Elias Borge-Ilseth, 11075526
Eskil Myklatun Østbø, 11074713 04/04/2025

Internet of Things Challenge 2

Part 1 – Questions on the PCAP file

All questions in Part 1 have been answered using the attached Python program, *part1.py*. We utilize PyShark for packet analysis and applied Wireshark-compatible display filters. The code is commented as well as described in this report.

The code includes separate functions solving each Challenge Question, along with additional helper methods for retrieving DNS responses for hostnames and obtaining client identifiers (IP address and port) for packet senders and receivers

CQ1

Answer: 22

The script uses PyShark to open the .pcapng file and filter for CoAP packets. The, we find all Confirmable PUT requests:

```
coap.type == 0 and coap.code == 3
```

and store all the unique tokens in a list to obtain the list of all unique Confirmable PUT Requests. The script then looks for responses to all unique Confirmable PUT Requests with a CoAP status code ≥ 128 (indicating error responses starting with 4.xx or 5.xx) from the local server:

```
coap.code>=128 and ip.src == 127.0.0.1
```

Finally, it counts all the unsuccessful response of Confirmable PUT requests and prints the result.

CQ2

Answer: 3

We begin by filtering for CoAP GET requests directed to the IP address of the public server, which yields a list of all requests to the various resources on the server:

```
coap and coap.code == 1 and ip.addr == 134.102.218.18
```

For each packet, we then track the requested resource and maintain two sets containing all tokens for both confirmable and non-confirmable requests to that resource. These sets represent the unique requests querying each resource.

Finally, we filter out any resource where the number of confirmable and non-confirmable requests differs and count the number of resources where they are equal.

CQ3

Answer: 4

We begin by filtering for all MQTT subscribe requests that contains a multi-level wildcard and is sent to the public broker of HiveMQ:

```
mqtt and mqtt.msgtype == 8 and mqtt.topic contains '#'

and ip.dst in {18.192.151.104, 35.158.43.69, 35.158.34.213}
```

We obtain the correct ip addresses with the *extract_dns_ips* method.

This filter correctly identifies all clients subscribing to the broker using multi-level wildcards. However, it may not only return unique clients, as a single client can subscribe to multiple topics. Therefore, we must filter out duplicate entries that originate from the same client.

CQ4

Answer: 1

We begin by filtering for all MQTT request that specify a will topic that begins with “university”:

```
mqtt and mqtt.willtopic[:10] == university
```

Similar to CQ3, this filter correctly identifies all valid connect commands. While it could potentially return multiple instances from the same client, there is only one request in this case. To maintain consistency, we still filter out duplicate clients.

CQ5

Answer: 3

In this question, we opted not to create a comprehensive display filter because we needed to collect and correlate three distinct types of packets: connect commands, subscribe requests, and publish messages. For each packet type, we extracted the relevant fields and stored them in separate lists if they met the specified requirements. Connect commands were required to have a predefined Last Will topic and message. Subscribe requests were only considered if they did not contain wildcard topics. All Publish messages were stored.

After collecting all packets, we joined them together to identify sent messages with a topic and message that matched a Last Will, ensuring that the recipient had subscribed without using wildcards. Finally, we counted the number of such messages.

CQ6

Answer: 208

This question can be solved with a filter, where we look for MQTT public messages that are directed to the mosquitto broker with Quality of service set to 0 and Retain set to true:

```
mqtt and mqtt.msgtype == 3 and mqtt.qos == 0 and mqtt.retain == True

and ( ip.dst in {5.196.78.28} or ipv6.dst in {2001:41d0:a:6f1c::1} )
```

The ip addresses is identified from the DNS responses in the file. We then simply count the number of displayed packets.

CQ7

Answer: 0

In this task we found no packets. We started with trying to filter on MQTTS, which return no results. Then we also tried to find any packets that that are sent to the port 1885:

```
udp.port == 1885 or tcp.port == 1885
```

This also returns no packets. We consider the task done and will not contruct a more comprehensive filter.