Security assessment of common open source MQTT brokers and clients

Edoardo Di Paolo[1], Enrico Bassetti[1] and Angelo Spognardi[1]

[1] *Computer Science dept., Sapienza University of Rome, Italy*

Abstract

Security and dependability of devices are paramount for the IoT ecosystem. *Message Queuing Telemetry Transport* protocol (MQTT) is the de facto standard and the most common alternative for those limited devices that cannot leverage HTTP. However, the MQTT protocol was designed with no security concern since initially designed for private networks of the oil and gas industry. Since MQTT is widely used for real applications, it is under the lens of the security community, also considering the widespread attacks targeting IoT devices. Following this direction research, in this paper we present an empirical security evaluation of several widespread implementations of MQTT system components, namely five broker libraries and three client libraries. While the results of our research do not capture very critical flaws, there are several scenarios where some libraries do not fully adhere to the standard and leave some margins that could be maliciously exploited and potentially cause system inconsistencies.

1. Introduction

The number of devices connected to the Internet is growing very rapidly recently, driving a new wave of technologies and applications in various fields. One of these trends is the so-called *Internet-of-Things*: the explosion of low-cost, small/micro devices (often single-purposes and with an IP stack), Ethernet port and some space for programming paved the way for a whole new spectrum of applications.

Usually, these devices have a tiny amount of resources, so that common protocols like HTTP cannot be efficiently implemented without sacrificing key features of the protocol itself (leading to non-standard implementation) or key parts of the "business logic" (i.e. the main purpose of the device). To overcome this limitation, several lightweight protocols were invented, like MQTT (*Message Queuing Telemetry Transport*) protocol or AMQP (*Advanced Message Queuing Protocol*) [1]. When resources are severely limited (simple sensors/actuators) and the system is in under-constrained environments (low-speed wireless access), the former is the preferred choice [2]. MQTT is a publish-subscribe protocol based on a simple message structure, basic features and a minimal packet size (considering the message headers). Thanks to this design, nearly all IoT devices use MQTT or similar lightweight protocols to talk to each other and communicate with the rest of the world. Also, it has undergone several standard processes, and MQTT v. 3.1.1 and 5.0 are both ISO standards [3].

ITASEC21: Italian Conference on Cybersecurity, April 07–09, 2021, Online

dipaolo.1728334@studenti.uniroma1.it (E. D. Paolo); bassetti@di.uniroma1.it (E. Bassetti); spognardi@di.uniroma1.it (A. Spognardi)

CEUR Workshop Proceedings (CEUR-WS.org)

The protocol was conceived with no security concern, since initially designed for private networks of the oil and gas industry [4]. The adoption of the protocol has ramped up, and several statistics show that many devices use it without any protection [5]. Also, considering the privacy aspects, given its quite limited features, the MQTT protocol has no built-in encryption features; farther, the use of TLS to provide a secure communication channels is very limited: at the time of writing, comparing with the Shodan search engine the prevalence of the exposed IoT and IIoT devices using MQTT, we have that those that use port 8883 (MQTT over SSL/TLS) is 42, while those using port 1883 (MQTT-unencrypted) is 154632 [5]. Moreover, MQTT applications keep receiving critiques, with the claim that they adopt weak protocol implementations, even if some of them, like the Mosquitto library, offer extension plugins to improve security[1] (i.e. role-based authentication or Access Control List, not part of the MQTT standard).

Since MQTT is widely used for real applications, it is under the lens of the security community, also considering the widespread attacks targeting IoT devices. The research is focusing on shifting towards ensuring secure IoT systems, for example, implementing access control mechanisms [6], lightweight cryptography [7] or remote attestation of devices [8]. An essential aspect of this context is discovering unforeseen security risks resulting from the necessary interoperability with different implementations of MQTT libraries.

Following this research direction, in this paper we present an empirical security evaluation of several widespread implementations of MQTT system components, namely five broker libraries and three client libraries. Moreover, we also applied our security analysis to an MQTT client embedded in a real IoT device, namely a Shelly DUO Bulb. This IoT device is a remote-controlled LED light bulb. It supports Wi-Fi connectivity and acts as an MQTT subscriber to receive commands, like powering on/off or light dimming.

Our evaluation has aimed to verify the responses of the components of the different libraries to different MQTT messages to see their behaviour in situations where the standard does not indicate clearly how the message (or the connection itself) is supposed to be handled. These mishandling might create interoperability issues or even open doors to malicious attackers. While the results of our research do not capture very critical flaws, there are several scenarios where some of the libraries do not fully adhere to the standard and leave some margins that could be maliciously exploited and potentially cause system inconsistencies.

The structure of the paper is the following: Section 2 reports the state of the art concerning the security analysis of MQTT, while in Section 3 we provide the details about our research methodology. Section 4 reports the results of our security analysis, and the last section concludes the paper with some remarks and future directions.

2. Related works

As the abundance of surveys suggests [9, 10, 11], security and dependability of IoT devices is paramount for the whole ecosystem. In this context, the MQTT protocol plays a determinant role. In 1999 Andy Stanford-Clark (IBM) and Arlen Nipper (then working for Eurotech, Inc.) proposed the MQTT protocol [12] to monitor oil pipelines within the SCADA framework [13]. Since then, it has been revised in two main versions, namely 3.1.1 (last update December 2015)

[1] https://mosquitto.org/documentation/dynamic-security

and 5 (last update March 2019). To date, the former is by far the most used in real applications, the latter being much newer and still not well adopted [13].

Like all the network protocols becoming a standard, it has undergone many reviews both formally and empirically. Several papers focus on MQTT formal modelling and performance analysis [14, 15, 16], others on its possible vulnerabilities, and many others on its security analysis. In this research, we focused on the security analysis and the comparison of several of the most spread software libraries implementing the MQTT protocol. Instead of using static analysis of their code, as in [17], we performed a dynamic analysis using the *fuzzing* methodology. In [18], the authors proposed a template-based fuzzing framework and tested its effectiveness against two implementations of MQTT. Using this method, they found some security issues: Moquette and Mosquitto brokers were affected by a vulnerability that would have led to a DoS attack in specific settings if exploited. In our research, we are focusing not only on possible DoS attacks but also on the effects of standard violations of both brokers and clients. Moreover, our analysis applies to five different brokers, three clients and a physical device.

In [19], the authors evaluated the robustness of several MQTT implementations against a subtle family of attacks known as low-rate denial of service. Similarly to this work, a real testbed was set up, and several experiments performed, validating the open vulnerability of all the MQTT implementations.

In [20], authors describe a new strategy to test MQTT through fuzzing and how much it is efficient against the protocol. However, they do not present any results about the application of their strategy. A similar approach is adopted in [21], where the authors propose to apply fuzzing techniques in a container-based environment (Docker). This would allow a large scale test of the MQTT protocol. However, again, the authors do not compare different implementations (they only consider Mosquitto), neither describe the type of attacks they performed.

A different methodology based on *attack patterns* [22] was proposed by *Sochor et al.* and was used to spot hidden vulnerabilities in different broker implementations. They adopted a method to randomly generate test sequences (Randoop) to challenge the different brokers, and they were able to find several failures and unhandled exceptions. Our research adopted a different methodology, tested different broker MQTT implementations, and included clients.

Another methodology to perform a dynamic analysis is model-based testing, as proposed for MQTT applications in [23]. The methodology considers using a finite state machine that verifies the properties of the software and proposes extensions to model-based tools for MQTT applications.

2.1. MQTT overview

MQTT implements the publish-subscribe communication paradigm (Figure 1): *clients* send messages on a *topic* to servers (named *brokers*) that are responsible for delivering them to the interested clients, the final recipients of the messages. Brokers are, then, intermediaries that accept messages and forward a copy of each message to the clients who previously *subscribed* for a given *topic*. A topic is a UTF-8 string obtained joining one or more topic levels with the slash character, like in `/home/basement/kitchen/temperature/` and the client subscriptions can be made to a topic or part of it, thanks to the use of wildcards, like in `/home/basement/#`.

In a general MQTT session, a client establishes a connection with a broker (CONNECT-CONNACK exchange), subscribes to one or several topics (SUBSCRIBE-SUBACK exchange), publishes contents (PUBLISH-PUBACK exchange), receives other client contents (PUBLISH or PUBLISH-PUBREC-PUBREL exchange, according to the QoS) and terminates the session (client DISCONNECT). The CONNECT packet can implement an authentication mechanism, based on username and password. All the exchanges happen using a clear text TCP session on port 1883 or, if TLS is used, using an encrypted session on port 8883. Encrypted exchanges are mainly used when authentication is enforced, so that username and password are protected against eavesdropping.

In addition to the topic, any message also has a *Quality-of-Service* value (*QoS*), taken in the range 0–2. A QoS equals 0 corresponds to no guarantees, and it means that the message can be lost or delivered multiple times. A client sending a message with QoS equals 1 requires that the message should never be discarded, while it might be delivered multiple times; in this case the sender stores the message until it receives back a PUBACK packet that ensures reception. Similarly, with a QoS set to 2, the message should never be discarded and it should be delivered exactly one time; in this case the client will wait for a PUBREC packet and, once received, it will discard the PUBLISH and it will send a PUBREL packet. The last packet that the client will receive in this exchange is the PUBCOMP that will release the id of the PUBLISH for reuse. It is important to highlight that the publisher QoS is not associated with the subscriber QoS – unless the implementation supports this non-standardized feature.

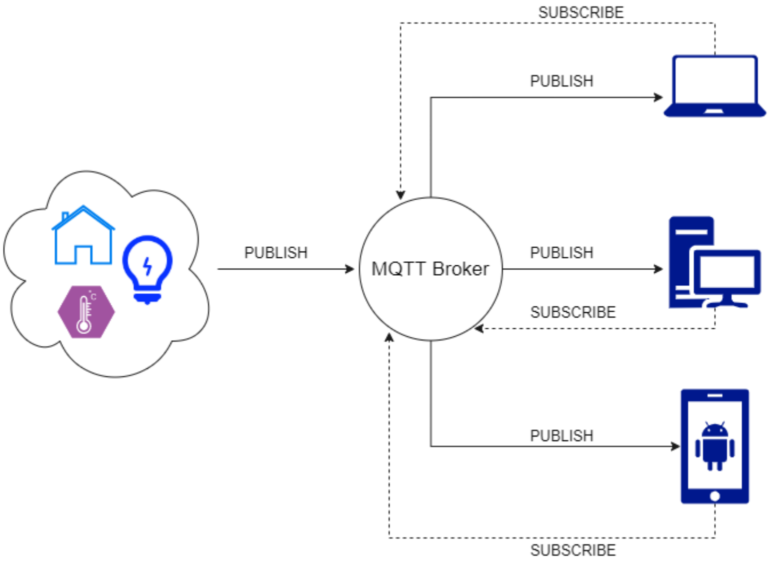

Figure 1: Typical MQTT architecture: IoT devices (clients) publish their messages to the broker. Subscribers ask the broker to receive only those messages with topics they subscribe. Broker relays (publishes) to each subscriber only messages with subscribed topics.

3. Methodology

The purpose of our research was to compare the behaviour of different implementations of MQTT. The first step has been finding and setting up the most popular open-source brokers and client libraries that people use to manage their devices or develop common software solutions. To determine the popularity, we took into account the number of stars and forks on GitHub repositories and the number of blog posts citing the examined brokers.

We focused only on open source libraries, namely: *Mosquitto, EMQ X, HiveMQ, Moquette* and *Aedes* for the brokers, *paho, mqttools and mqtt.js* for the clients. We will discuss the brokers and the clients respectively in Section 4.1 and in Section 4.2. Some of these have thousands of instances running in "production" environments, in common consumer and business-to-business solutions. We also tested a popular low-cost *Internet-of-Things* device, namely the Shelly DUO Bulb (Section 4.3).

The next step has been to evaluate the type of tests to apply, considering the MQTT standard specification, version 3.1.1. We specifically looked for undefined behaviours, unspecified states or other missing information about message handling. Also, we looked for parts of the standard that might lead to a wrong implementation (e.g. expected actions by the broker/client that are implied but not specified or not clearly specified). This allowed us to focus our testing on a restricted subset of cases, as explained in Section 4.

We created different sets of experiments to find possible anomalies in MQTT implementations, developing our fuzzer written in python, with the help of the twisted library[2]. Our custom fuzzer allowed us to manage different streams correctly and send custom packets: for example, we could change every bit of the packets to see the brokers behaviour even in the presence of malformed packets. Standard, common MQTT libraries, instead, implement some state-machine which are expected in some part of the protocol (e.g. QoS2): a straightforward use of such libraries would not allow arbitrary changes in the flow of the messages, like out-of-order messages. Each experiment has been codified in a *JSON* file that specifies the sequence of actions or packets that the test should run on/against the software under test, and the final behaviour of the involved parties have been logged and analyzed.

4. Experimental results

4.1. Brokers

A broker is a fundamental component in an MQTT architecture. Its job is to accept messages from clients (acting as "publishers") and then forward them to all clients with subscriptions ("subscribers") matching the topic of the message. This loosely coupled architecture allows the clients to not communicate directly with each other.

Modern brokers support many concurrent connections and messages per second. A flaw in message state machines, packet parsing, topic logic, etc., might expose a high impact vulnerability, which a malicious actor might exploit to launch some attacks like a *Denial-of-Service.*

We analyzed five common MQTT brokers:

[2] https://github.com/twisted/twisted

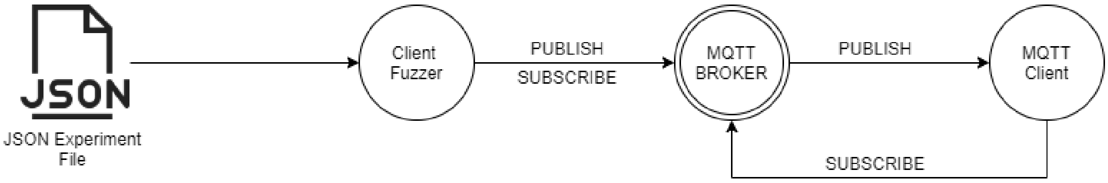

Figure 2: Schema of the testbed for the experiments: the fuzzer, which acts as a typical client, takes in input a "JSON experiment file" containing the client's packets to the MQTT broker. The fuzzer will also receive all the PUBLISH packets sent to the broker. The MQTT Client, instead, uses one of the libraries that are examined in the subsection 4.2.

- **Mosquitto**[3]: it is one of the most used MQTT brokers in the world. It is a single-threaded, lightweight broker written in C. This broker has been widely used thanks to its flexibility.
- **EMQ X**[4]: it is written in Erlang and it claims to be so efficient to be "the Leader in Open Source MQTT Broker for IoT".
- **HiveMQ**[5]: a broker written in Java. It supports MQTT version 3.x and 5.0 and it is widely used in automation and industrial systems. We tested the Community Edition.
- **Moquette**[6]: another Java-powered open-source broker. It is very lightweight but it is less-known and less-used, when compared to other brokers.
- **Aedes**[7]: a broker written in JavaScript/NodeJS. It is the successor of MoscaJS. It does not support version 5 of MQTT, but it is fully compatible with version 3.x and supports several extension libraries.

Each broker underwent the same set of tests specified in the next section. We performed more than 60 different experiments on a consumer-grade PC with local connections. A summary of the results is in Table 1.

4.1.1. Experiments and results

Publish QoS 2 and 1. In this experiment, the client performs the following steps:

1. it sends a SUBSCRIBE packet with a specific topic;
2. it sends the first PUBLISH packet with a *quality of service* 2 and with id 1 over the topic specified in the subscription;
3. it sends the second PUBLISH packet with a *quality of service* 1, still with id 1 over the topic specified in the subscription;
4. it sends a PUBREL packet for the first packet sent.

We noticed different broker behaviours: Mosquitto publishes the first received packet with QoS 2, but then it loses the second packet that is not published to the clients due to the PUBCOMP

[3] https://mosquitto.org/
[4] https://www.emqx.io/
[5] https://www.hivemq.com/developers/community/
[6] https://github.com/moquette-io/moquette
[7] https://github.com/moscajs/aedes

Table 1
Brokers test result summary. The tested versions were the latest stable, available at the time of our experiments.

Broker	**Anomalies found**	**Security problems**	**Version**
Mosquitto	when handling *quality of service.*	Possible unwanted application scenarios.	1.16.12
EMQ X	when handling *quality of service* and *long topics.*	Possible unwanted application scenarios.	4.2.1
HiveMQ	when handling *quality of service* and *long topics.*	Possible unwanted application scenarios.	2020.5
Moquette	when handling *quality of service* and *long topics.*	Possible *denial of service* and unwanted application scenarios.	0.13
Aedes	when handling *quality of service* and *packet references.*	Possible *denial of service.*	0.43.0

packet that is not received, and so the packet id is not available for use. The EMQ X broker publishes both packets; it handles the flow for the first packet and then the flow for the second one. In HiveMQ and in Moquette, the client that sends packets receives the publication first and after the *pubcomp*, concerning the first packet. Additionally, in HiveMQ the client receives the *pubcomp* back first and then the *pubrec*. Aedes publishes both packets, but the *pubcomp* arrives at the client after the two publications. This behaviour repeated several times, also in the other experiments regarding the *quality of service* that are described below.

Publish QoS 2 and 0. This experiment is very similar to the one described above, but in this case, the client performs the following steps:

1. it sends a SUBSCRIBE packet with a specific topic;
2. it sends the first PUBLISH packet with a *quality of service* 2 and with the id 1 over the topic specified in the subscription;
3. it sends the second PUBLISH packet with a *quality of service* 0 and with the id 1 over the topic specified in the subscription;
4. it sends a PUBREL packet for the first packet sent.

Mosquitto, in this case, publishes both packets but in reverse order: it handles the one with *quality of service* 0 first, and then it handles, correctly, all the flow regarding the first packet sent with *quality of service* 2. EMQ X and HiveMQ maintain the order of the packets published by the client; also, in the case of HiveMQ, the client received back the *pubcomp* first and then the *pubrec* regarding the packet with *quality of service* 2. Moquette behaves similarly to EMQ X, but, in this case, the *pubcomp* arrives after the publication of the second packet. Aedes has the same behaviour as Mosquitto, but the *pubcomp* arrives after the publication as in the previous experiment.

Double publish QoS 2. In this experiment, the client performs the following steps:

1. it sends a SUBSCRIBE packet with a specific topic;

2. it sends the first PUBLISH packet with a *quality of service* 2 and with the id 1 over the topic specified in the subscription;
3. it sends the second PUBLISH packet with a *quality of service* 2 and with the id 1 over the topic specified in the subscription;
4. it sends a PUBREL packet for the first packet sent;
5. it sends a PUBREL packet for the second packet sent.

In Mosquitto, only one publication referred to the first packet sent, but the flow regarding the *quality of service* is properly handled. EMQ X, in this case, has the same behaviour as Mosquitto. Instead, HiveMQ and Moquette publish both packets in the correct order. In Aedes, there is a different behaviour: the broker publishes two packets, but they are the same packet, the first one sent by the client.

Long topic. In the MQTT standard, the maximum length topic is 65536 bytes, but we saw that in the source code of EMQ X there is a constant that represents the maximum length, and its value is 4096. So, we tried to subscribe to a topic with more than 4096 bytes. In Mosquitto the subscription to the topic is successful. Instead, HiveMQ cuts the topic to which the client is subscribing to. In Moquette there is an *IOException* and then the client disconnection. In Aedes there is a crash of the broker and the client; in particular, the exception thrown by the experiment generated an error like "*too many words*". In EMQ X the client disconnects.

Other experiments. Further experiments are listed below. They have been briefly summarized, since the behaviour of all the brokers was correct.

- some experiment where the *client id* value in the packet contains characters non-UTF-8 encoded: no anomalies. In detail, we have built a connection packet with the *client id* containing particular characters and the experiment was handled correctly by all brokers;
- *Keep-alive* field in connection packet as a string: in all brokers there is the client disconnection due to malformed packet;
- subscription (or publication) in an invalid *wildcard*: in all brokers there is the client disconnection due to "invalid topic";
- topics and *wildcard* encoded in: *utf-16, zzlib, bz2* and *base64*. In the last three cases, there were no anomalies to report. In the *utf-16* experiment, in all brokers the client disconnects. A particular experiment was the one with many " / " in the topic value; in *Mosquitto, EMQ X, Moquette* and *Aedes* there was client disconnection while in HiveMQ there was a cut of the topic and then the client subscription;
- packets flood with QoS 0: all brokers handled the flood well;
- invalid protocol name (or version) in the connection packet: in all cases, the client disconnects;
- sending a *pubrel* packet that references a publication packet that was never sent: all brokers, except for Aedes, sent back a *pubcomp* message. In Aedes there is the client disconnection.

4.2. Clients

In addition to tests on brokers, we also carried out tests on client libraries available in the web. In particular, we studied three different client libraries: *paho-mqtt*[8], *mqttools*[9] and *mqtt.js*[10]; the first two are written in *python* while the third is written in *javascript*. Again, we considered metrics like the number of stars and forks on GitHub repositories. However, the experiments have not found particular anomalies. Here there is a list of tests we tried:

- invalid QoS level: all libraries report an error about the QoS, blocking the sending of the packet;
- invalid *wildcard* subscription: in this case *mqtt.js* generates an "Invalid topic" error, while the other two libraries timeout;
- *client id* not encoded in utf-8: in *mqttools* the client cannot connect to the broker, in *paho-mqtt* there is a successful connection to the broker and *mqtt.js* generates an error with the consequent client disconnection;
- publication (or subscription) to a topic with a length more than 65536 characters: in all libraries there is the client disconnection.

Table 2
Client libraries test results. The tested versions were the latest stable, available at the time of our experiments.

Library	Anomalies found	Security problems	Version
paho-mqtt	when handling subscription (or publication) to an *invalid wildcard*.	It produces an *hang*.	1.5.1
MQTT.js	when handling an invalid *quality of service*.	There is a *crash* of the client due to a *TypeError*.	4.2.1
mqttools	when handling subscription (or publication) to an *invalid wildcard* and when the *client id* value contains not *utf-8* characters.	It produces an *hang* and an infinite connection loop.	0.47.0

4.3. Physical device

In "home automation" the MQTT protocol is widely used as most smart devices supports it. Many software applications allow you to use the protocol to manage the smart devices, and one of them, for example, is *Home Assistant*; also *Amazon*, in *AWS IoT*, uses MQTT to connect the user's devices to other devices and other services.

We decided to perform the previous experiments on a physical device. In this case, we tested a *Shelly* light bulb that supports MQTT: it is possible, for example, to turn on or off the device through specific commands sent in the local network. In this device, the protocol configuration can be done in a simple web interface, available in the local network, where the user has to

[8] https://pypi.org/project/paho-mqtt/
[9] https://pypi.org/project/mqttools/
[10] https://github.com/mqttjs/MQTT.js

specify the broker's IP and port. The username and password are not mandatory during the configuration; this could be a security issue, depending on the context. This device does not have an "anti-flood" regarding the packets it receives; for example, it is possible to turn off and on the light bulb repeatedly and quickly by sending a PUBLISH packet on the specific topic with specific content. The software that runs in the light bulb is the same as other Shelly devices, so this problem also affects them. Therefore, it is possible to send many packets that overload the device's electronic components to make it useless.

To confirm the obtained results in the brokers, we performed on the device the same set of experiments previously carried out on both brokers and clients. For example, the experiment "Publish QoS 2 and 1" (discussed in Section 4.1) has confirmed the expected results. In this case, we sent a "turn on packet" with QoS 2 and then we sent a "turn off packet" with QoS 1. When Mosquitto was the broker used, the light bulb turned on but then did not go off, while in all other cases, the light bulb turned on and then turned off.

In addition to these experiments already performed for the various brokers, we have tried to generate some *buffer overflows*, through the payload sent to the device, with a consequent *DoS*. However, the light bulb passed all tests without errors; in particular, the device ignores any form of payload other than what it expects to receive.

4.4. Discussion

Interesting results have been obtained from the experiments carried out on brokers, clients, and the physical device. In [5] Kant et al. shown that many consumer-grade devices do not use a secure transport for MQTT (like TLS); this is due to the few resources on-board (in turn, this is caused by the target price of these devices in the consumer market, which is very low). Sometimes these devices lack proper authentication protocols, again due to missing resources or insecure default configurations. These security issues could lead attackers to control devices (e.g. they could control an entire *home environment* remotely) directly in some cases (e.g. with *Man-in-the-Middle* attacks).

In our work, the model of the attacker includes the capability to modify the MQTT packet flow, delaying the transmission or making it out of order, or modifying MQTT packets payload, injecting invalid values. This capability can be exploited with limited access to the broker or intermediate network devices, or even remotely, by using other attacks like *Distributed Denial-of-Service* or *flooding* against a network device in the path of the packet flow (for delaying packets, for example). Some of these vulnerabilities can be exploited with an older version of TLS protocol itself[11]: for example, SSL used a vulnerable *Message Authentication Code* until TLS [24]; vulnerabilities in TLS HMAC implementations are still found years after the standard [25].

We described bad behaviours for brokers in Section 4.1.1. Some of them can be classified as vulnerabilities, and they can lead to attacks if some conditions happen. In fact, we saw that brokers publish some messages violating the protocol state machine in some tests. An example of this is the out-of-order Aedes (and other brokers) use of *pubcomp*, which might be used to trigger a **replay attack** if the *pubcomp* itself is delayed or dropped by a malicious actor. This attack can disrupt or even damage some devices: for example, IoT-mechanical devices can be continuously triggered until the mechanical part is over-stressed.

[11] Note that low-cost IoT devices often implement old protocols, sometimes even partially

Another violation of the standard which leads to a vulnerability (in all brokers but Mosquitto) is the bad handling of long topics: in the MQTT standard, the maximum length topic is 65536 bytes. However, trying to publish to a very long topic (>4096 bytes) leads to a disconnection of the client. A malicious actor that can inject (even indirectly, think user-provided information) some characters in the topic may cause a Denial of Service for that client. **Even worst, in Aedes there is a crash of the broker itself, leading to a *Denial of Service* for the entire MQTT network**.

Some violations of the standard are so misinterpreted that each broker does a different thing: in *Double publish QoS 2* test, nearly all brokers (the only exception is Mosquitto) violates the "unique identifier" feature of MQTT in different ways. This causes no direct impact as a violation, but it can be exploited if some client library shows some bad handling of this situation.

Among all brokers, our tests show that Mosquitto seems to be the strongest one in terms of MQTT standard adoption, and so the safest from a security point of view.

Instead, client libraries have shown only minor issues, many of them relating to encoding errors or long topic subscription issues. Our tests show that they are quite robust, sometimes even better than some brokers.

5. Conclusions

MQTT is considered one of the enabling technologies for the IoT ecosystem. It is adopted by almost all IoT applications that run on devices with limited computational power, thanks to the high availability of open source libraries implementing MQTT. In this paper, we have presented an empirical study of the most popular implementations of both brokers and clients, considering the possible behaviour deviations from the standard that could lead applications to possibly inconsistent or even critical states. We also tested a physical smart-home device. The results of our experiments were noticeable: while almost all the considered libraries are correctly handling most of the interactions, as expected, some anomalies have been detected that could be exploited and target the applications, mainly exposing them to denial of service attacks.

Given the promising results, we plan to expand the number of considered libraries further and to include in our experiments some real applications to create some proof-of-concept attacks that exploit the found anomalies. Similarly, we plan to extend the experiments also considering the libraries that also support the new version of the MQTT protocol, namely version 5.

Acknowledgments

This research has been partially supported by MIUR (Italian Ministry of Education, University, and Research) under grant "Dipartimenti di eccellenza 2018-2022" of the Computer Science Department of Sapienza University of Rome.

References

[1] N. Naik, Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http, in: 2017 IEEE International Systems Engineering Symposium (ISSE), 2017, pp. 1–7. doi:10.1109/SysEng.2017.8088251.

[2] J. E. Luzuriaga, M. Perez, P. Boronat, J. C. Cano, C. Calafate, P. Manzoni, A comparative evaluation of amqp and mqtt protocols over unstable and mobile networks, in: 2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC), 2015, pp. 931–936. doi:10.1109/CCNC.2015.7158101.

[3] ISO, ISO/IEC 20922:2016 Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1, 2016. URL: https://www.iso.org/standard/69466.html.

[4] MQTT.org, Who invented MQTT, 2021. https://mqtt.org/faq/, last accessed: 28/02/2021.

[5] D. Kant, A. Johannsen, R. Creutzburg, Analysis of IoT Security Risks based on the exposure of the MQTT Protocol, Technical Report, Technische Hochschule Brandenburg, 2021.

[6] P. Colombo, E. Ferrari, Access Control Enforcement within MQTT-Based Internet of Things Ecosystems, in: Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies, SACMAT '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 223–234. URL: https://doi.org/10.1145/3205977.3205986. doi:10.1145/3205977.3205986.

[7] Dinculeană, Dan and Cheng, Xiaochun, Vulnerabilities and Limitations of MQTT Protocol Used between IoT Devices, Applied Sciences 9 (2019). URL: https://www.mdpi.com/2076-3417/9/5/848. doi:10.3390/app9050848.

[8] A. Francillon, Q. Nguyen, K. B. Rasmussen, G. Tsudik, A minimalist approach to remote attestation, in: 2014 Design, Automation Test in Europe Conference Exhibition (DATE), DATE '14, Leuven, BEL, 2014, pp. 1–6.

[9] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, W. Zhao, A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications, IEEE Internet of Things Journal 4 (2017) 1125–1142. doi:10.1109/JIOT.2017.2683200.

[10] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, M. Ayyash, Internet of things: A survey on enabling technologies, protocols, and applications, IEEE Communications Surveys Tutorials 17 (2015) 2347–2376. doi:10.1109/COMST.2015.2444095.

[11] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, S. Clarke, Middleware for internet of things: A survey, IEEE Internet of Things Journal 3 (2016) 70–95. doi:10.1109/JIOT.2015.2498900.

[12] A. Banks, R. Gupta, MQTT version 3.1.1 plus errata 01, 2014. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[13] Dave Locke, IBM UK Labs, MQTT Past, Present and Future: 20 Years of MQTT, 2019. https://info.thingstream.io/hubfs/IBMWatsonMQTT,MQTT-SNpresentation.pdf, last accessed: 28/02/2021.

[14] M. Houimli, L. Kahloul, S. Benaoun, Formal specification, verification and evaluation of the mqtt protocol in the internet of things, in: 2017 International Conference on Mathematics and Information Technology (ICMIT), 2017, pp. 214–221. doi:10.1109/MATHIT.2017.8259720.

[15] K. Hofer-Schmitz, B. Stojanović, Towards formal methods of iot application layer protocols,

in: 2019 12th CMI Conference on Cybersecurity and Privacy (CMI), 2019, pp. 1–6. doi:10.1109/CMI48017.2019.8962139.

[16] M. Collina, M. Bartolucci, A. Vanelli-Coralli, G. E. Corazza, Internet of things application layer protocol analysis over error and delay prone links, in: 2014 7th Advanced Satellite Multimedia Systems Conference and the 13th Signal Processing for Space Communications Workshop (ASMS/SPSC), 2014, pp. 398–404. doi:10.1109/ASMS-SPSC.2014.6934573.

[17] Ferrara, Pietro and Mandal, Amit Kr and Cortesi, Agostino and Spoto, Fausto, Static analysis for discovering IoT vulnerabilities, International Journal on Software Tools for Technology Transfer 23 (2021) 71–88. doi:10.1007/s10009-020-00592-x.

[18] S. Hernández Ramos, M. T. Villalba, R. Lacuesta, MQTT Security: A novel fuzzing approach, Wireless Communications and Mobile Computing 2018 (2018). doi:10.1155/2018/8261746.

[19] I. Vaccari, M. Aiello, E. Cambiaso, Slowite, a novel denial of service attack affecting mqtt, Sensors (Basel, Switzerland) 20 (2020).

[20] L. G. Araujo Rodriguez, D. Macêdo Batista, Program-aware fuzzing for mqtt applications, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 582–586. URL: https://doi.org/10.1145/3395363.3402645. doi:10.1145/3395363.3402645.

[21] G. Casteur, A. Aubaret, B. Blondeau, V. Clouet, A. Quemat, V. Pical, R. Zitouni, Fuzzing attacks for vulnerability discovery within MQTT protocol, in: 16th International Wireless Communications and Mobile Computing Conference, IWCMC 2020, Limassol, Cyprus, June 15-19, 2020, IEEE, 2020, pp. 420–425. URL: https://doi.org/10.1109/IWCMC48107.2020.9148320. doi:10.1109/IWCMC48107.2020.9148320.

[22] H. Sochor, F. Ferrarotti, R. Ramler, Automated security test generation for mqtt using attack patterns, in: Proceedings of the 15th International Conference on Availability, Reliability and Security, ARES '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1–9. URL: https://doi.org/10.1145/3407023.3407078. doi:10.1145/3407023.3407078.

[23] K. Tanabe, Y. Tanabe, M. Hagiya, Model-based testing for mqtt applications, in: M. Virvou, H. Nakagawa, L. C. Jain (Eds.), Knowledge-Based Software Engineering: 2020, Springer International Publishing, Cham, 2020, pp. 47–59.

[24] T. Dierks, C. Allen, et al., The tls protocol version 1.0, 1999.

[25] MITRE, CVE-2015-4458., Available from MITRE, CVE-ID CVE-2015-4458., 2015. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4458.