IOT Challenge #2

Francesco Spangaro - Luca Tosetti

20 March 2025

Academic Year 2024 - 2025

Contents

1 Introduction and setup

To solve the questions asked in the Challenge2.pdf file we analized the challenge2.pcap file using colab and tshark. We also used WireShark for a visual representation of the captured packets. In colab, we used tshark to parse the pcap file into two separate csv files, one for coap packets and one for mqtt packets. The fields we extracted are:

For coap:

- ip.src
- ip.dst
- udp.srcport
- udp.dstport
- coap.type
- coap.token
- coap.code
- coap.mid
- coap.opt.uri_path_recon

For mqtt:

- ip.src
- ip.dst
- ipv6.src
- ipv6.dst
- tcp.srcport
- tcp.dstport
- mqtt.msgtype
- mqtt.qos
- mqtt.retain
- mqtt.topic
- mqtt.willtopic
- mqtt.willmsg
- mqtt.msg

Code used to parse the pcap file into two different csv files:

```
import subprocess

fields = ["ip.src", "ip.dst", "udp.srcport", "udp.dstport", "coap.type",
       "coap.token", "coap.code", "coap.mid", "coap.opt.uri_path_recon"]


tshark_cmd = ["tshark", "-r", f"/content/challenge2.pcapng",
              "-T", "fields", "-E", "header=y", "-E", "separator=,",
              "-E", "quote=d", "-E", "occurrence=f",]
for field in fields:
    tshark_cmd.extend(["-e", field])
with open("/content/coap.csv", "w") as f:
    tshark_process = subprocess.run(tshark_cmd, stdout=f, text=True)

fields = ["ip.src", "ip.dst", "ipv6.src", "ipv6.dst",
    "tcp.srcport", "tcp.dstport",
    "mqtt.msgtype","mqtt.qos","mqtt.retain",
    "mqtt.topic", "mqtt.willtopic",
    "mqtt.willmsg", 'mqtt.msg']

tshark_cmd = ["tshark", "-r", f"/content/challenge2.pcapng",
              "-T", "fields", "-E", "header=y", "-E", "separator=,",
              "-E", "quote=d", "-E", "occurrence=f",]

# Append the fields to the tshark command
for field in fields:
    tshark_cmd.extend(["-e", field])
with open("/content/mqtt.csv", "w") as f:
    tshark_process = subprocess.run(tshark_cmd, stdout=f, text=True)
```

We then setup the dataframes for both the coap and mqtt messages, using pandas dataframes. Code:

```
import pandas as pd

# Load the CoAP data from the CSV file into a pandas DataFrame
starting_df = pd.read_csv("/content/coap.csv")
# Drop all the rows that contains only NULL values
rows_to_drop = starting_df.isna().all(axis=1)
coap_packets = starting_df[~rows_to_drop]

# Load the MQTT data from the CSV file into a pandas DataFrame
starting_df = pd.read_csv("/content/mqtt.csv")
# Drop all the rows that contains only NULL values
rows_to_drop = rows_to_drop.isna().all(axis=1)
mqtt_packets = starting_df[~rows_to_drop]

# Some packets don't have IP since is localhost IPv6 and we print IP src and dst of IPv4
```

2 Challenge Questions

2.1 CQ1

To evaluate the number of confirmable PUT requests that obtained an unsuccessful response from the local CoAP server we prepared a mask in order to properly filter packets. Firstly, we parsed all messages with ip.src and ip.dst fields set to the localhost IPv4 address. By parsing all CoAP packets with this mask we obtain only the local ones. Then, we parse them, only taking the ones that have coap.type set to 0, meaning they are confirmable requests, and coap.code set to 3, meaning that these are put requests. We then create a mask containing all packets with coap.type set to 2, meaning ack packets, and coap.code set to a number in the range [65, 69]. These are all the ack packets that indicate successful delivery of the corresponding request. To get the unsuccessful put requests we then take, from all the confirmable put requests, only the ones that don't have an ack packet in the ack set we obtained earlier. This leaves us with 22 unsuccessful confirmable put requests.
Code used for CQ1:

```
local_messages_mask = (coap_packets['ip.src'] == "127.0.0.1") &
                      (coap_packets['ip.dst'] == "127.0.0.1")
local_messages = coap_packets[local_messages_mask]

# coap.type == 0 ---> CONFIRMABLE
# coap.code == 3 ---> PUT
confirmable_put_mask = (local_messages['coap.type']== 0) &
                       (local_messages['coap.code']== 3)
confirmable_df = local_messages[confirmable_put_mask]
# coap.type == 2 ---> ACK
successful_acks_mask = (local_messages['coap.type'] == 2) &
                       ((local_messages['coap.code'] == 65) |
                           (local_messages['coap.code'] == 66) |
                           (local_messages['coap.code'] == 67) |
                           (local_messages['coap.code'] == 68) |
                           (local_messages['coap.code'] == 69))
ack_df = local_messages[successful_acks_mask]

unsuccessful_put_mask = ~confirmable_df['coap.token'].isin(ack_df['coap.token'])
unsuccessful_requests = confirmable_df[unsuccessful_put_mask]

print(f"The number of PUT requests that obtained an unsuccessful response is:
        {len(unsuccessful_requests['coap.token'].unique())}")
```

2.2 CQ2

To get the number of CoAP resources in the coap.me public server that received the same number of unique Confirmable and Non Confirmable GET requests, we created a mask parsing all packets that had as ip.src the ip address: 10.0.2.15 and as ip.dst the address: 134.102.218.18. We then grouped these packets by coap.opt.uri_path_recon. We looped on each group of packets and checked whether the number of unique confirmable requests matched the number of unique non confirmable requests. If the numbers matched and were both greater than 0, we increased a counter. Alas, the solution we obtained was 4.

Code used for CQ2:

```
public_server_messages_mask = (coap_packets['ip.src'] == "10.0.2.15") &
                              (coap_packets['ip.dst'] == "134.102.218.18")
remote_messages = coap_packets[public_server_messages_mask]

df_resource_grouped = remote_messages.groupby('coap.opt.uri_path_recon')

count = 0
for resource, group in df_resource_grouped:
    # coap.type == 0 --> CONFIRMABLE
    # coap.type == 1 --> NON CONFIRMABLE
    confirmable_df = group[group['coap.type']==0]
    non_confirmable_df = group[group['coap.type']==1]

    different_confirmable_count = confirmable_df['coap.mid'].nunique()
    different_non_confirmable_count = non_confirmable_df['coap.mid'].nunique()

    if (different_confirmable_count == different_non_confirmable_count and
            different_confirmable_count > 0):
        count += 1

print(f"The number of CoAP resources that received the same number of unique
        Confirmable and Non Confirmable GET requests is: {count}")
```

2.3 CQ3

To get the number of different MQTT clients subscribed to the public broker HiveMQ using multi-level wildcards first we had to find the public ips of the HiveMQ broker. To do so, in WireShark, we filtered for DNS packets that had HiveMQ in the payload, and that were answers. Then, by checking the payload, we found the three different ips that the public HiveMQ broker answered to:

- 35.158.43.69
- 35.158.34.213
- 18.192.151.104

We then filtered all packets that had as ip.dst one of the found ips, and that, in their subscribe request message, had a non-null topic and said topic contained '#', which is the multi-level wildcard. By counting the number of unique subscribers, we got as answer the number 4.
Code used for CQ3:

```
hivemq_messages_mask = (mqtt_packets['ip.dst']=="35.158.43.69") |
                       (mqtt_packets['ip.dst']=="35.158.34.213") |
                       (mqtt_packets['ip.dst']=="18.192.151.104")
hivemq_packets = mqtt_packets[hivemq_messages_mask]
# mqtt.msgtype == 8 (SUBSCRIBE REQUEST)
subscribe_with_multi_wildcard_mask = (mqtt_packets['mqtt.msgtype']==8) &
                                     (mqtt_packets['mqtt.topic'].notnull()) &
                                     (mqtt_packets['mqtt.topic'].str.contains('#'))
df_wildcards = hivemq_packets.loc[subscribe_with_multi_wildcard_mask]
print(f"Number of clients using multi-level wildcards:
        {df_wildcards['tcp.srcport'].nunique()}")
```

2.4 CQ4

To get the number of different MQTT clients that specify as last will topic a topic having at first level 'university', we parsed all connect messages containing a topic that starts with the word 'university', then counted the unique tcp source ports, effectively obtaining all the different clients. As a result, we obtained that a single client follows all the specified constraints.
Code used for CQ4:

```
# mqtt.msgtype == 1 (CONNECT REQUEST)
university_topic_connect_mask = (mqtt_packets['mqtt.msgtype']==1) &
                                (mqtt_packets['mqtt.willtopic'].str
                                    .startswith('university/'))
res_df = mqtt_packets[university_topic_connect_mask]

print(f"Number of different MQTT clients with last will message directed to
        topic 'university/...' is: {res_df['tcp.srcport'].nunique()}")
```

2.5 CQ5

To count the number of subscribers that received a last will message from a subscription done without a wildcard, we:

1. Parsed all messages with type publish that were received by the clients from the broker
2. Took all the willtopics and will messages specified in any connect mqtt packet
3. Parsed all the subscriptions that did not have a specified wildcard, and that matched the willtopics we found at point (2)
4. Parsed all the published messages with a topic equal to the willtopic found at point (2)
5. Joined the messages found at point (4) with the will topics and will messages found at point (2), finding all the messages that matched any will message specified by clients while connecting.

At the end of this process, we found that 3 subscribers have received a last will message derived from a subscription without a wildcard.
Code used for CQ5:

```
# ---------------------------------INFORMATION GATHERING---------------------------------
# First, we take all the messages of type PUBLISH received
# by the clients from the broker

# mqtt.msgtype == 3 (PUBLISH REQUEST)
publish_messages_from_broker_mask = (mqtt_packets['mqtt.msgtype']==3) &
                                    (mqtt_packets['tcp.srcport']==1883)
publish_messages_from_broker = mqtt_packets[publish_messages_from_broker_mask]

# Second, we take all the willtopics specified by any
# CONNECT mqtt packets in the dataset

# mqtt.msgtype == 1 (CONNECT REQUEST)
connect_messages_with_willtopic_mask = (mqtt_packets['mqtt.msgtype']==1) &
                                       (mqtt_packets['mqtt.willtopic'].notnull())
conn_messages_willtopics = mqtt_packets[connect_messages_with_willtopic_mask]

# Third, we take all and only the subscriptions to topics that are
# also willtopics, and these subscriptions must not contain any
# type of wildcards in their topic string

# mqtt.msgtype == 8 (SUBSCRIBE REQUEST)
sub_no_wildcards_to_willtopics_mask = (mqtt_packets['mqtt.msgtype']==8) &
        (~mqtt_packets['mqtt.topic'].str.contains('#|\+', na=False)) &
        (mqtt_packets['mqtt.topic']
        .isin(conn_messages_willtopics['mqtt.willtopic'].unique()))
subscribe_no_wildcards_to_willtopics =
        mqtt_packets[sub_no_wildcards_to_willtopics_mask]

# ---------------------------------INFORMATION USAGE---------------------------------
# Then we use the first two informations to extract
# all the PUBLISH messages that have a topic equal to
# any willtopic specified by a client CONNECT
# (this way we get all the publish messages about willtopics)
publish_messages_on_willtopics_mask = (publish_messages_from_broker['mqtt.topic']
        .isin(conn_messages_willtopics['mqtt.willtopic'].unique()))
publish_messages_on_willtopics =
        publish_messages_from_broker[publish_messages_on_willtopics_mask]

# Then we take the DataFrame of the PUBLISH messages on the willtopics, and the one
# containing all the willtopics and merge them on 'mqtt.willmsg' for the first one
# and 'mqtt.msg' on the second one. In this way we obtain all the PUBLISH
# messages sent to the willtopics that are ACTUALLY will messages, and not any
# message sent to the same willtopic
merged_df = pd.merge(publish_messages_on_willtopics,
        conn_messages_willtopics, left_on=['mqtt.willmsg'],
right_on=['mqtt.msg'], how='inner')

# Finally we take from the merged DataFrame all the packets sent to Clients that
# has subscribed to the willtopics with no wildcards
subscribers_willmessage_receivers_ports = merged_df.loc[merged_df['tcp.dstport_x']
        .isin(subscribe_no_wildcards_to_willtopics['tcp.srcport'].unique()),
        'tcp.dstport_x']
subscribers_willmessage_receivers =
        publish_messages_on_willtopics[publish_messages_on_willtopics['tcp.dstport']
            .isin(subscribers_willmessage_receivers_ports)]
```

```
print(f"Number of MQTT subscribers that has received a
        last will message derived from a subscription without a wildcard is:
        {subscribers_willmessage_receivers['tcp.dstport'].nunique()}")
```

2.6 CQ6

To count the number of MQTT publish messages directed to the public broker mosquitto that were sent with the retain flag to one and set the QoS to "At most once", we parsed the mqtt packets with a mask that checks that:

- The ip.dst is set to the ip of the public broker mosquitto: "5.196.78.28"
- The mqtt.msgtype is set to 3, marking a publish type message
- The mqtt.retain flag is set to 1
- The mqtt.qos flag is set to 0

As a result, we obtained 208 messages.
Code used for CQ6:

```
# mqtt.msgtype == 3 (PUBLISH)
publish_mosquitto_packets_mask = (mqtt_packets['ip.dst']=="5.196.78.28") &
                                 (mqtt_packets['mqtt.msgtype']==3)
publish_mosquitto_packets = mqtt_packets[publish_mosquitto_packets_mask]

# QoS == 0 (At most once)
retained_qos0_mosquitto_packets_mask = (publish_mosquitto_packets['mqtt.retain']==1) &
                                       (publish_mosquitto_packets['mqtt.qos']==0)
res_df = publish_mosquitto_packets[retained_qos0_mosquitto_packets_mask]

print(f"Number of MQTT publish messages directed to the public broker mosquitto
        are sent with the retain option and use QoS 'At most once' is: {len(res_df)}")
```

2.7 CQ7

In order to count the number of MQTT-SN messages on port 1885 sent by the clients to a broker in the local machine, we changed the MQTT-SN protocol in WireShark in order to properly filter the messages sent to the port. We then filtered all packets with protocol MQTT-SN. Since we found 0 records, we can confidently say that the number of messages we are looking for is 0.