IoT hacking toolbox

Márton Kerekes

Project Laboratory report

Abstract

Nowadays, Internet of Things (IoT) devices are getting more and more use cases, such as smart homes, hospital sensors, vehicles like self-driving cars. As a result, the number of these devices is rapidly growing. Cyberattacks against these IoT systems can cause large economical loss or, in the worst-case scenario, life-threatening situations. A common type of attack is the Man-in-the-Middle (MITM) attack, which is greatly overlooked in IoT networks. The communication within an IoT network uses various kinds of protocols for transporting data. Since IoT devices are usually small with limited resources, the protocol needs to be simple and use small bandwidth. In this paper, we will see how Message Queuing Telemetry Transport (MQTT) -- one of these IoT protocols does -- work, how one can set up an environment for it, and we also describe a new MITM tool that can be used against the MQTT protocol.

Introduction

Due to the rapidly growing popularity of Internet-of-Things (IoT), almost everything can be considered an IoT device, like our lamp, fridge, watch, shoes. So, it is very important to secure these devices.

Let us see one of these protocols that IoT uses: the Message Queuing Telemetry Transport (MQTT). It was developed by Andy Stanford-Clark and Arlen Nipper in 1999. Since then, it has become one of the most used protocols in the IoT world. It is also used widely in the industry. This is a lightweight protocol to make sure that every device can use it, like sensors or other small devices, to send data to a central server or into the cloud. Also, the bandwidth requirement of MQTT is really low.

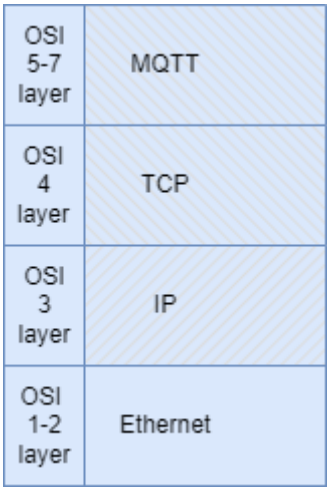

Figure 1: MQTT on top of TCP in the OSI layers

The MQTT protocol uses a basic publish-subscribe model. Since it works in the application layer and it resides on top of the TCP/IP protocols (see Figure 1), it can be considered a

somewhat reliable protocol. Its core aspect is the low bandwidth usage with low resources and in addition reliability thanks to TCP.

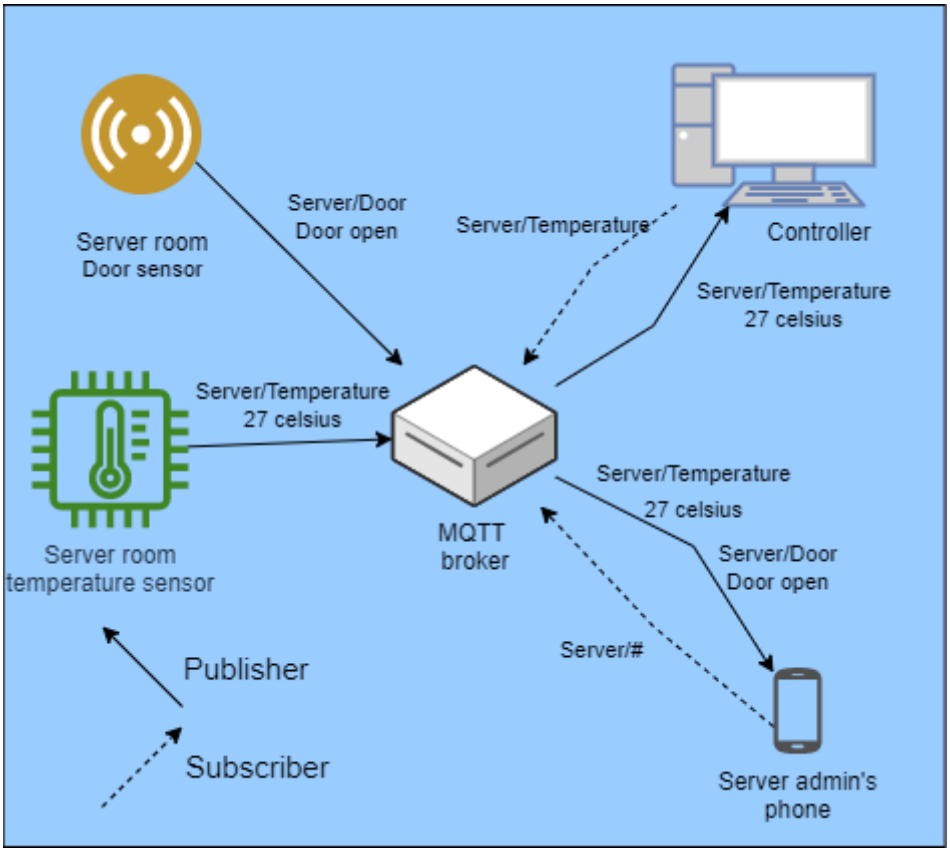

Figure 2: Simple MQTT network

The participants of the MQTT protocol are a central server (also called the broker) and the IoT devices, or clients, that connect to the broker. An example seen in Figure 2 for a network with the MQTT protocol. There are two types of clients due to the used model. There are publishers, who provide data to the broker, and subscribers who get the data from the broker. Any device can have both roles. The publisher can publish messages about a specific topic and the subscriber can subscribe to a specific topic. These topics determine which subscriber can get what messages based on the subscribed topics.
Every published message has a field for the topic, which must be filled out. It is an UTF-8 string[1], which the broker uses to determine to where to send the message. Topics can be broken apart to hierarchy levels with the (‘/’ 0x2F)

Example:

A1building/floor1/room1/rightside/temperature

Topics do not pre-exist on the broker, but they are created on-the-fly, when a message is published for that topic, or a device subscribes to that topic. Any topic name is valid except a few.

MQTT is one of the best for communicating with IoT devices, but it lacks security mechanisms on its own, e.g., it does not provide any encryption against sniffing. The best way

[1] UTF-8 string is a sequence of characters with variable-width character encoding

to secure it is the use of SSL (Secure Socket Layer) or TLS [2](Transport Layer Security). Unfortunately, this is not possible in many cases where the hardware of these devices has very low resources.

In this paper, we will see how to build a test environment for MQTT and create a MITM attack tool against MQTT. The MITM attack is a kind of attack where the attacker gets between two communicating devices and each device thinks that it communicates with the other device, but in really, the devices talk with the attacker. Once the attacker got in this position, he can alter messages, as we will see in depth later.

Related works

MITM attack on MQTT

There is a paper (Wong, 2020) where they talk about the generation of malicious messages for MQTT MITM attack in order to evade Intrusion Detection Systems (IDS) technology. They generate messages with Bidirectional Encoder Representations from Transformers(BERT), which is a machine learning(ML) technique for natural language processing. They made a parser for MQTT and tested the malicious message generator. They mainly focus on ensuring that the generated fake messages remain undetected. In contrast, I mainly focused on creating a useable tool that can support filters and rules for acting upon different messages and that provides an interactive session for the user to decide about what to do with certain captured messages.

Intrusion Detection System for MQTT

On the other hand, (Muhammad Almas Khan, 2021) presents a Deep Neural Network (DNN) based detection system for MQTT-enabled IoT smart systems. Also compared its performance with other traditional machine learning (ML) algorithms, such as a Naive Bayes (NB), Random Forest (RF), k-Nearest Neighbour (kNN), Decision Tree (DT), Long Short-Term Memory (LSTM), and Gated Recurrent Units (GRUs). The results show that the DL-based model for Bi-flow and Uni-flow featured data can achieve 99% accuracy and 98% accuracy for binary and multi-class attack classification, respectively.

The importance of IoT devices security

The summary of this paper (Bhanujyothi H C, 2020) tells that it is necessary to protect the IoT connected devices from malicious attacks and misuse which could prevent the evolution of IoT as a secure and reliable paradigm. Before providing security need to know about what the different security scenarios are. This paper gives idea about different attacking scenarios to detect different attacks that target the IoT connected devices in MQTT protocol. Also discussed about the requirements to provide security for MQTT protocol.

Specification

First, we need to setup the environment. Suppose we have a data center with servers. Each of the servers has temperature that must be between a certain range to work properly. Every hardware has a sensor which measures it and sends it over to the controller of thermostat. If one of the servers goes beyond this range, it can crash or in worst-case, set a fire in the building.

[2] https://www.rfc-editor.org/rfc/rfc8446.html

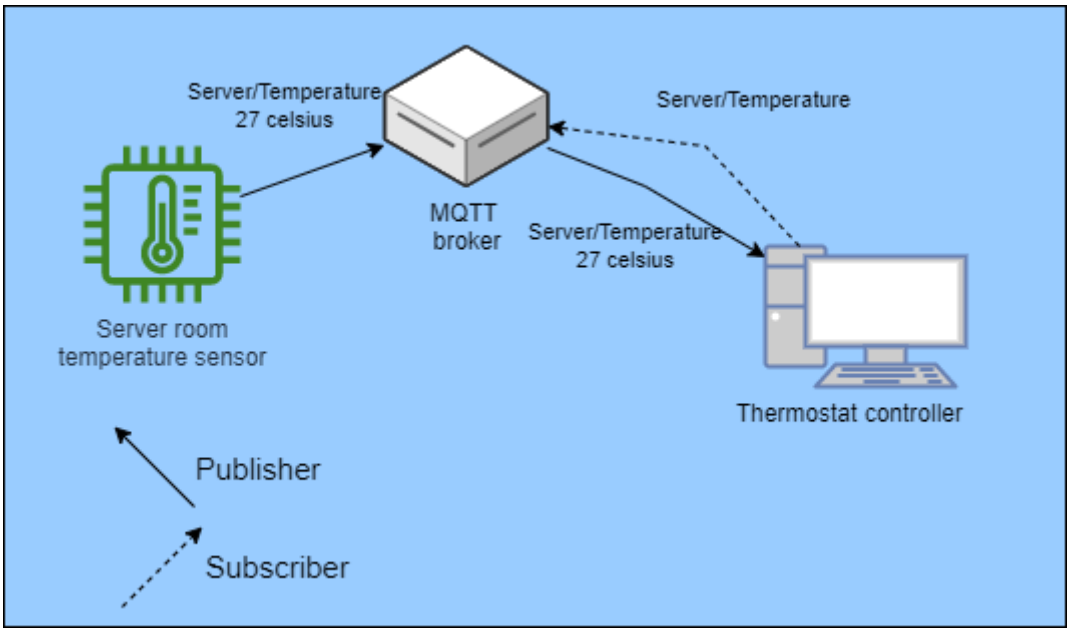

Figure 3: Basic network for example testing

To make up the environment, we will use a simple network, as shown in Figure 3. We will need the sensor for measure the room temperature, this will be our publisher. We also need a controller that will control the thermostat and play the role of the subscriber. The controller gets the temperature value from the sensor. Lastly, we need a broker to send the temperature value from the sensor to the thermostat. The controller should be a subscriber to the 'Server/Temperature' topic and will get these kinds of messages. The sensor should publish messages with topic 'Server/Temperature' and the message content will be the measured temperature value to the MQTT broker. The broker sees that 'Server/Temperature' is a topic that has a subscriber (the controller), so it sends the messages containing the temperature values to it. The thermostat processes the temperature and can change the state of the air conditioning depending on the value.

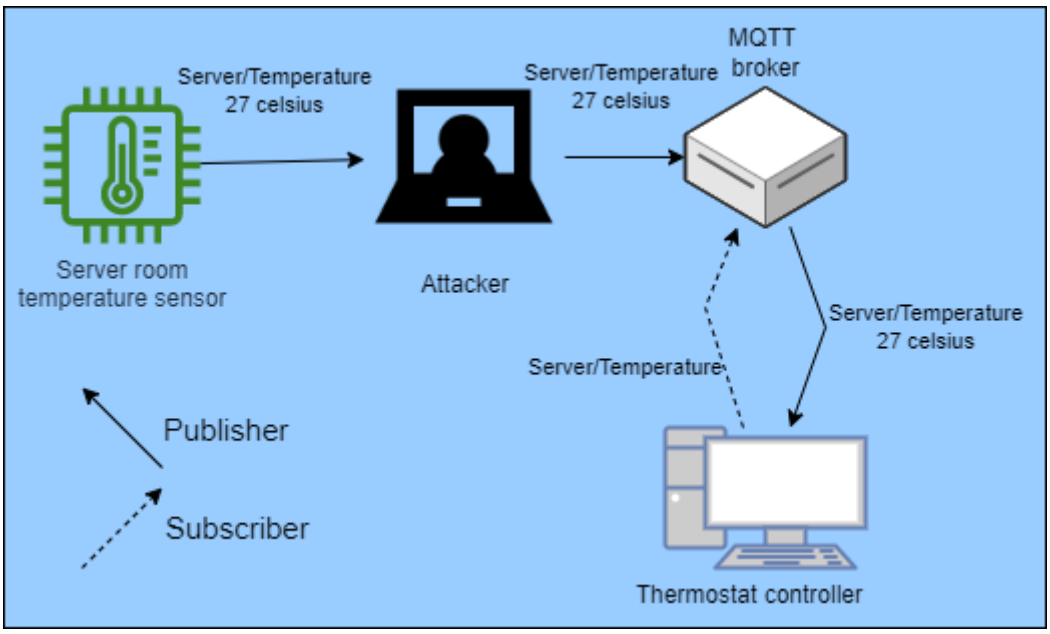

Figure 4: Basic network with an attacker

When the network is up and running, we want to intercept messages and change their content depending on the actual content and destination of the message. Figure 4 shows an attacker in a MITM attack.

```
#Dump all the data to log.txt
SAVE "log.txt"
#Save seen topics
SAVE topic "topics.txt"
#Save user data
SAVE auth "users.txt"
#Rules are evaluated from top to the bottom
#INCLUDE <TOPIC <regex>| MSG <regex>| DEFAULT>
#EXLUDE <TOPIC <regex>| MSG <regex>| DEFAULT>
#INTERACTIVE <TOPIC <regex>| MSG <regex>>
#INTERACTIVE MSG ([5-9]\d|\d{3,}) #example if the message content has greater number then 49 it will ask for what to do with it
#DISALLOW MSG ([5-9]\d|\d{3,}) #example if the message content has greater number then 49 it will drop the message

INTERACTIVE MSG ([5-9]\d|\d{3,})
#Drop message if it has in topic room1
DISALLOW TOPIC "room1"
#DISALLOW MSG "Temperature"

#Default allow or disallow messages if there us no more rule to apply ALLOW|DISALLOW
ALLOW DEFAULT
#DISALLOW DEFAULT

#With regular expression you can change message topic and content
#Every message content will be "Temp:100"
#CHANGE MSG if . to "Temp:100"
#If topic has room1 it will change to room2
CHANGE TOPIC if "room1" to "room2"
#The below rule applied to "Tempeture:13" and changed to "Tempeture:15"
CHANGE MSG if [1-9] to "Tempeture:15"
ALLOW TOPIC [1-9]
ALLOW MSG [1-9]
```

Figure 5: Example of the configuration file

Figure 5 shows what a configuration file looks like. You can filter out messages and change them the way you like and can save different kind of data from these messages. One of the interesting features you can access in the configuration file is the interactive mode with INTERACTIVE keyword. After a message is caught with a rule that is interactive the program will ask the user for further instructions. We can see this mode on Figure 6.

```
/C-Implementation/MiTM$ python3 MiTM.py
The server side starts listening.
What to do with this packet?
MQTT packet Publish message
      - Topic: 'Temperature'
      - Message: '0. tempeture: 55.6'
1. Allow
2. Disallow
3. Change
```

Figure 6: Interactive mode example

Design

Every message we get from the sensor the attacker must parse it first, so we can know what to do with it. In MQTT every message type has a unique variable header and need to parse differently. The type is determined by the first byte in the header.

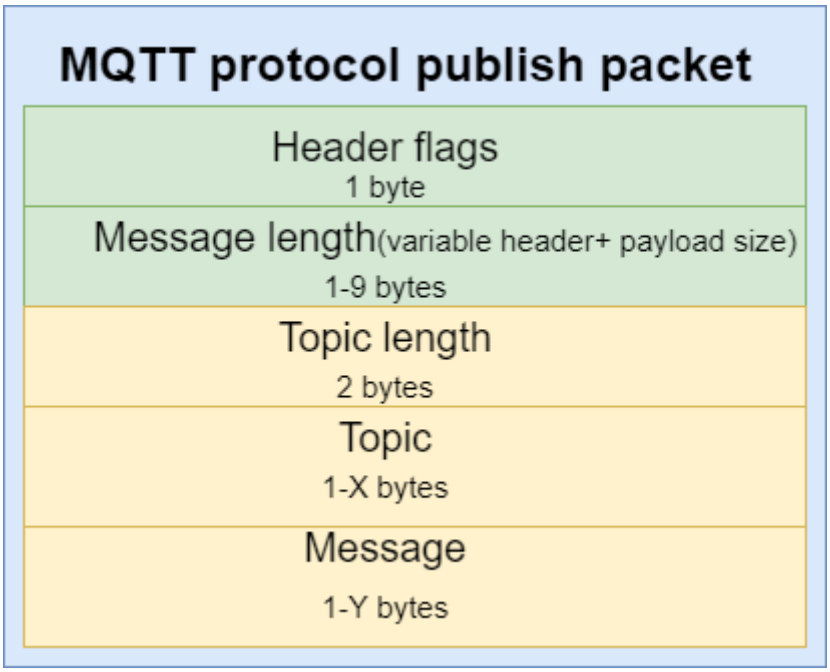

Figure 7: MQTT packet with publish type

In Figure 7, we can see a publish message, where the green part is same for every packet, but the yellow part is unique for every publish message. First, we need to parse this packet in order to get its fields. After that, we need to apply the rules that were set in the configuration file. The program at the start will read the configuration file line by line. On each line it will try to interpret it to a rule that can be later used to apply on the messages. It applies the rules in the given order, and if a rule return with an action, then we will execute it.

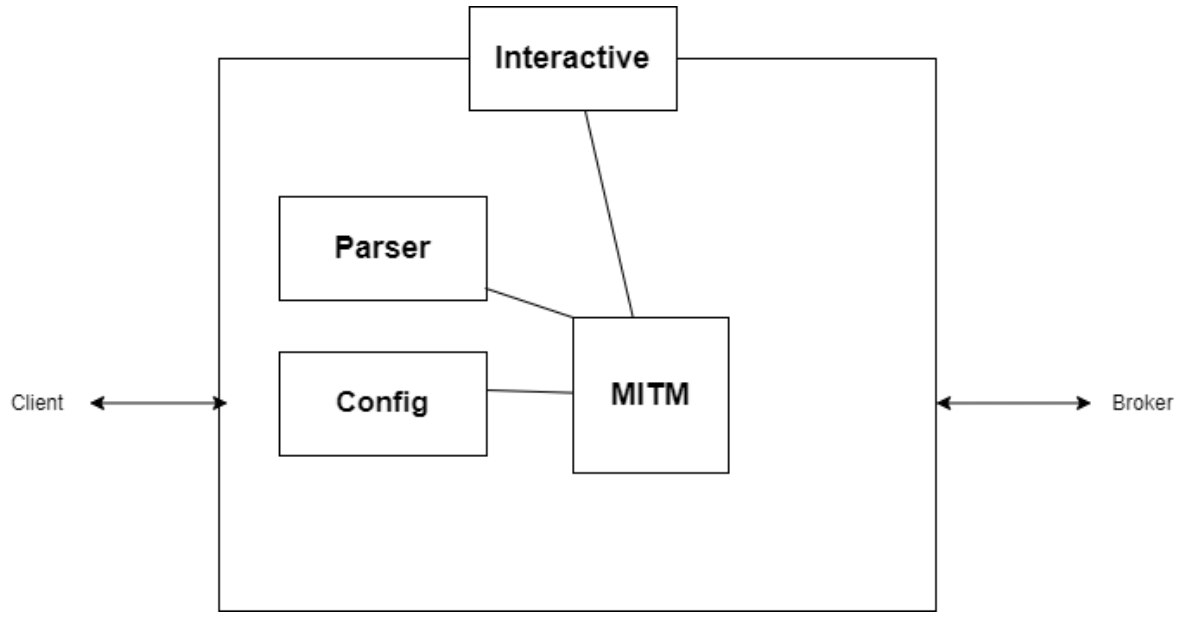

Figure 8: Architecture

Figure 8 displays the architecture of the device, where MITM is the central element responsible for communicating with the client and the broker. The Parser splits each message so that you can work with them and rebuild them if necessary. Config reads the configuration file to supervise messages based on the configuration. Interactive is the user interface.

Implementation

First, we define the listen and destination addresses. A socket start listening on the given address and wait for a connection. When the connection has been established and messaging starts between the two side each message goes through the *check()* method of the *Config_Check* class that applies the rules on the message. Depending on the return value, the message can pass (*True)*, it is blocked (*False)*, it is changed (if *bytes* were given back), or it will be put in a queue for interactive use to the user for overlook in a separate thread (if the return value of the check() function is *None*).

The main role of the Config_Check class is to apply rules on the given message and log them depending on the read config file. Process each line in the configuration file is a simple switch case construction and the interesting part is how to apply rules on the given message.

```
83      def disallow(self,command):
84          if len(command) >= 2:
85              util.valid_regex(command[1].strip('"'))
86
87          def isallow(msg):
88              if re.search(command[1].strip('"'),msg) == None:
89                  return None
90              return False
91          rule = isallow
92          if command[0] == "TOPIC":
93              self.rulesTopic.append(rule)
94          elif command[0] == "MSG":
95              self.rulesMsg.append(rule)
96          elif command[0] == "DEFAULT":
97              self.default = False
98          else:
99              raise SyntaxError
```

Figure 9: Function that add a rule to drop specific messages

Based on which part we land on the switch case we will add a function, either to the topic list or the message list. We first must check if the given regular expression syntax is correct, we do that with compiling it and seeing if there is any error. On Figure 9 we can see the function that creates rules for dropping certain messages and stores them in lists.

We read the config file and put in a function for every rule we needed. In the check function we go through each of the list first the topic depends on the return value from the function in the list we return to a value or go to the next rule. We do the same with the rules for messages. And if there is no match for any rule, we return with the default value which is *True* or *False*.

The parser simply dissects the MQTT message based on the official documentation and saw in Figure 9. For MQTT packets we need new fields depends on the message type. For *changeTopic* we calculate the new message size, the new topic length and change the topic value for the new one. For the message content we use the *changeMessage* to change the message size and the message content to the new one. With the *getHex()* we recreate the message with the new values and recalculated sizes.

In interactive mode, we use a separate thread for messages that require the attacker's attention. Whenever there is element in the queue, we take it out and with switch case we ask the user what to do with it. After we made our changes, we can drop or allow our message to send though the socket.

Demonstration

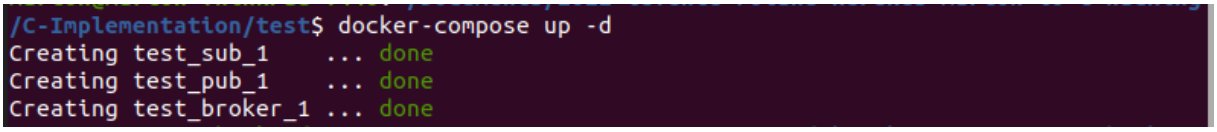

Figure 10: Start the environment

First fill the .env file and after started the environment make a user on broker based on .env file with *docker-compose exec broker mosquitto_passwd -c /mosquitto/config/mosquitto.passwd <Username>*
We can start the environment for testing purposes with the docker-compose up -d in the test folder as seen in Figure 10.

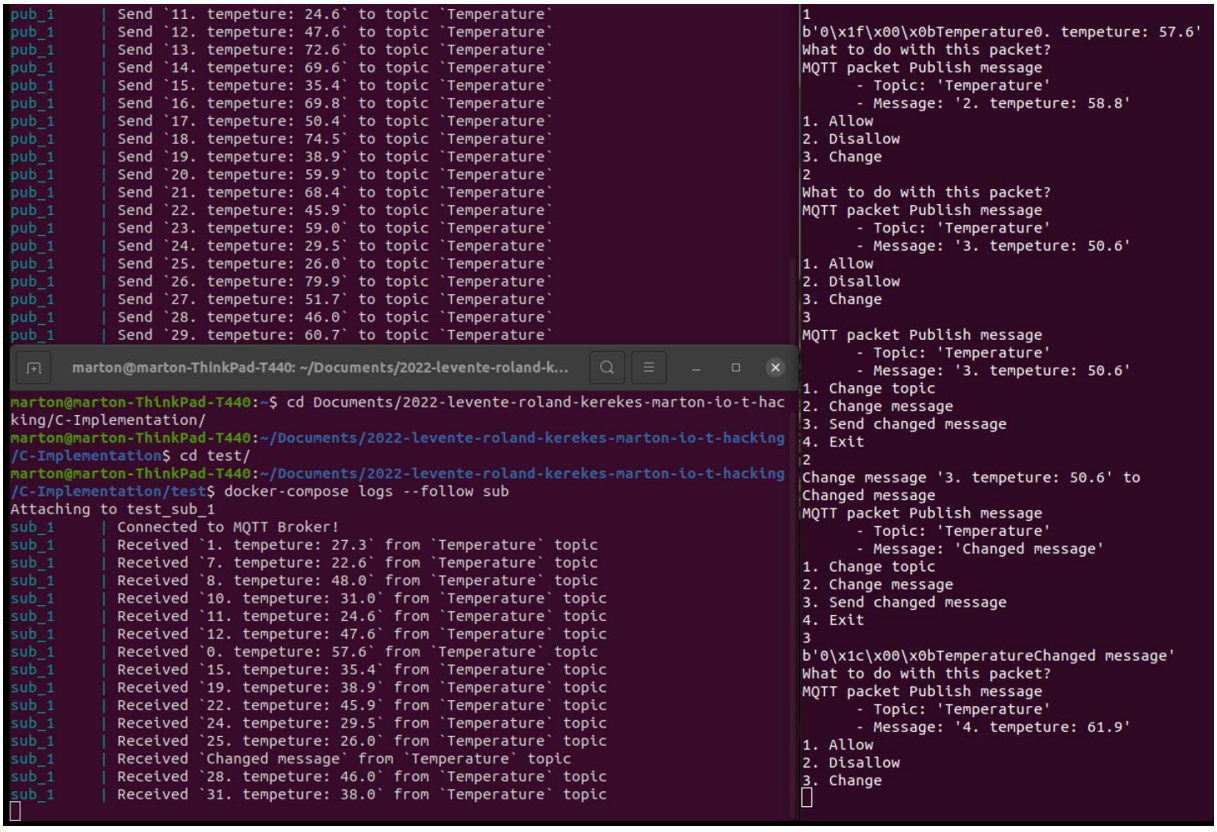

Figure 11: Publisher top left, subscriber bottom left, attacker right

On Figure 11, we can see the logs of the publisher on the top left and the subscriber on the bottom left. The attacker is on the right in the figure. We can see that the subscriber did not get most of the messages because the attacker captured them and they are waiting for user input on the right. After the attacker allowed the message to pass, we can see on bottom left

that the subscriber got the message. If the attacker wants to change the message, then he can do that on both the topic and message as seen before. On the subscriber side, we can see the message appears with the changed content.

Conclusion

In this paper we could see how the MQTT protocol works. We have shown how a MITM attack works and made a tool to capture messages and to drop or change them in a testing environment. How you set up predefined rules from a file for filtering messages. In the future, one can add new types of rules to filter out messages in more depth depending on the header, like for example QoS feature of the message or to implement an undetectable message generator like with BERT discussed in related work. You can use this tool for just attacking a network for testing out how to detect this kind of MITM attacks.

Reference

[1]“HiveMQ. *HiveMQ.* https://www.hivemq.com/mqtt-essentials/

[2]Bhanujyothi H C, V. J. (2020). *Diverse Malicious Attacks and security Analysis on.* Journal of Xi'an University of Architecture & Technology.

[3]Muhammad Almas Khan, M. A. (2021). *A Deep Learning-Based Intrusion Detection System for MQTT.* MDPI.

[4]Wong, H. (2020). *Man-in-the-Middle Attacks on MQTT-based IoT Using BERT.* San Diego, CA: KDD 2020 Workshops (AIoT).