Part 4: IoT Networks - Improving the Security of MQTT

Exercise 1:

1.1 What is MQTT? Briefly describe the protocol and its purpose/relation to the IoT

`MQTT` is a network lightweight protocol that transports messages between devices. Basically it runs over `TCP/IP` and it consists of 3 main components which are `broker`, `publisher` and `subscriber`. Clients can have one of two roles they can either `subscribe` and that means they can see any messages sent by a publisher to a specific topic or the client can be a publisher who can sent messages to all praticipants. Broker is the communcation medium between the clients in which it depends on the topic and it sends the published messages to all subscribers.

1.2 Set up your own IoT Network using MQTT
Answer:

We installed `mosquitto broker` and client on Ubuntu using the following commands below

```
$: sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa
$: sudo apt-get install mosquitto
$: sudo apt-get install mosquitto-clients
```

After the installation you can check if the installation is successful as shown below

```
omar@omar-VirtualBox:~$ sudo systemctl status mosquitto
[sudo] password for omar:
● mosquitto.service - Mosquitto MQTT v3.1/v3.1.1 Broker
    Loaded: loaded (/lib/systemd/system/mosquitto.service; enabled; vendor pr>
    Active: active (running) since Tue 2022-01-04 11:32:48 CET; 6 days ago
      Docs: man:mosquitto.conf(5)
            man:mosquitto(8)
  Main PID: 76906 (mosquitto)
     Tasks: 3 (limit: 4651)
    Memory: 1.1M
    CGroup: /system.slice/mosquitto.service
            └─76906 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf
```

1.3 Set up 2 MQTT Subscribers and 2 MQTT Publishers and exchange some messages via MQTT (should contain your group name as topic or payload])
Answer:

We connected 2 publishers and send publish request as shown below to `/Group4` topic with message `"Hello from Group 4"`

```
omar@omar-VirtualBox:~$ mosquitto_pub -p 1883 -h localhost -m "Hello from Group
 4" -t /Group4
```

Also we connected subscribers to the same topic to test the communication through the broker and we received the message that was sent by the publisher as shown in the screenshot below

```
omar@omar-VirtualBox:~$ mosquitto_sub -p 1883 -h localhost -t /Group4
Hello from Group 4
```

1.4 Use wireshark to inspect the sent packages and explain how the protocol works.
Answer:

We used wireshark to inspect the `MQTT packets` and check if we can see the published message in plain text or not. We were able to filter MQTT packets through wireshark filter and we found the published message as shown below

```
666 193804.88443… 127.0.0.1          127.0.0.1          MQTT     103 Connect Command
668 193804.88448… 127.0.0.1          127.0.0.1          MQTT      70 Connect Ack
670 193804.88506… 127.0.0.1          127.0.0.1          MQTT      95 Publish Message [/Gr
672 193804.88511… 127.0.0.1          127.0.0.1          MQTT      95 Publish Message [/Gr
674 193804.88522… 127.0.0.1          127.0.0.1          MQTT      68 Disconnect Req
716 193848.93060… 127.0.0.1          127.0.0.1          MQTT      68 Ping Request
718 193848.93071… 127.0.0.1          127.0.0.1          MQTT      68 Ping Response
```

Also when we opened the published packet we were able to see the content in plain text.
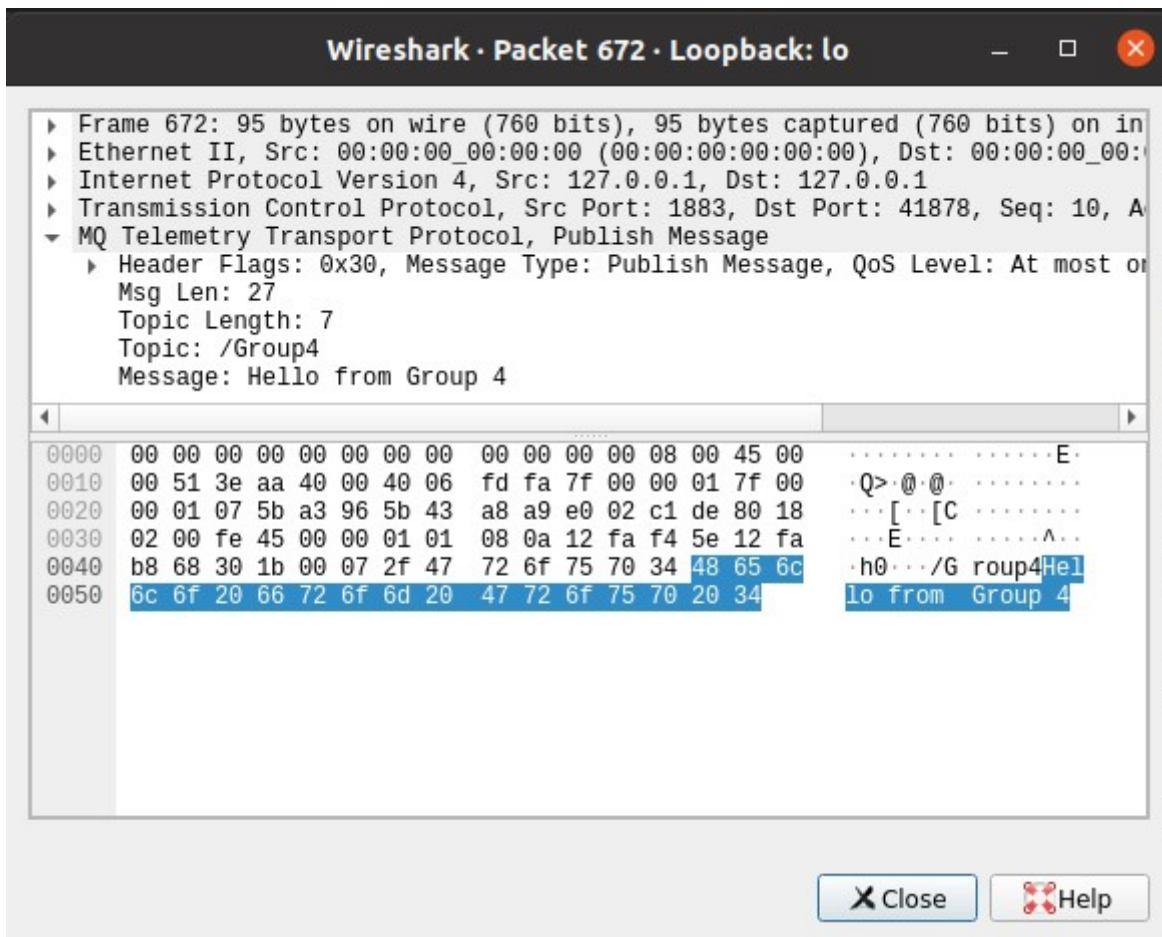
1.5 Can you spot any vulnerabilities? If so, which security goals are violated?.
Answer:

Multiple vulnerabilities can be found in the basic implementation of MQTT as all text is sent in plain text which compromise the `confidentiality` of data. Also there is no proof that connected broker is authentic nor the messages it sent violating `Integrity` property.

Exercise 2:

2.1 Enforce TLS on your MQTT Broker
Answer:

In order to enforce `TLS` we need to generate key and certificates for CA, broker and clients. We used `OpenSSL` on Ubuntu to achieve that. First step we created a directory for CA and generated key and certificate using the command below

```
$ openssl req -new -x509 -days 365 -extensions v3_ca -keyout ca.key -out ca.crt
```

Now we have a certificate for CA and we need to create keys and certificates for the broker. To create the key we use the command below

```
$ openssl genrsa -out broker.key 2048
```

After generating the key we create a signing request from the generated key.

```
$ openssl req -out broker.csr -key broker.key -new
```

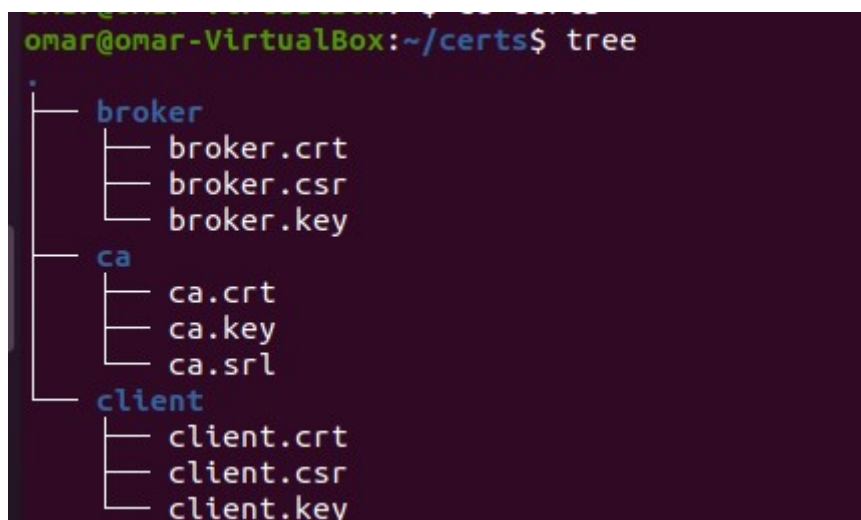We can pass the csr file we created for the broker in the previous step to the validation authority

```
$  openssl x509 -req -in broker.csr -CA ../ca/ca.crt -CAkey ../ca/ca.key -
  CAcreateserial -out broker.crt -days 100
```

This will confirm if the signature is ok or not

We will go with generating the keys and certificates for clients also using the same steps as for the broker

```
$ openssl genrsa -out client.key 2048
$ openssl req -out client.csr -key client.key -new
$ openssl x509 -req -in client.csr -CA ../ca/ca.crt -CAkey ../ca/ca.key -
  CAcreateserial -out client.crt -days 100
```

After finishing the configuration of the keys and certificates we can have the following tree as shown in the screenshot below.

Next step is we need to modify configuration of mosquitto in order to request certificates and check them and this can be done by accessing mosquitto.conf file as shown below

```
listener 8883
cafile /home/omar/certs/ca/ca.crt

#capath /home/openest/certs/ca
# Path to the PEM encoded server certificate.

certfile /home/omar/certs/broker/broker.crt

# Path to the PEM encoded keyfile.

keyfile /home/omar/certs/broker/broker.key
require_certificate true
```

The configuration recalls the `CA` and broker certificates and ask client for certificates. We will try to publish again on topic `/Group4` and check what will happen.

```
omar@omar-VirtualBox:~/certs/client$ mosquitto_pub -p 8883 --cafile ../ca/ca.cr
t --cert client.crt --key client.key -h localhost -m "Hello from Group 4" -t /G
roup4
```
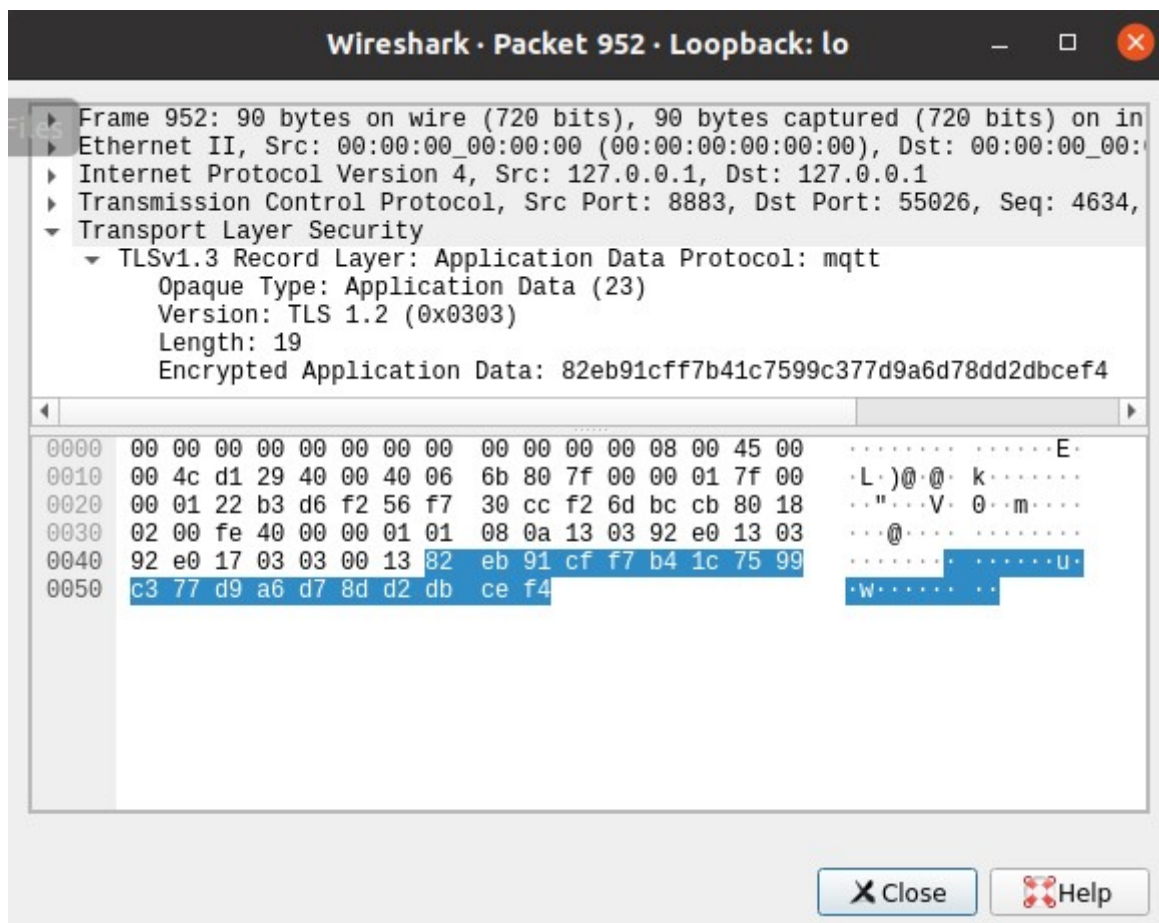
```
omar@omar-VirtualBox:~/certs/client$ mosquitto_sub -p 8883 --cafile ../ca/ca.cr
t --cert client.crt --key client.key -h localhost -t /Group4
Hello from Group 4
```

We can see that we provided the `pub` and `sub` request with client keys and certificate and we received the published message without issues and after checking wireshark we found that data is encrypted so even if we intercept the publishing packet we cannot check what are the contents of the packet which fixes the problem with confidentiality.

```
    943 449371.16577…  127.0.0.1        127.0.0.1           TCP       66 8883 → 55028 [FIN, A
    944 449371.16582…  127.0.0.1        127.0.0.1           TCP       66 55028 → 8883 [ACK] S
    945 449371.16584…  127.0.0.1        127.0.0.1           TLSv1.3   90 Application Data
    946 449371.16585…  127.0.0.1        127.0.0.1           TCP       54 8883 → 55028 [RST] S
    947 449383.18047…  127.0.0.1        127.0.0.1           MQTT      68 Ping Request
    948 449383.18058…  127.0.0.1        127.0.0.1           MQTT      68 Ping Response
    949 449383.18058…  127.0.0.1        127.0.0.1           TCP       66 41878 → 1883 [ACK] S
```

Wireshark · Packet 952 · Loopback: lo

```
▶ Frame 952: 90 bytes on wire (720 bits), 90 bytes captured (720 bits) on in
▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶ Transmission Control Protocol, Src Port: 8883, Dst Port: 55026, Seq: 4634,
▼ Transport Layer Security
   ▼ TLSv1.3 Record Layer: Application Data Protocol: mqtt
        Opaque Type: Application Data (23)
        Version: TLS 1.2 (0x0303)
        Length: 19
        Encrypted Application Data: 82eb91cff7b41c7599c377d9a6d78dd2dbcef4
```

```
0000  00 00 00 00 00 00 00 00   00 00 00 00 08 00 45 00   ············ ·····E·
0010  00 4c d1 29 40 00 40 06   6b 80 7f 00 00 01 7f 00   ·L·)@·@·  k·······
0020  00 01 22 b3 d6 f2 56 f7   30 cc f2 6d bc cb 80 18   ··"···V·  0··m····
0030  02 00 fe 40 00 00 01 01   08 0a 13 03 92 e0 13 03   ···@····  ········
0040  92 e0 17 03 03 00 13 82   eb 91 cf f7 b4 1c 75 99   ·······  ·······u·
0050  c3 77 d9 a6 d7 8d d2 db   ce f4                      ·w······ ···
```

Exercise 3:

3.1 Configure your MQTT Broker such that it allows the connections via TCP as wellas via TLS (Port 1883 and Port 8883
Answer:

We will add listener to `port:1883` to the TLS configuration so broker will be able to work on both ports at the same time to operate on the devices that are not able to support TLS also.

```
listener 1883 localhost

#port 8883


listener 8883
cafile /home/omar/certs/ca/ca.crt

#capath /home/openest/certs/ca
# Path to the PEM encoded server certificate.

certfile /home/omar/certs/broker/broker.crt

# Path to the PEM encoded keyfile.

keyfile /home/omar/certs/broker/broker.key
require_certificate true
```

3.2 Connect 2 MQTT Publishers (one via port 1883 and the other one via port 8883) and 2 MQTT Subscribers (one via port 1883 and the other one via port 8883) to the broker. All clients should publish/subscribe to the same topic. Document your observations!
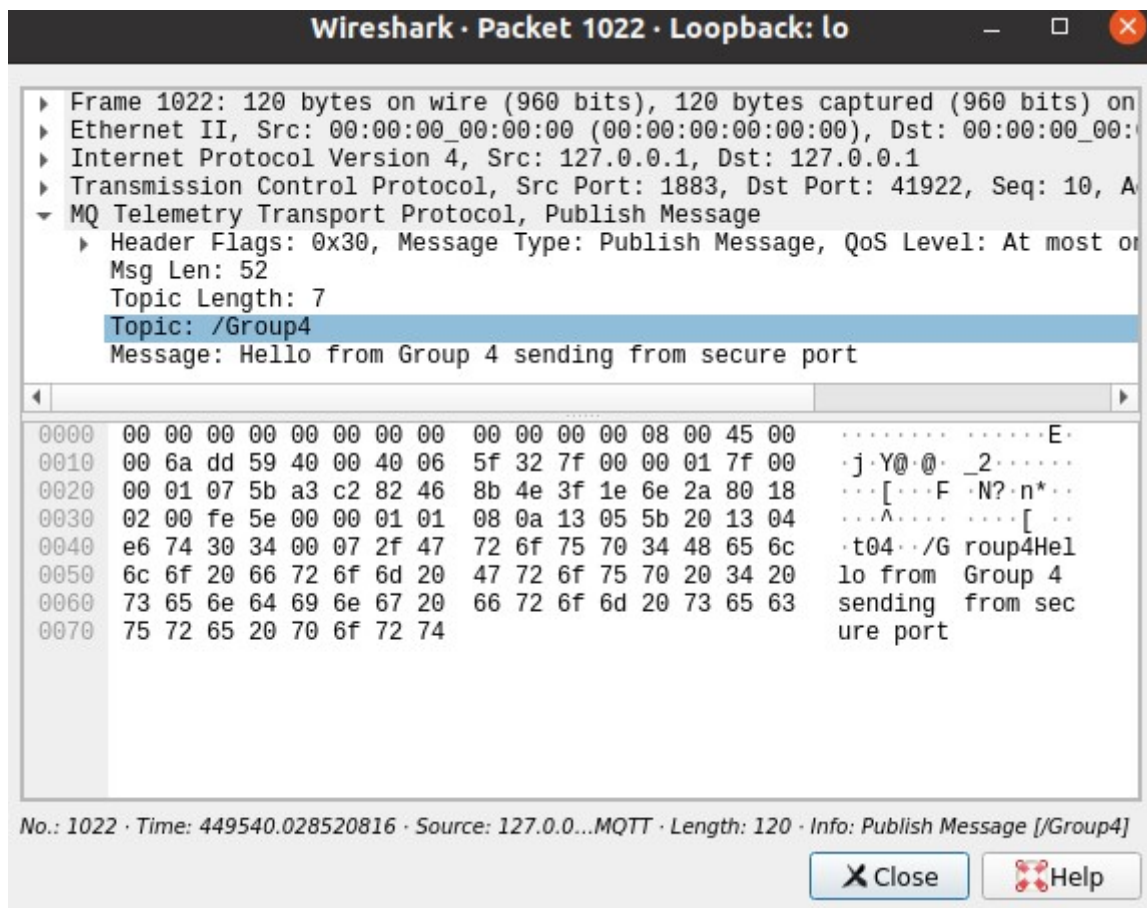
Answer:

We opened two publishers one publish on `port:8883` and the other publish on `port:1883`. At the same time we used two subscribers one on each port and all of the publishers and subscribers were on the same topic which is `/Group4`. After sending a publish request from `port:8883`, we found that we can view the messages on both subscribers as shown below although subscriber of `port:1883` has no provided certificates, so any message published can be viewed on both ports `1883` , `8883`.

```
omar@omar-VirtualBox:~/certs/client$ mosquitto_pub -p 8883 --cafile ../ca/ca.cr
t --cert client.crt --key client.key -h localhost -m "Hello from Group 4 sendin
g from secure port" -t /Group4
```

```
omar@omar-VirtualBox:~$ mosquitto_sub -p 1883 -h localhost -t /Group4
Hello from Group 4 sending from secure port
```

The screenshots below are from wireshark inspection which shows published message arrive on both ports and we can see it as plain text on `port:1883`.

No.	Time	Source	Destination	Protocol	Length	Info
1019	449540.02848…	127.0.0.1	127.0.0.1	TCP	66	41878 → 1883 [A
1020	449540.02850…	127.0.0.1	127.0.0.1	TLSv1.3	142	Application Dat
1021	449540.02851…	127.0.0.1	127.0.0.1	TCP	66	55026 → 8883 [A
1022	449540.02852…	127.0.0.1	127.0.0.1	MQTT	120	Publish Message
1023	449540.02852…	127.0.0.1	127.0.0.1	TCP	66	41922 → 1883 [A

3.3 Assume that an attacker has access to the network and is able to connect to the MQTT Broker via port 1883 (no authentication). Is this a security issue? If so, what are the possible attacks that the attacker could execute?
Answer:

If an attacker connects to `port:1883` this is a major security issue as it will compromise the confidentiality(as data transmission is in plaintext) of data and if the attacker can have the plain text and the ciphertext of the same message he can easily obtain the encryption key which can compromise the security of the whole network(although this is only possible in the local network).

- Since there is no authentication, attacker can create multiple connections, thus overaloading the broker and causing denial of service(DoS).

Exercise 4: Improving the Security of MQTT once again

We aim to solve `Confidentiality` by encrypting the message using `AES-128` and `Integrity` with digital signature.
Setting up the environment

- We used `HBMQTT` an open-source python implemention of `MQTT`.

Installation:

```
pip install hbmqtt
```

> Make sure to have python3 and above.

- Providing `Confidentiality` by encrypting the `publish` messages. For this, `AES-128` is used. This was chosen after comparing many other algorithms like `SPECK`, `SIMON`, `PRESENT`, but they tend to be weak with their key sizes, although they are ideal for low compuation intensive devices.

> Shared secret keys are assumed to be shared.

Steps:

1. Add the topic name `"Group4/verify",` to the list in `broker.py`.
2. Run the broker

```
$: python broker.py
```

```
D:\new_insider\mqtt>python broker.py
start broker
[2022-01-14 12:11:36,410] :: INFO :: transitions.core :: Finished processing state new exit callbacks.
[2022-01-14 12:11:36,410] :: INFO :: transitions.core :: Finished processing state starting enter callbacks.
[2022-01-14 12:11:36,457] :: INFO :: hbmqtt.broker :: Listener 'default' bind to localhost:9999 (max_connections=-1)
[2022-01-14 12:11:36,457] :: INFO :: transitions.core :: Finished processing state starting exit callbacks.
[2022-01-14 12:11:36,457] :: INFO :: transitions.core :: Finished processing state started enter callbacks.
[2022-01-14 12:11:36,473] :: INFO :: hbmqtt.broker :: Listener 'default': 1 connections acquired
[2022-01-14 12:11:36,473] :: INFO :: hbmqtt.broker :: Connection from ::1:65386 on listener 'default'
[2022-01-14 12:11:36,473] :: INFO :: transitions.core :: Finished processing state new exit callbacks.
[2022-01-14 12:11:36,473] :: INFO :: transitions.core :: Finished processing state connected enter callbacks.
[2022-01-14 12:11:36,488] :: INFO :: transitions.core :: Finished processing state new exit callbacks.
[2022-01-14 12:11:36,488] :: INFO :: transitions.core :: Finished processing state connected enter callbacks.
inside subscription
[2022-01-14 12:11:36,488] :: INFO :: __main__ :: Subscribed!
```

Publisher

1. Read the input message from command

```
nonce, ciphertext, tag = encrypt(input())
```

2. Encrypt the input message

```python
def encrypt(msg):
    cipher = AES.new(key, AES.MODE_EAX)
    nonce = cipher.nonce
    ciphertext, tag = cipher.encrypt_and_digest(msg.encode('ascii'))
    return nonce, ciphertext, tag
```

3. The `encrypt()` method returns `nonce`, `ciphertext`, `tag`.
   - `nonce` - for preventing replay attacks
   - `tag` - to verify the authenticity of the message.
4. These values are sent `key-value` (`dictionary`) pairs.

```
data = {'nonce': nonce, 'ciphertext':ciphertext, 'tag':tag }
```

5. Then publish the message. Make sure to convert `data` to string before sending.

```
client.publish("Group4/test", str(myDict))
```

6. Encrypted message is published to topic `Group4/test`.

```
D:\new_insider>cd mqtt

D:\new_insider\mqtt>python publisher.py
Hello
{'nonce': b'Jmb\xb3\xe7\xaa\xadEd69\xf4L\xfdoC', 'ciphertext': b'\x9ef\x90[\x85', 'tag': b'X,\xafU\xd4I\xfe\x824}I\xcdV\
xb7\xef\xe2'}
```

Subscriber

1. Subscribe to topic `Group4/test`
2. Decrypt the incoming message

> make sure to convert received message in string to dictionary class object

```
cipherObject = message.payload.decode() # received message in string
cipherObject = eval(cipherObject) # Convert back to dictionary
```

3. Decrypting the message

```python
def decrypt(nonce, ciphertext, tag):
    cipher = AES.new(key, AES.MODE_EAX, nonce=nonce)
    plaintext = cipher.decrypt(ciphertext)
    try:
        cipher.verify(tag)
        return plaintext.decode('ascii')
    except:
        return False
```

Result

```
D:\new_insider\mqtt>python subscriber.py
Connected to a broker!
Message
<paho.mqtt.client.MQTTMessage object at 0x0351EC30>
Received {'nonce': b'Jmb\xb3\xe7\xaa\xadEd69\xf4L\xfdoC', 'ciphertext': b'\x9ef\x90[\x85', 'tag': b'X,\xafU\xd4I\xfe\x82
4}I\xcdV\xb7\xef\xe2'}
Decrypting the message
Hello
```

Integrity

- It is no proof that connected device or data received is from actual broker. As broker does not offer any way to identify itslef.
- Most common approach to solve is using `digital signature`, where anyone can see the signed message and evrify with broker's public-key.
- To pass the digital signature, broker publishes it ona sepatrate topic 'Group4/verify'.

Broker

1. Add `Group4/verify` to the list of topics.
2. Sign a message (any message or something unique to broker: This message is public)

```python
msg = b'A message for signing'
hash = int.from_bytes(sha256(msg).digest(), byteorder='big')
signature = pow(hash, d, n)
print("Signature:", hex(signature))
```

where :

- **n:** Product of two random secret distinct large primes
- **(e,d):** public and private key pair_

3. Publlish the signature

```python
client.publish("Group4/verify", signature)
```

```
D:\new_insider\mqtt>python broker_pub.py
Signature: 0x72fd04109a6a7882d8610cde05b48b89ca163833f9cef54234f64d9092cafc0664d83f435458316bfd02171547f026c6f9b8bad2f4f
6bed34f17af86c7a6b67340b1362ec145ed07525ed8efd2da11296e52d3d99b37590ef564ef7c8c0b925898cd9e9638d393622f4e24162d4208eaf4c
88e2e7743ea5a53b2fca25a806d2f
```

Subscriber(s)

1. Subscribe to topic `Group4/verify`.
2. Calculate the hash for the message.

```
signature = message.payload.decode()
msg = b'A message for signing' # public

# Calculate the hash with broker's public key
hash = int.from_bytes(sha256(msg).digest(), byteorder='big')
hashFromSignature = pow(int(signature), e, n)

# Check equality
print("Signature valid:", hash == hashFromSignature)
```

3. Compare the hash with the received hash.
4. If they are equal, connected broker is legitimate.

This ensures that broker is how he says to be.

5. clients can now unsubscribe to `Group4/verify` and continue subscribing to other topics as needed.
6. In case of invalid signature. clients are automatically disconnected from the broker.

Improvement:

- Since the publisher is running all the time with its `digital signature` ,whether or not an active connection is established, this causes overhead on the broker.
- If no active connections are established to the broker, it can simply stop pusblishing its signature on topic `Group4/verify`. For this, broker need to keep track of active connections.