

# **IoT Challenge #2**

## Packet Analysis

Giuliano Crescimbeni, 10712403 - Arimondo Scrivano, 10712429  
Politecnico di Milano

March 2025

# Contents

<b>1</b>	<b>Methodology</b>	<b>3</b>
<b>2</b>	<b>CQ1: Unsuccessful PUT Requests</b>	<b>3</b>
2.1	Result . . . . .	4
<b>3</b>	<b>CQ2: Equal Unique GET Requests on coap.me</b>	<b>4</b>
3.1	Result . . . . .	5
<b>4</b>	<b>CQ3: MQTT Clients Using Multi-Level Wildcard Subscriptions</b>	<b>5</b>
4.1	Result . . . . .	6
<b>5</b>	<b>CQ4: Clients Specifying a Last Will Topic Starting with “university/”</b>	<b>6</b>
5.1	Result . . . . .	7
<b>6</b>	<b>CQ5: MQTT Subscribers Receiving a Last Will Message from Subscriptions Without Wildcards</b>	<b>7</b>
6.1	Result . . . . .	8
<b>7</b>	<b>CQ6: MQTT Publish Messages to Mosquitto with QoS 0 and Retain Set</b>	<b>8</b>
7.1	Result . . . . .	9
<b>8</b>	<b>CQ7: MQTT-SN Messages Sent to Local Broker on Port 1885</b>	<b>10</b>
8.1	Result . . . . .	10

# 1 Methodology

To analyze the PCAP file provided in the challenge, we used **Wireshark** as the primary tool. Specific display filters were applied to identify relevant packets. When required, we used Python scripts with **pyshark** to automate counting operations or confirm observations.

## 2 CQ1: Unsuccessful PUT Requests

To answer this question, we analyzed the PCAP file using a custom Python script. The script iterates over all CoAP packets and identifies Confirmable PUT requests (type = 0 and code = 3). For each such request, it attempts to find a matching response based on the CoAP token and checks whether the response indicates an error.

Here is the core of the logic implemented:

```
1 # Iterate over all CoAP packets in the capture
2 for packet in cap:
3     try:
4         # Check if the packet is a Confirmable PUT
5         # request
6         if packet.coap.type == '0' and packet.coap.code
7         == '3':
8             # Find the corresponding response using the
9             # same token
10            response = find_response(cap, packet.coap.
11            token)
12
13            # Check if the response is invalid (error
14            # codes)
15            if response and is_invalid_response(response
16            ):
17                invalid_responses_count += 1
18    except AttributeError:
19        # Ignore packets without the required CoAP
20        # fields
21        continue
```

Listing 1: Python code used to identify unsuccessful Confirmable PUT requests

## 2.1 Result

The total number of unique Confirmable PUT requests that received an unsuccessful response is: **22**.

## 3 CQ2: Equal Unique GET Requests on coap.me

To address this question, we wrote a Python script to iterate over all CoAP packets and analyze GET requests directed to the public server `coap.me`. The IP address of `coap.me` (134.102.218.18) was identified by inspecting the DNS request/response packets within the PCAP file.

The script filters GET requests (method code 1), distinguishes between Confirmable (CON) and Non-Confirmable (NON) messages, and counts how many unique resources received the same number of both types of requests. The logic is summarized in the following code:

```
1 for packet in cap:
2     try:
3         # Check if the packet is directed to or from
4         # coap.me
5         if packet.ip.dst == "134.102.218.18":
6             # Check if it's a GET request
7             if packet.coap.code == '1': # Code for GET
8                 resource = packet.coap.opt_uri_path
9
10                if resource not in resource_count:
11                    resource_count[resource] = {'CON': 0, 'NON': 0}
12
13                # Increment count based on message type
14                if packet.coap.type == '0': # Confirmable
15                    resource_count[resource]['CON'] += 1
16                elif packet.coap.type == '1': # Non-Confirmable
17                    resource_count[resource]['NON'] += 1
18            except AttributeError:
19                continue
20
21 # Count how many resources have equal counts of CON and
22 # NON
```

```

21 equal_count = sum(1 for counts in resource_count.values
    () if counts['CON'] == counts['NON'])

```

Listing 2: Python code to count resources with equal CON and NON GET requests

### 3.1 Result

The number of CoAP resources on `coap.me` that received the same number of unique Confirmable and Non-Confirmable GET requests is: **3**.

## 4 CQ3: MQTT Clients Using Multi-Level Wildcard Subscriptions

To determine how many different MQTT clients subscribed using multi-level wildcards (“#”), we used a Python script to analyze the PCAP file. The script inspects MQTT SUBSCRIBE packets (message type = 8), checks if the topic contains the multi-level wildcard “#”, and identifies unique clients by their source IP and source port combination.

We only considered packets directed to known public HiveMQ broker IPs, which were identified by analyzing DNS request/response packets:

- 18.192.151.104    • 35.158.34.213    • 35.158.43.69

The following Python code was used:

```

1 for packet in cap:
2     try:
3         # Check if it's an MQTT SUBSCRIBE packet
4         if packet.mqtt.msgtype == '8' and (packet.ip.dst
            == "18.192.151.104" or ...): # Message
            type 8 is SUBSCRIBE that goes to HiveMQ
            broker
5             topic = getattr(packet.mqtt, "topic", "")
6
7             # Extract source IP and TCP port
8             src_ip = packet.ip.src
9             src_port = packet.tcp.srcport
10
11            # Build a unique client identifier
12            client_identifier = f"{src_ip}:{src_port}"

```

```

13
14         # Check if the topic contains multi-level
           wildcard '#' and goes to a HiveMQ broker
15         if '#' in topic:
16             unique_clients.add(client_identifier)
17     except AttributeError:
18         # Ignore packets missing MQTT fields
19         continue

```

Listing 3: Python code to count unique clients using wildcard #

## 4.1 Result

The total number of distinct MQTT clients using multi-level wildcard subscriptions to the HiveMQ public broker is: 4.

## 5 CQ4: Clients Specifying a Last Will Topic Starting with “university/”

To solve this question, we scanned all MQTT CONNECT packets (message type = 1) and checked for the presence of a Last Will Topic field (`willtopic`). When such a topic was found, it was extracted and printed for manual inspection.

The following Python code was used to identify the Last Will Topics:

```

1 for packet in cap:
2     try:
3         # Check if the packet is a CONNECT packet
4         if packet.mqtt.msgtype == '1': # CONNECT
5             # Check if a Last Will Topic is defined
6             if hasattr(packet.mqtt, 'willtopic'):
7                 topic = str(packet.mqtt.willtopic)
8                 print(topic)
9     except AttributeError:
10        continue # Skip non-MQTT or incomplete packets

```

Listing 4: Python code to extract Last Will Topics from CONNECT packets

## 5.1 Result

The script printed the following 4 Last Will Topics:

- [university/department12/room1/temperature]
- [metaverse/room2/floor4]
- [hospital/facility3/area3]
- [metaverse/room2/room2]

As can be seen from the topics listed above, only one of them begins with the prefix **university/**. Therefore, the total number of MQTT clients specifying a Last Will Topic with **university** as the first level is: **1**.

## 6 CQ5: MQTT Subscribers Receiving a Last Will Message from Subscriptions Without Wildcards

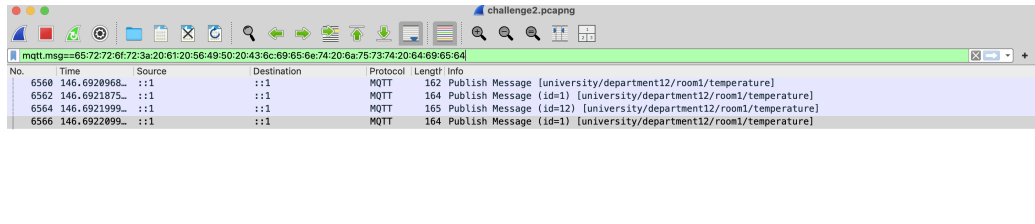
For this question, we considered the four Last Will Topics identified previously (see CQ4). For each of these topics, we extracted the corresponding Last Will Message payloads from the CONNECT packets. The payloads are listed below:

- Topic [university/department12/room1/temperature] →  
→ Payload: [65:72:72:6f:72:3a:20:61:20:56:49:50:20:43...]
- Topic [metaverse/room2/floor4]  
→ Payload: [65:72:72:6f:72:3a:20:70:65:75:69:76:64:71:6c]
- Topic [hospital/facility3/area3]  
→ Payload: [65:72:72:6f:72:3a:20:78:6c:7a:61:71:70:74:6]
- Topic [metaverse/room2/room2]  
→ Payload: [65:72:72:6f:72:3a:20:7a:6a:7a:77:72:63:64:70]

We used Wireshark display filters to search for these payloads in all MQTT PUBLISH packets. Among them, only the topic [TOPIC.X] showed multiple matching messages, confirming that the Last Will was actually published. A screenshot of the corresponding Wireshark output: (Figure 1).

For each received message, we identified the subscriber client by IP and port, and traced back its SUBSCRIBE packet to analyze whether a wildcard

was used in the topic filter. Below is the list of clients and the presence of wildcard in their subscriptions:



No.	Time	Source	Destination	Protocol	Length	Info
6560	146.6920968...	:::1	:::1	MQTT	162	Publish Message [university/department12/room1/temperature]
6562	146.6921875...	:::1	:::1	MQTT	164	Publish Message (id=1) [university/department12/room1/temperature]
6564	146.6921999...	:::1	:::1	MQTT	165	Publish Message (id=12) [university/department12/room1/temperature]
6566	146.6922099...	:::1	:::1	MQTT	164	Publish Message (id=1) [university/department12/room1/temperature]

Figure 1: Wireshark output showing Last Will message published on topic [university/department12/room1/temperature]

- Dst Port: 39551 ✓ No wildcard
- Dst Port: 53557 ✓ No wildcard
- Dst Port: 51743 ✗ Contains wildcard
- Dst Port: 41789 ✓ No wildcard

## 6.1 Result

As shown above, three clients received the Last Will message through subscriptions that did not use any wildcard. Therefore, the final answer is: **3**.

## 7 CQ6: MQTT Publish Messages to Mosquitto with QoS 0 and Retain Set

To answer this question, we used Wireshark to filter all MQTT PUBLISH messages that satisfy the following conditions:

- The message type is PUBLISH (`mqtt.msgtype == 3`)
- The message is sent to the Mosquitto broker (`ip.dst == 5.196.78.28`)
- QoS level is 0 (`mqtt.qos == 0`)
- The retain flag is set (`mqtt.retain == 1`)

The following Wireshark display filter was applied:



```
1 ((mqtt) && (mqtt.msgtype == 3) && (ip.dst ==
    5.196.78.28) && (mqtt.qos == 0) && (mqtt.retain ==
    True))
```

Listing 5: Wireshark display filter used

## 7.1 Result

Upon applying this filter, Wireshark returned a total of **208** packets matching the specified criteria.

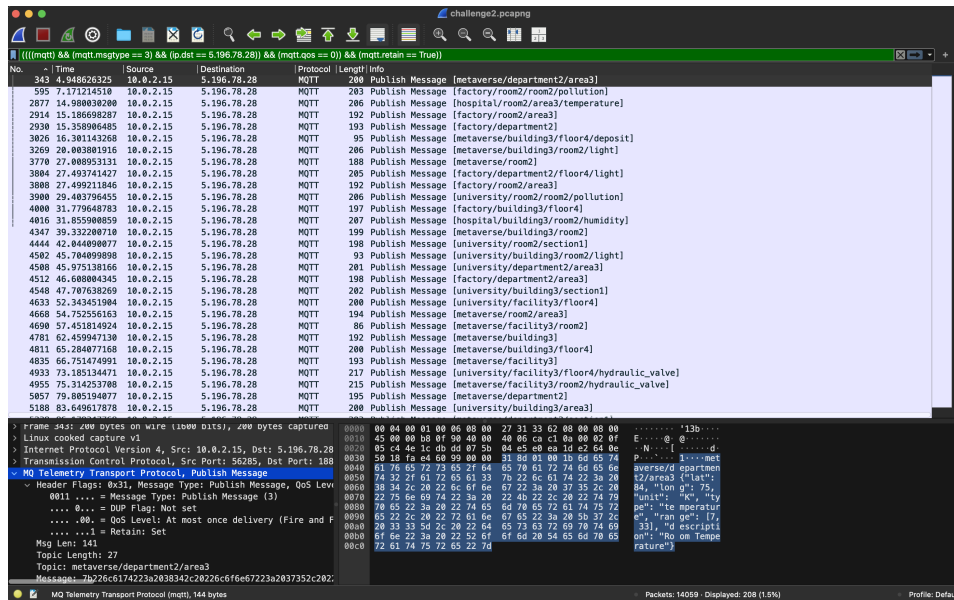


Figure 2: Filtered MQTT PUBLISH messages to Mosquitto broker (QoS 0, retain = true)

## 8 CQ7: MQTT-SN Messages Sent to Local Broker on Port 1885

During the analysis of the captured traffic, we specifically searched for any packets using the MQTT-SN (MQTT for Sensor Networks) protocol, particularly those sent to the local broker on port 1885. However, no such packets were found in the dataset.

Applying this filter doesn't return any result:

```
1 udp.port = 1885
```

### 8.1 Result

Therefore, the answer is: **0**.