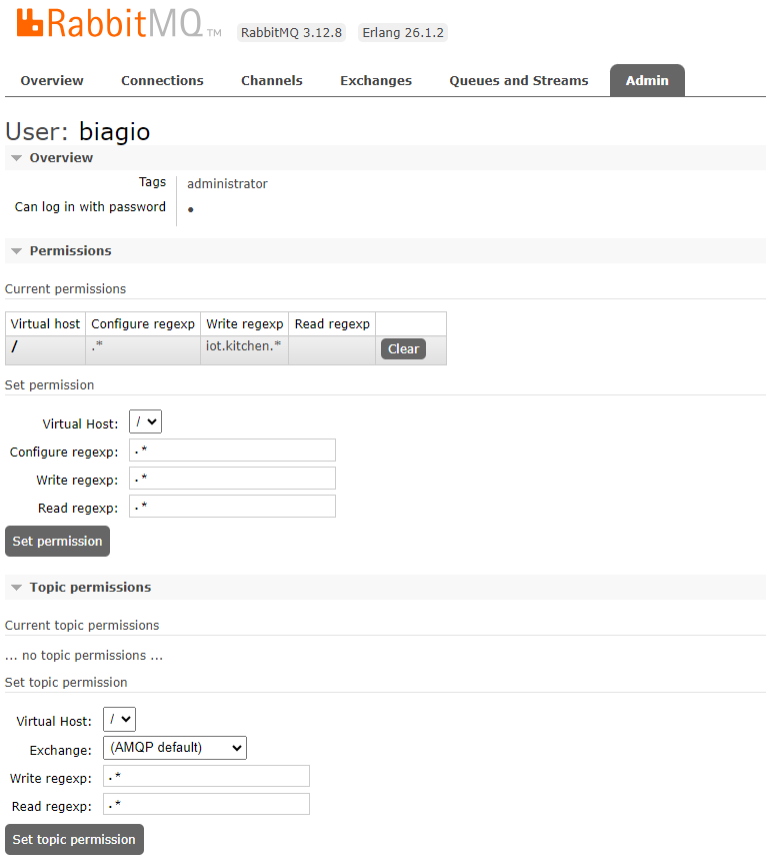


::.. MQTT Authorization

The users must be authorized to write and read-only within the topic of their interest.

Thanks to the hierarchy behind the topic syntax, it is possible to give access to all the topic starting with a regex.

This is the UI of the MQTT Authorization.



Node-RED

::.. Node-RED

Node-RED is a low-code programming tool for visual programming. It is a browser-based flow editor that makes it easy to wire together hardware devices, APIs, and online services. Node-RED is built on Node.js, so it is event-driven and non-blocking, making it ideal for IoT applications.

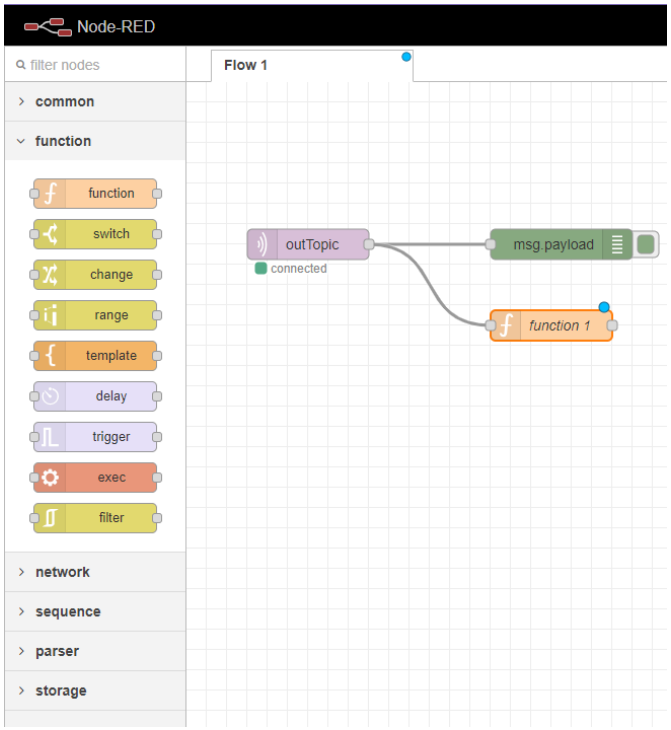
Using the drag and drop interface it is possible to connect blocks and create powerful application without writing code.

It offers multiple network connectors such as MQTT, WebSocket, HTTP.

::.. Node-RED

Functions are the most important component of Node-RED. It is possible to write functions using Javascript and execute them when needed.

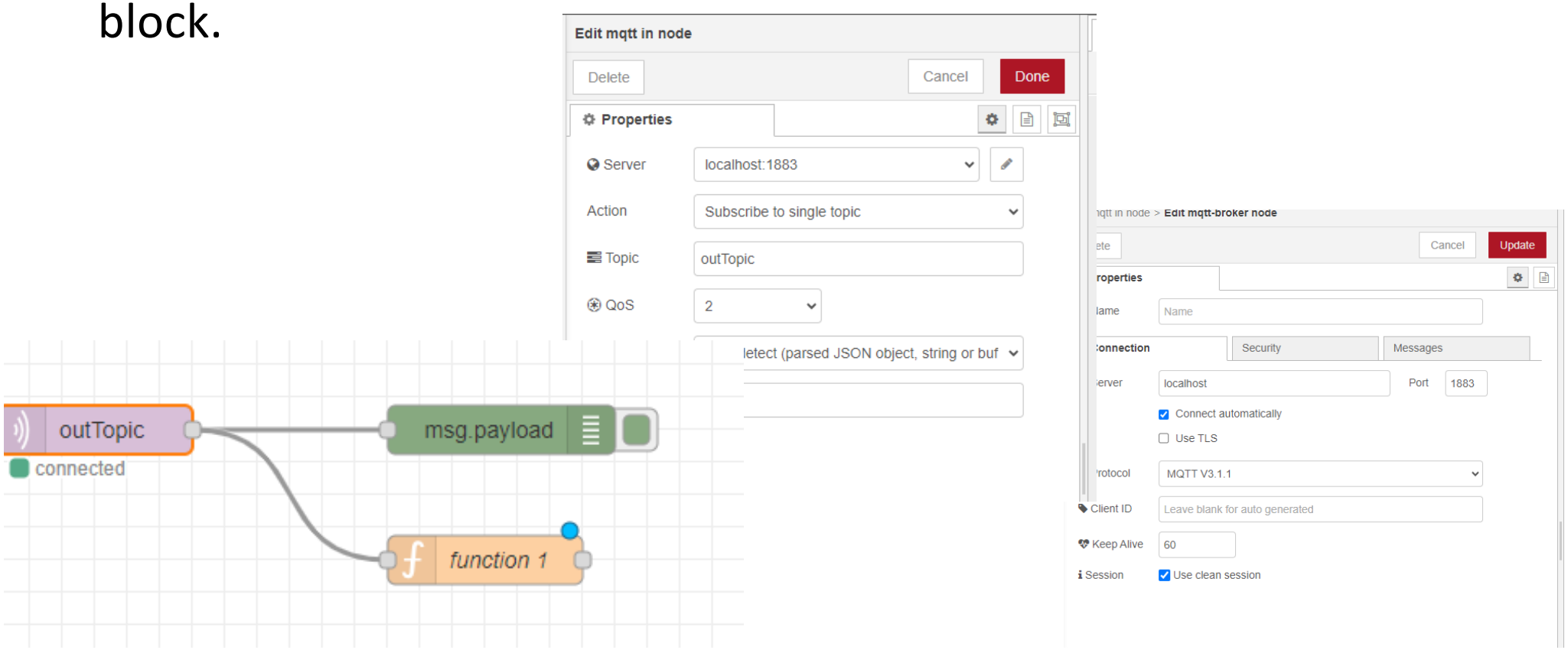
In the flow depicted here, we subsribe to a MQTT topic “outTopic”, and whenever a new message will be published, Node-RED execute a function and print the message in the debug console.



::.. Node-RED and MQTT Configuration

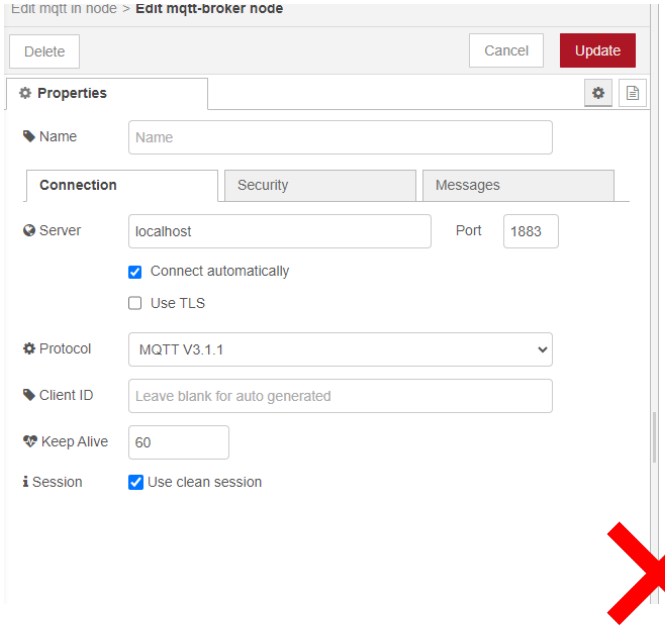
In this case, Node-RED flow is a client that connect to the server. In our case, the Node-RED is hosted on the same address of the broker.

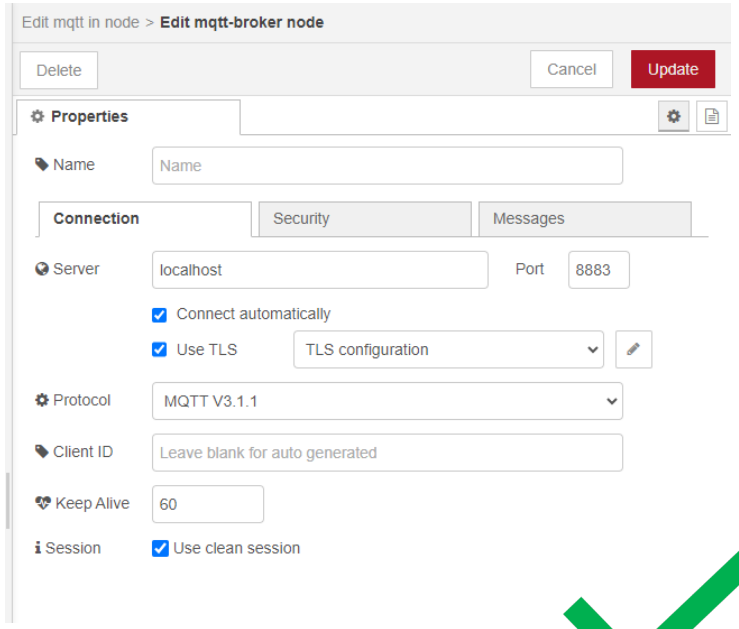
It is possible to configure the connection clicking on the MQTT block.



::.. Node-RED and MQTT Configuration

Notice that if the flow connects to the unsecure protocol (port 1883), the TLS is not requested and the traffic will be visible, so the default configuration must be changed.

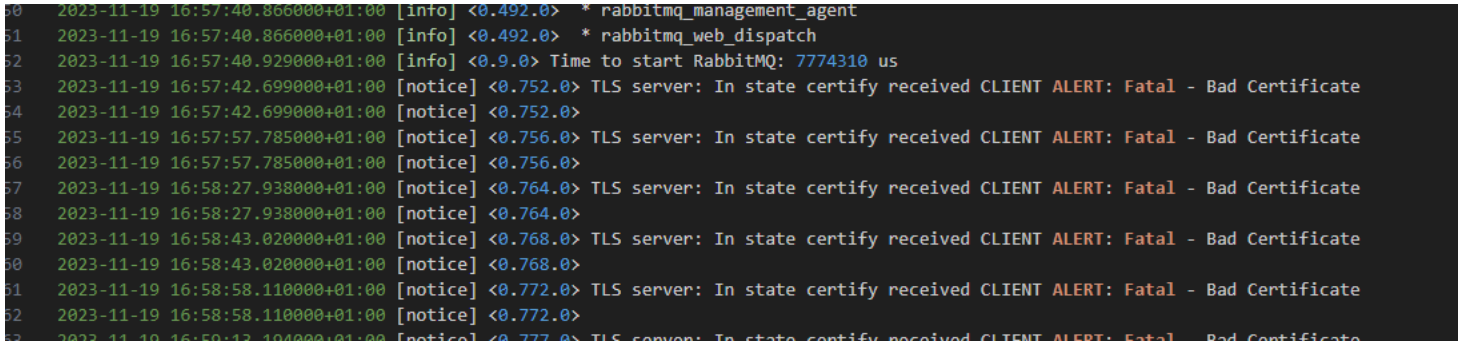




::.. Connection issue #1 Bad Certificate

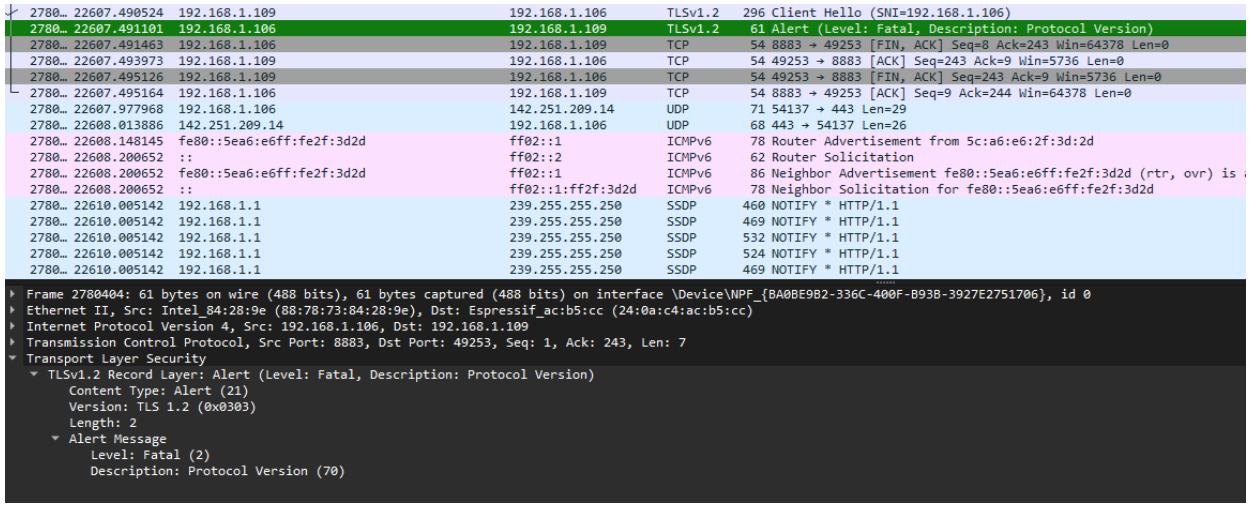
If the client does not include the CA certificate within his valid chain then the client will not be able to establish a secure connection because does not trust the server.

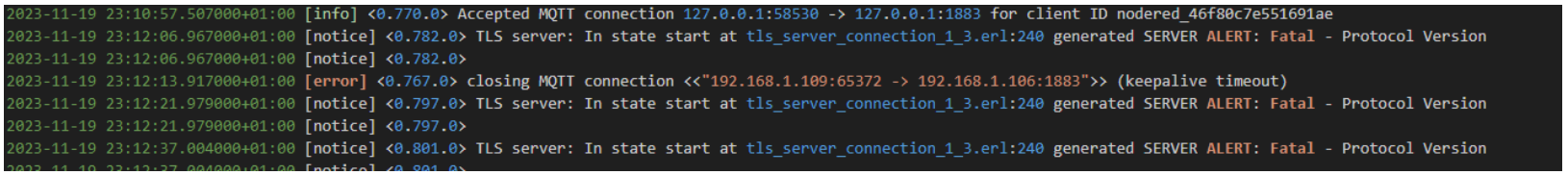
To solve this problem, the client must include the CA certificate within the chain of valid CAs.



::.. Connection issue #2 Protocol Version

If the client is using a version of TLS (v1.2) and the server only accept the new one (v1.3), then a protocol version alert will be generated.



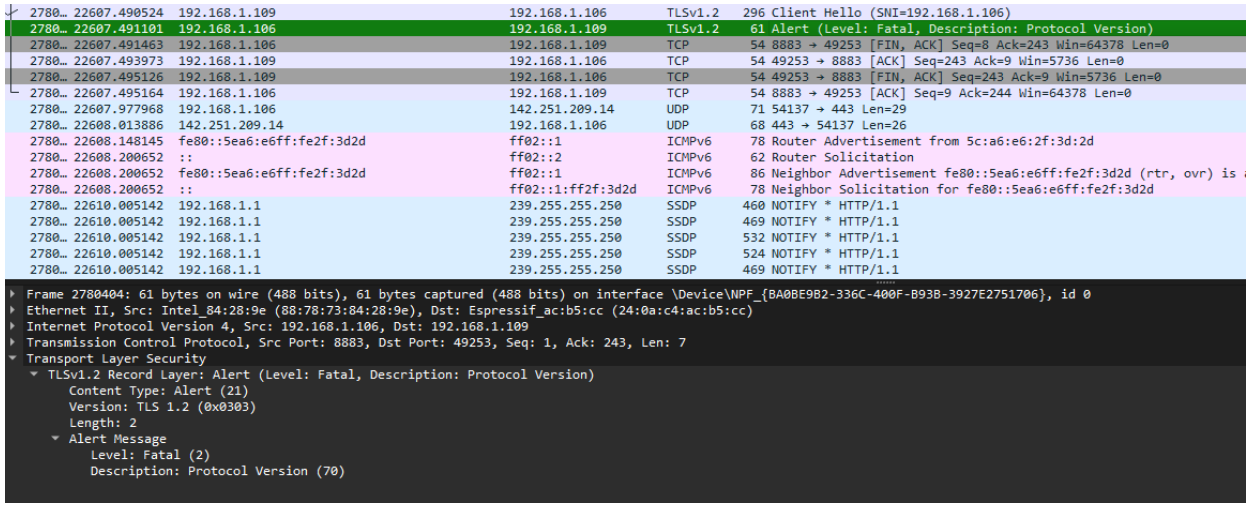


::.. Connection issue #2 Protocol Version

It is possible to solve the issue by setting the version of TLS that must be used from the broker in the .config file.

```
# ... port and certificate config

# Set the tls version to use
ssl_options.versions.1 = tlsv1.2
```




```
2023-11-19 23:10:57.507000+01:00 [info] <0.770.0> Accepted MQTT connection 127.0.0.1:58530 -> 127.0.0.1:1883 for client ID nodered_46f80c7e551691ae
2023-11-19 23:12:06.967000+01:00 [notice] <0.782.0> TLS server: In state start at tls_server_connection_1_3.erl:240 generated SERVER ALERT: Fatal - Protocol Version
2023-11-19 23:12:06.967000+01:00 [notice] <0.782.0>
2023-11-19 23:12:13.917000+01:00 [error] <0.767.0> closing MQTT connection <<"192.168.1.109:65372 -> 192.168.1.106:1883">> (keepalive timeout)
2023-11-19 23:12:21.979000+01:00 [notice] <0.797.0> TLS server: In state start at tls_server_connection_1_3.erl:240 generated SERVER ALERT: Fatal - Protocol Version
2023-11-19 23:12:21.979000+01:00 [notice] <0.797.0>
2023-11-19 23:12:37.004000+01:00 [notice] <0.801.0> TLS server: In state start at tls_server_connection_1_3.erl:240 generated SERVER ALERT: Fatal - Protocol Version
```

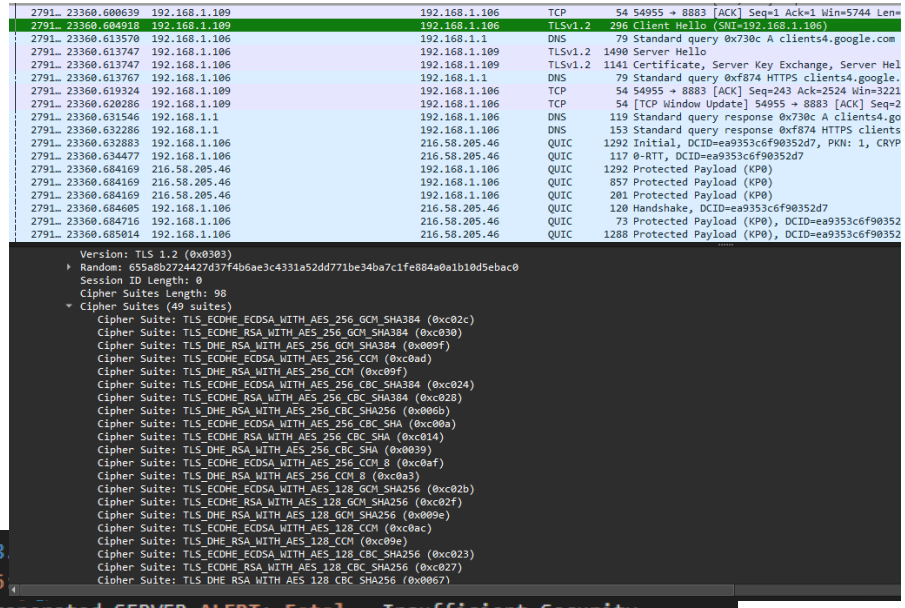
::.. Connection issue #3 Insufficient Security

If the server is not able to find a cipher for the secure connection, than the handshake fails and notify an error.

It is possible to solve the problem by specifying the supported ciphers according to those provided from the device / Node-RED by looking at the Hello message.

```
# ... port and certificate config

# Set the tls version to use
ssl_options.versions.1 = tlsv1.2
ssl_options.ciphers.1  = TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
```




```
+01:00 [info] <0.783.0> Accepted MQTT connection 192.168.1.101:60641 -> 192.168
+01:00 [info] <0.783.0> MQTT connection <<"192.168.1.101:60641 -> 192.168.1.106
+01:00 [notice] <0.891.0> TLS server: In state hello at tls_handshake.erl:354 generated SERVER ALERT: Fatal - Insufficient Security
+01:00 [notice] <0.891.0>  - no_suitable_ciphers
+01:00 [notice] <0.899.0> TLS server: In state hello at tls_handshake.erl:354 generated SERVER ALERT: Fatal - Insufficient Security
+01:00 [notice] <0.899.0>  - no_suitable_ciphers
+01:00 [notice] <0.906.0> TLS server: In state hello at tls_handshake.erl:354 generated SERVER ALERT: Fatal - Insufficient Security
+01:00 [notice] <0.906.0>  - no_suitable_ciphers
```

Arduino UNO WiFi

::.. Arduino UNO WiFi

Arduino UNO WiFi includes a library for handling WiFi connection using WiFi NINA embedded on the board.

Such library is used for connecting to an access point and checking the connection status

Connection information are typically stored on a file flashed together with the script. In this case «arduino_secrets.h»

```
#include <WiFiNINA.h>
#include "arduino_secrets.h"

char ssid[] = SECRET_SSID;        // your network SSID (name)
char pass[] = SECRET_PASS;    // your network password (use for WPA, or use as key for WEP)
int status = WL_IDLE_STATUS;     // the WiFi radio's status

void setup() {

  /* Check for the WiFi module */
  if (WiFi.status() == WL_NO_MODULE) {
    Serial.println("Communication with WiFi module failed!");
    // don't continue
    while (true);
  }

   /* Attempt to connect to WiFi network */
  while (status != WL_CONNECTED) {
    Serial.print("Attempting to connect to WPA SSID: ");
    Serial.println(ssid);
    status = WiFi.begin(ssid, pass);
    delay(5000);
  }

   IPAddress ip = WiFi.localIP();
   Serial.print("Connected with current IP Address: ");
   Serial.println(ip);
}
```

::.. Arduino UNO WiFi

To manage SSL connection it uses the WiFiSSLClient class, which is included as core library.

The object is able to establish protected connection using the SSL protocol. To verify the identity of servers, the library uses the certificate embedded into the Arduino firmware.

Since we are using a self-signed certificate as CA, it is needed to include this certificate in the chain of Arduino CAs.

This operation can be performed using Firmware and Certificate Updater. It is a tool used for update firmware and include new certificate within the board.

::.. Certificate Updater

We will not cover the firmware update, which is a good tool for recovery the firmware in case of permanent damage.

To begin with the certificate updater, we have to recall which certificate we have to flash. The one needed is the CA certificate (stored in myCA.pem).

Once the tool is ready, we can update the certificates by including the previous (Arduino and Google certificates) and the new one.

```
IoTSecurity@local:~$ arduino-fwuploader certificates flash --url arduino.cc:443,google.com:443 -f rootCA.pem -b arduino:megaavr:uno2018 -a COM3
```

::.. PubSubClient

Once we upload the certificate, we have to set up a secure connection. As said, the WiFiSSLClient automatically detects the myCA.pem certificate, as it is now embedded in the chain. We can init the PubSubClient object needed for MQTT communication.

```
#include <SPI.h>
#include <WiFiNINA.h>
#include "arduino_secrets.h"

char ssid[] = SECRET_SSID;        // your network SSID (name)
char pass[] = SECRET_PASS;    // your network password (use for WPA, or use as key for WEP)
int status = WL_IDLE_STATUS;     // the WiFi radio's status
WiFiSSLClient wifiClient;

/* MQTT import */
#include <PubSubClient.h>
void callback(char* topic, byte* payload, unsigned int length);
const char* mqtt_server = "192.168.1.106";
const int mqtt_port = 8883; //default 1833
PubSubClient client(mqtt_server , mqtt_port, callback, wifiClient);
char msg_to_send[50];

void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Attempt to connect
    if (client.connect("arduinoClient", "biagio", "biagio")) {
      Serial.println("connected");
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}
```

::.. PubSubClient

After the authentication performed using .connect()

```
client.connect(CLIENT_ID, USERNAME, PASSWORD);
```

It is possible to publish message in a topic

```
client.publish(TOPIC, MESSAGE);
```

Or to subscribe to a given topic

```
client.subscribe(TOPIC);
```

::.. PubSubClient

Whenever a message is published in a topic, it is possible to read it using a callback function.

A typical callback is to print on the serial port the message published on the topic.

```
/* Callback on msg receive */
void callback(char* topic, byte* payload, unsigned int length) {
  /* Actions to do when recieve a msg */
  char msg[length];
  for (int i = 0; i < length; i++) {
    msg[i] = (char)payload[i];
  }
  Serial.println(msg);
}
```

Now we are able to publish messages on a topic and to read from a topic in a secure way!

::.. Remarks

Once we created a secure connection, it is possible to exchange data between nodes and broker without any possibility for an attacker to perform an eavesdropping attack.

The entire flow is protected using SSL/TLS, but a strong authentication and authorization must be put in place on the RabbitMQ server, which also offer more advanced authentication methods (certificate-based auth) or oAuth.

The usage of user-centric credentials can enhance the security of the system, without the need of storing credentials on the broker server.

Arduino NANO

::.. Arduino NANO

Arduino NANO supports SSL certificates within the script. By using the SSLClient library (only compatible with a few boards - https://www.arduino.cc/reference/en/libraries/sslclient/), we can import the certificates using variables.

```
const unsigned char cert[] = \
                             "-----BEGIN CERTIFICATE-----\n" \
                             "MIIDtjCCAp4CCQDw8o5eUMLmozANBgkqhkiG9w0BAQsFADCBnDELMAkGA1UEBhMC\n" \
                             "SVQxEDAOBgNVBAgMB1NBTEVSTk8xETAPBgNVBAcMCEZJU0NJQU5PMQ4wDAYDVQQK\n" \
                             "DAVVTklOQTEgnQlK2+F4rLHCA9dPgHszJv35/WIdV43J2mbPSBiEhyfloVP7Y\n" \
                             "uU0LcCERChrWGsn5gEdtwqZtMv9KwrUUwdL/hR+VyKlFZpXJx6DfMLxS\n" \
                             "-----END CERTIFICATE-----\n";


char idNode[] = "Monitor_01";
int seq_msg = 0;

int status = WL_IDLE_STATUS;
IPAddress server(192,168,103,229);
const int port = 60000;

WiFiClient baseClient;
SSLClient client(baseClient, TAs, (size_t)2, 14);
```

::.. Arduino NANO

```
const unsigned char cert[] = \
                             "-----BEGIN CERTIFICATE-----\n" \
                             "MIIDtjCCAp4CCQDw8o5eUMLmozANBgkqhkiG9w0BAQsFADCBnDELMAkGA1UEBhMC\n" \
                             "SVQxEDAOBgNVBAgMB1NBTEVSTk8xETAPBgNVBAcMCEZJU0NJQU5PMQ4wDAYDVQQK\n" \
                             "DAVVTklOQTEgnQlK2+F4rLHCA9dPgHszJv35/WIdV43J2mbPSBiEhyfloVP7Y\n" \
                             "uU0LcCERChrWGsn5gEdtwqZtMv9KwrUUwdL/hR+VyKlFZpXJx6DfMLxS\n" \
                             "-----END CERTIFICATE-----\n";


char idNode[] = "Monitor_01";
int seq_msg = 0;

int status = WL_IDLE_STATUS;
IPAddress server(192,168,103,229);
const int port = 60000;

WiFiClient baseClient;
SSLClient client(baseClient, TAs, (size_t)2, 14);
```

The script leverages Trust Anchors (TAs) to create a chain of certificates. In this way, we can include multiple certificates coming from various CAs.

::.. Remarks

Notice that using Arduino UNO WiFi or Arduino NANO it is only possible to upload the CA certificate, meaning that it is not possible to establish a two-way authentication over TLS (mTLS).

The Arduino will always be able to check the identity of the server since it stores the CA certificate, but it will never be able to present a certificate for the mTLS.

For this reason, we have to modify the .config file removing the check on peer identity, decreasing the security of the system.

```
ssl_options.verify                = verify_none
ssl_options.fail_if_no_peer_cert = false
```

But … something better can be done …

ESP32/ESP8266

::.. ESP32/8266

Arduino Updater is only available for the Arduino boards, which means that it is not available for ESP32 or ESP8266.

These boards offer a different approach to managing certificates, they allow to manage CAs certificates directly within the script (similar to Arduino UNO), but they also allow to use of a certificate for authenticating the client, which is needed in mTLS.

The init is similar to the other boards, but in this case, we will use the **WiFiClientSecure** library.

```
#include "FS.h"
#include <ESP8266WiFi.h>;
#include <PubSubClient.h>;


/* Timing libraries */
WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP, "pool.ntp.org");

/* Define client secure and client for the connection to mqtt */
WiFiClientSecure espClient;
PubSubClient client(broker_ip, 8883, callback, espClient);
```

::.. ESP32/8266

Such library implement method for uploading CA certificates and client certificates.

```
/* Load all the certificates */
  /* Client Certificate */
  File cert = SPIFFS.open("/client_certificate.pem", "r");
  if (!cert)
    Serial.print("Failed to open cert file\n");
  else
    Serial.print("Success to open cert file\n");
  if (espClient.loadCertificate(cert))
    Serial.println("cert loaded");
  else
    Serial.println("cert not loaded");

  /* Client Key */
  File private_key = SPIFFS.open("/client_key.pem", "r"); //replace private eith your uploaded file name
  if (!private_key)
    Serial.print("Failed to open private cert file\n");
  else
    Serial.print("Success to open private cert file\n");
  if (espClient.loadPrivateKey(private_key))
    Serial.print("private key loaded\n");
  else
    Serial.print("private key not loaded\n");

  /* CA Certificate */
  File ca = SPIFFS.open("/ca_certificate.pem", "r"); //replace ca eith your uploaded file name
  if (!ca) {
    Serial.println("Failed to open ca ");
  } else
    Serial.println("Success to open ca");
  if(espClient.loadCACert(ca))
    Serial.println("ca loaded");
  else
    Serial.println("ca failed");
```

It is possible to store the certificates on the ESP memory, and then load the certificates in the client that will be used for the communication.

::.. ESP32/8266

In this way we solve the problem of mTLS, giving to the device a certificate that can be released using the same method used for the broker certificate.

The CA created in the previous step will be responsible for signing a new certificate that will be used from the IoT node for implementing mTLS with the Broker.

The same approach must be take into consideration also by the Node-RED flows, which should use a certificate for the mTLS.