Certificate-Based TLS use and Security of Default Settings in MQTT Software

Kevin Sullivan

100896774

Supervised by Dr. David Barrera

School of Computer Science

Carleton University

COMP4905 - Honours Project

March 2021

Abstract

MQ Telemetry Transport (MQTT) is a messaging protocol that is designed for use by low-resource devices in low-bandwidth and unreliable networks. As such, it has gained popularity for use in the Internet of Things(IoT), where many devices are low-power, and network conditions vary. The MQTT organizatino describes MQTT as "Security Enabled"[MQTT, 2020], however the security of the protocol is minimal, and the MQTT specification places the burden of security on the implementer of the protocol while suggesting some security implementations[mqtt-v3.1.1, 2014]. MQTT software often has the option of using Transport Layer Security(TLS) to encrypt MQTT packets. This report aims to investigate how well existing MQTT software have implemented certificate-based TLS solutions, and whether default MQTT broker configurations can be considered safe. During this investigation it is revealed that certificate-based TLS clashes with the fundamental goals of the MQTT protocol, troubleshooting TLS issues over MQTT presents interesting challenges, and that MQTT broker default configurations are not safe. To conclude, some suggestions are made towards providing greater troubleshooting information for issues with certificate-based TLS solutions, and more secure default settings for MQTT brokers are proposed.

Acknowledgements

I would like to thank Dr. David Barrera and Hemant Gupta for their support and guidance.

Contents

List of Figures

List of Tables

1 Introduction

The Internet of Things (IoT) is the ever-growing system of physical devices that communicate with each other over the Internet. Many of these devices are small, low-power devices with little computing power, such as temperature sensors. MQ Telemetry Transport(MQTT) is a messaging protocol that is widely used by devices in the IoT (Internet of Things) due to its lightweight computing requirements.

The MQTT organization claims that MQTT is used in the following industries: automotive, logistics, manufacturing, smart home, consumer products, and transportation [MQTT, 2020]. Cyber attacks against industrial targets are increasing, and adding networked devices increases the potential attack surface. If devices in industrial networks are using MQTT for communication, there is a need to understand just how secure implementations of the protocol are.

As the number of devices in the IoT continues to grow, so does the number of devices that use MQTT. As such, there is the need to evaluate the security of the MQTT protocol so the risk posed by including devices that communicate using MQTT in a computer network is better quantifiable.

2 Background

2.1 MQTT

MQTT follows a publish-subscribe model, where MQTT clients can publish messages to and/or subscribe to receive messages from topics that are hosted

by an MQTT broker. This provides the ability to send a message to many devices that are subscribed to a topic simultaneously by having a single client publish a message to the topic.

MQTT supports username-and-password-based authentication, however MQTT communications are not encrypted. The MQTT 3.1.1 specification notes that "As a transport protocol, MQTT is concerned only with message transmission and it is the implementer's responsibility to provide appropriate security features."[mqtt-v3.1.1, 2014], and the MQTT 5.0 specification shares the same sentiment[mqtt-v5.0, 2019]. Both specifications provide some suggestions on considerations when implementing secure MQTT software, and both strongly recommend using TLS to provide security for communications. However, it is emphasized that the burden of security lies on the implementer.

2.2 MQTT Over TLS

Transport Layer Security (TLS) is a protocol that is used to encrypt application-layer data, providing confidentiality for both parties engaging in communications. Most TLS implementations make use of digital certificates, data that is used to authenticate the parties trying to communicate and to generate secrets required for encryption and decryption. MQTT packets sent over a TCP/IP connection can be encrypted using TLS. This adds an additional step to the MQTT connection process, as shown in Figure 1.

Figure 1: MQTT-Over-TLS Connection Process

As such, using TLS increases the computation and network resources used when communicating. This is incongruent with MQTT's goal to be a lightweight protocol for low-power, low-resource devices, and adds complexity to the implementation of a properly secure MQTT software.

This report will focus on the implementation of TLS using digital certificates. There are 3 ways that certificate-based TLS is implemented in the case of an MQTT system:

1. The broker has a certificate that is provided to clients that initialize the TLS handshake
    - In this case, neither the broker nor the client authenticate each other
2. Clients keep a copy of trusted certificates, and check the certificate provided by the broker against their trusted certificates
    - In this case, the client authenticates the broker
3. Clients pass a copy of their own certificate to the broker after verifying the broker's certificate, and the broker verifies that the client certificate has been signed by a trusted CA
    - In this case, the broker and the client authenticate each other

For the second and third systems certificate files must be supplied to the client. This creates additional overhead in the setup of the MQTT network, as for system 2 each new client must first be provided a copy of the broker certificates that it should trust, and in system 3 each client will need to:

be provided a copy of the trusted broker certificates, generate a signing key, generate a certificate signing request, have their certificate signing request fulfilled by a CA trusted by the broker, and then possibly provide a copy of the certificate to the broker to be added to a trust store.

As certificates are only valid for a period of time determined at the time of generation, clients and brokers will need to repeat the process when each certificate expires. This may seem like motivation to have certificates be long-lived, however long-lived certificates have been shown to be security risks.

2.2.1 TLS Error Alerts

As the TLS handshake occurs before the start of MQTT communication, troubleshooting MQTT over TLS is challenging. When an error occurs in the TLS handshake, it is almost always deemed fatal and by TLS protocol the broker and client must end their connection[Dierks and Rescorla, 2008]. Therefore, the broker and client are not able to engage in MQTT communications, requiring that errors be relayed between broker and client through the TLS header.

When an error occurs in the TLS Handshake, the party that detects the error sends a TLS error alert to the other party to notify them of the error. These error alerts have two defined levels: warning, and fatal, while some error alerts' severity is left undefined. The severity of TLS error alerts is tabulated in Table 1, where we can see that the severity of error alerts for issues with certificates is largely undefined. Furthermore, the protocol recommends that in the case that a non-fatal error does not cause termination of the

connection that the error alert is in fact not sent to the connecting party — "For example, if a peer decides to accept an expired certificate (perhaps after confirming this with the user) and wants to continue the connection, it would not generally send a certificate_expired alert."[Dierks and Rescorla, 2008].

3 Experiment Setup

Data was gathered using two computers communicating over a local network. One computer hosted two brokers: one mosquitto broker, and one HiveMQ CE broker. The other computer used mosquitto_pub and Paho clients to send messages to the brokers.

OpenSSL 1.1.1f was used to generate a CA key for the mosquitto broker, which was then used to generate self-signed x509 CA certificates and to sign broker and client certificates.
keytool was used to generate Java keystores, truststores, and certificates for the HiveMQ broker and clients.
broker 1 mosquitto version 1.6.9, Ubuntu 20.04.2 LTS 64-bit
broker 2 HiveMQ CE version 2020.2, Ubuntu 20.04.2 LTS 64-bit
client 1 mosquitto version 2.0.8, Windows Subsystem for Linux 4.4.0-19041-Microsoft 64-bit

Table 1: (Unreserved) TLS 1.2 Error Alert Severities As Defined by the Protocol

Error Alert	Severity
unexpected_message	fatal
bad_record_mac	fatal
record_overflow	fatal
decompression_failure	fatal
handshake_failure	fatal
bad_certificate	
unsupported_certificate	
certificate_revoked	
certificate_expired	
certificate_unknown	
illegal_parameter	
unknown_ca	fatal
access_denied	fatal
decode_error	fatal
decrypt_error	fatal
protocol_version	fatal
insufficient_security	fatal
internal_error	fatal
user_canceled	warning
no_renegotiation	warning
unsupported_extension	fatal

4 Tools Developed

4.1 Golang MQTT Packet Decoding

Golang's `gopacket` library provides an API for capturing and decoding packets from a network interface. The `layers` library does not contain layers for MQTT packets, so a collection of layers was created for each control packet type, as well as the MQTT fixed header.

4.2 Golang Packet Capture

To accompany the MQTT layer decoders, a CLI packet capturing utility was built in Golang.

5 MQTT over TLS using Certificates

If an error occurs in the TLS handshake due to certificates, we expect them to occur at one of the three points noted in Figure 2:

1. The client's verification of the broker's certificate has failed
2. The broker has requested the client's certificate, but client did not provide a certificate to the broker
3. The broker's verification of the client's certificate has failed

If implementing TLS for the MQTT network is hard to troubleshoot, it is more likely that it will be implemented incorrectly, or that it will not be implemented at all.

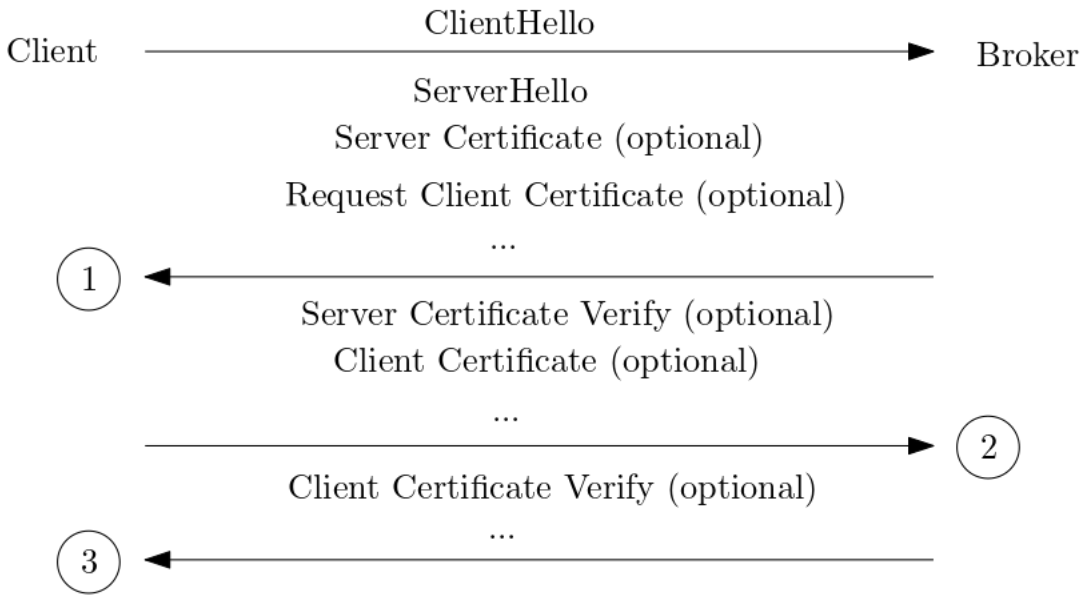

Figure 2: Possible Error Locations Due to Certificates in the TLS Handshake

5.1 Relaying Error Messages Through an Unencrypted Listener

One option to relay additional information to the client in the case that the TLS handshake in unsuccessful is for the broker to publish error messages and troubleshooting information to some unencrypted listener. As defined in the MQTT specifications, MQTT topics that begin with the character "$" are used by the broker to publish information on the broker's state, such as the broker's uptime, system time, and the broker version, though the exact topics that each broker implementation should conform to is still in discussion [MQTT Community, 2021]. The MQTT specification also requires that clients be unable to use "$" topics for their own purposes, i.e. clients may not publish to these topics. As such, a "$" topic, for example `$tls-errors`,

could be a good candidate for the broker to publish TLS troubleshooting information for clients. For the following examples, consider the top-level topic to be .

`<client IP>`: Given that we do not want attackers to be able to map the MQTT network simply by subscribing to all topics beginning with "$", this would force implementing a dynamic topic access control list on the broker that compares the IP of the client sending the subscribe packet against each topic. Also, if a user is connecting multiple clients to the broker it may be difficult to differentiate troubleshooting messages between clients. If the error occurred during the verification of the client's certificate by the broker,

5.2 Notify Expiration Continue Connection

6 Maximum MQTT Packet Size

Mentioned in the MQTT specifications, an MQTT control packet's greatest possible size is 268, 435, 455 bytes (256 MiB less one byte). The only types of control packets that can achieve this length within the rules of the protocol are PUBLISH packets or SUBSCRIBE packets. The greatest possible payload for a MQTT 3.1.1 PUBLISH packet would be: 268, 435, 455 minus 1 byte for the packet type and flags, 4 bytes for the 'Remaining Length', 2 bytes of the topic name length, and at minimum 1 byte of topic name, resulting in a payload of 256 MiB minus 9 bytes.

The greatest possible payload for a MQTT 3.1.1 SUBSCRIBE packet would be: 268, 435, 455 minus 1 byte for packet type and flags, 4 bytes for the 'Remaining Length', 2 bytes for the packet identifier, resulting in 256

MiB minus 8 bytes to be used to encode topic subscriptions. Each topic subscription requires 2 bytes for the topic name length, up to 65535 bytes for the topic name, and 1 byte for the desired QoS — allowing for at most 4095 maximum-size topic subscriptions. In the case of MQTT 5.0, these sizes are equal or lesser given that the variable header of each packet may contain bytes representing packet properties.

Given that MQTT is meant to be a lightweight protocol that uses few network resources, we would expect that MQTT clients and brokers will be sending control packets that are much smaller than the maximum packet size. Therefore, it would be reasonable that MQTT brokers enforce a default maximum packet size that is smaller than the maximum packet size to ensure that malicious actors cannot cause network congestion by sending many large packets to the broker.

However, we see that in the case of both HiveMQ and mosquitto the default setting is that the maximum packet size is 256 MiB. Both softwares also allow for a maximum of 1000 queued messages per client.

6.1 MQTT Message Queueing Attack

An attacker could disrupt the broker by:

1. Creating a persistent session between any number of clients and the broker by having each client subscribe to any number of distinct topics with either QoS 1 or 2

2. Disconnecting the clients so that the messages sent to the subscribed topics are queued by the broker for later delivery to the offline clients

3. Using a different client (or clients) to publish up to 1000 maximum-sized messages to each topic that the clients previously subscribed to

Depending on the resources of the machine hosting the broker, simply keeping the queued messages could result in depleting the resources allocated to the broker's process as 1000 full-size messages requires 256,000 MiB, or approximately 268 GB of memory, and the defaultt settings allow for 1000 queued messages per client.

In the case that the broker is able to store all of the queued messages, we can then slow down communication between the broker and its non-malicious clients by reconnecting our clients to the topic that they subscribed to. Since the default for both mosquitto and HiveMQ is to never expire client sessions, we can reconnect our clients and begin to receive the queued messages at any time. mosquitto applies a default of a maximum of 20 messages to be inflight at a time with no maximum on the amount of inflight bytes, so if we have enough malicious clients we can request that mosquitto attempt to send 5120 MiB of packets to our clients. HiveMQ's documentation does not clearly state if it is possible to limit the amount of outgoing messages[HiveMQ-v4.5, 2021]. Given that MQTT is meant for low-bandwidth networks, this could take a significant amount of time.

In addition, as these are QoS 1 or 2 messages the broker will keep attempting to publish the message to the subscribing clients until the clients acknowledge the publish with a PUBACK control packet. We could have our malicious clients never respond with the PUBACK, forcing the broker to continue to retry publishing forever.

6.2 Mitigating the MQTT Message Queueing Attack Through Safe Defaults

7 Future Work

Retrieving TLS session keys from local MQTT traffic for debugging purposes?

References

[Dierks and Rescorla, 2008] Dierks, T. and Rescorla, E. (2008). RFC 5246 The Transport Layer Security (TLS) Protocol Version 1.2 - Section 7.2.2: Error Alerts. `https://tools.ietf.org/html/rfc5246#section-7.2.2`. [Online; accessed 12-March-2021].

[HiveMQ-v4.5, 2021] HiveMQ-v4.5 (2021). HiveMQ User Guide / Restrictions / Throttling bandwidth. `https://www.hivemq.com/docs/hivemq/4.5/user-guide/restrictions.html#throttle-bandwidth`. [Online; accessed 23-March-2021].

[MQTT, 2020] MQTT (2020). MQTT: The Standard for IoT Messaging. `https://mqtt.org`. [Online; accessed 11-March-2021].

[MQTT Community, 2021] MQTT Community (2021). MQTT Wiki/ SYS Topics. `https://github.com/mqtt/mqtt.org/wiki/SYS-Topics`. [Online; accessed 24-March-2021].

[mqtt-v3.1.1, 2014] mqtt-v3.1.1 (2014). MQTT Version 3.1.1. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. [Online; accessed 11-March-2021].

[mqtt-v5.0, 2019] mqtt-v5.0 (2019). MQTT Version 5.0. https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html. [Online; accessed 11-March-2021].