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

April 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. Most topics hosted on the broker are not pre-determined, instead they can be generated on-the-fly as clients publish and subscribe to them allowing for a great deal of flexibility.

MQTT also supports messaging with Quality of Service(QoS) - 0 for "at most once" delivery, 1 for "at least once" delivery, and 2 for "exactly once" delivery. This allows devices that communicate over unreliable networks to ensure that messages that they send to the broker are delivered to their target, or that messages sent to the topic that they are subscribed to reach the client.

In terms of built-in security, MQTT supports username-and-password-based authentication, however MQTT communications are not encrypted. This allows for sniffing of authentication credentials much like plain HTTP, as the MQTT packets are sent through the air or over the wire in the clear. 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 Fig. 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 cer-

tificates. 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. For example, the compromise of a root-level certificate authority like DigiNotar caused significant disruption as all certificates issued by DigiNotar, malicious or not, had to somehow be actively revoked, as at the time certificates were issued for multiple years [van der Meulen, 2013].

Though an MQTT network of self-signed certificates is at a much smaller scale than an international certificate authority, the issues caused by a compromise of the certificate chain are similar. If a broker certificate were compromised, this could lead to a attacker-in-the-middle attack where an attacker can set up a bridge with the compromised certificate that clients trust so that all of the traffic is received by the attacker before it is forwarded to the broker. Compromise of client certificates leads to attacks against the broker. To be able to react to a compromise of a certificate or certificate authority, the MQTT network administrators would need to implement some Online Certificate Status Protocol (OSCP) solution, or provide every device in the MQTT network with a Certificate Revocation List(CRL). This further increases the resources required to use and maintain the MQTT network.

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 Tools Developed

3.1 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.

3.2 Golang Packet Capture

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

4 Troubleshooting MQTT Over TLS

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

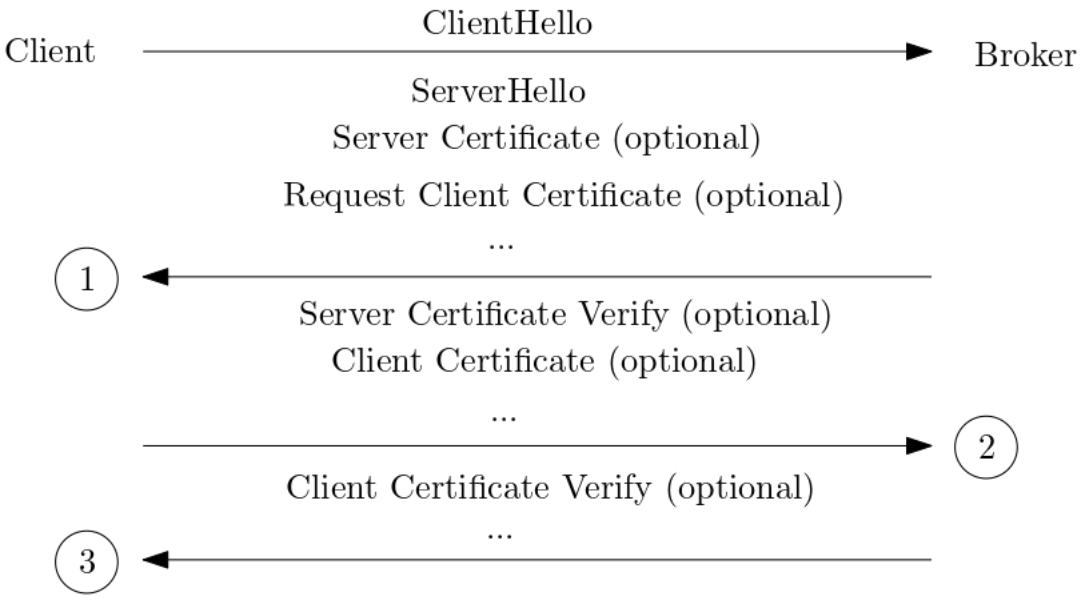

FIGURE 2: Possible Error Locations Due to Certificates in the TLS Handshake

Errors can be uncovered at the broker or at the client, and can be due to data originating from the broker, client, or the certificate authority. As such, the broker and the client must be providing robust troubleshooting information to each other so that the software user can determine what has caused the error, who has caused the error, and feels informed enough to correct the error.

As has been previously discussed, implementing security for MQTT is already conflicting with MQTT's overall goals of lightweight, reliable communication. Implementing TLS already requires additional network resources to do the TLS handshake and verify certificates, additional device resources to store certificates and keys, and additional oversight to monitor the sta-

tus of the certificates. If troubleshooting MQTT over TLS is difficult, it is more likely that it will be implemented incorrectly, or this could further convince the MQTT network implementer that the cost of implementing TLS is greater than the risk of a poorly-secured network.

4.1 MQTT Software Error Messages

To determine the effectiveness of the error messages generated by MQTT software, 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. Certificates were generated for the mosquitto and HiveMQ brokers using the tools recommended by the developing organizations — OpenSSL for the mosquitto broker, and keytool for HiveMQ. 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 certificate-signing requests that were also generated using OpenSSL. keytool was used to generate Java keystores, truststores, and certificates for the HiveMQ broker, and clients generated their own certificates using OpenSSL that were added to the truststore using keytool.

The mosquitto broker was using mosquitto version 1.6.9, and the HiveMQ broker was using HiveMQ CE version 2020.2. Both brokers were hosted on a laptop running Ubuntu 20.04.2 LTS 64-bit. The clients were generated using mosquitto_pub and mosquitto_sub version 2.0.8, and paho.mqtt.golang

v1.3.2, running through Windows Subsystem for Linux 4.4.0-19041-Microsoft 64-bit.

Table 2 lays out the errors we would expect to receive due to certain certificate errors based on the TLS 1.2 specification. One error of note occurs on row 4: certificates contain an attribute called the "common name", which should be an IP address or fully-qualified domain name (FQDN) that matches the address that the client is attempting to connecting to. The client checks this during the certificate verification in the case that there are no other higher-priority IDs that can be verified first, which will depend on the certificate [Saint-Andre et al., 2011]. In the case of a self-signed certificate, this is likely the case.

We see in Table 3 that the mosquitto broker opts to simply echo the error that is provided by OpenSSL. This provides some useful information in the case of rows 2, and 3, and arguably misleading information on row 4. "alert certificate expired" and "alert unknown ca" could be considered actionable - however the definition of internal error is defined in the TLS1.2 specification as " An internal error unrelated to the peer or the correctness of the protocol (such as a memory allocation failure) makes it impossible to continue."[**?**]. This would suggest that the error is not due to any issue with the certificate or with the TLS protocol. In addition, these errors are not plainly human-readable, containing long error numbers and OpenSSL internal function names.

The clearest error message occurs on row 5, where the error text clearly states that the peer (the client) did not return a certificate.

The errors on rows 6 and 7 include the function name `tls_process_client_certificate`, and the message "certificate verify failed", which lets the user know that there was an issue with the client certificate, but not what the issue was. If we capture the packets that cause this error, we see that despite the TLS alert error being set in the TLS headers, this information is not relayed through the broker's error message.

We see in Table 4 that the mosquitto client opts to let the user know that the client has found an error in a certificate by printing "Error: A TLS error has occurred" to the terminal, and it lets the user know that the broker has found an error in a certificate by printing "Error: The connection was lost" to the terminal. If we capture the packets that are sent to and by the client in the TLS handshake, we see that even the meager information contained in the TLS error alert string is not passed to user.

4.2 Parsing the TLS Layer

As we see

4.3 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

[1] mosquitto_pub and mosquitto_sub require that server certificates are verified by clients

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. A challenge presents itself in how the information can be published without compromising the security of the client. We consider some possible sub-topics as follows:

- `<client IP>`: The client's IP should uniquely identify their device, so the device can subscribe to this topic without any additional information from the broker. However, 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 device is connecting multiple clients to the broker it would be difficult to differentiate troubleshooting messages between clients.

- `<client certificate hash>`: If the MQTT broker is requesting a client certificate to authenticate the client, it could publish troubleshooting information on a subtopic of the client certificate's hash value. The

client has a copy of their own certificate, so they are able to subscribe to the topic without additional information from the broker as long as both the client and the broker are using the same hashing function. With a good hash function we could expect that the hash of two different client certificates be distinct. However, this requires that a device store as many certificates as clients that they plan to connect to the broker. Of course, this solution only applies for systems where the broker requests a certificate from the client.

4.4 Notify Error Continue Connection

As noted in section 2.2.1, the error severity for the `bad_certificate`, `unsupported_certificate`, `certificate_revoked`, `certificate_expired`, and `certificate_unknown` errors is undefined. As such, the MQTT software may make the decision that an such a certificate is not cause for terminating the connection. Though it is suggested that if the connection will not be terminated the error alert should not be sent at all, for troubleshooting purposes this report will recommend sending the error alert so that there is a record for the broker and the client that the certificate in question has some issue.

However, given that the MQTT software is not terminating the connection if certificates are expired, it would be recommended that the software be able to reject these connections if the certificate has been revoked. Without rejecting connections made using revoked certificates, there is no recourse to

a certificate being compromised short of creating a new CA certificate so that connections using the compromised certificate result in a fatal unknown_ca error.

5 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.

5.1 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 keep-

ing the queued messages could result in depleting the resources allocated to the broker's process. A single client is permitted under the default settings to queue 1000 full-size messages, which would require 256,000 MiB, or approximately 268 GB of memory. This single client's 1000 256 MiB messages may be sufficient, however, by default, each client is permitted up to 1000 queued messages.

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, 2021b]. Given that MQTT is meant for low-bandwidth networks, this could cause a significant amount of congestion.

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.

5.2 Mitigating the MQTT Message Queueing Attack Through Safe Defaults

A way of lessening the effect of this attack (without implementing authorization) is to lower the default maximum packet size. The optimal maximum packet size will depend on the makeup of the MQTT network, however it is unlikely that information sent with a protocol built for telemetry between low-resource devices on low-bandwidth networks would require an initial cap of 256 MiB. First we can consider that the largest fixed header is 5 bytes - 1 byte for the packet type and flags, and 4 bytes for the remaining length. The largest possible variable header occurs for a MQTT 5.0 PUBLISH packet - 2 bytes for the topic length, 65535 bytes of topic string, 2 bytes of packet identifier, 3 bytes of properties length, 1 byte of payload format indicator, 4 bytes of message expiry interval, 65537 bytes of content type, 65537 bytes of response topic, 65537 bytes of correlation data, 4 bytes of subscription identifier, 2 bytes of topic alias, and 65537 bytes of user property for a total of 327,701 bytes (approximately 320 KiB).

As an initial experiment, MQTT traffic from a smart home was captured and the payload length of each packet examined using `mosquitto_sub`. The greatest payload length seen was less than 8 KiB. The traffic was MQTT 3.1.1 traffic, which does not allow for properties in the variable header (which account for 262,162 of the 327,701 bytes), however we will consider the case of the MQTT 5.0 variable header. Given the maximum sized fixed header and variable header previously calculated, the greatest MQTT packet seen

was approximately 328 KiB. This means that the payload could increase by 32,768 times, and only just reach the maximum packet size. Of course, this is a sample of a few consumer products that are operating in a household, and IoT devices operating in industrial settings may have greater payloads. However, it does not seem unreasonable to conclude that a margin of over 32,000 times the packet size is necessary or safe. A smaller default maximum packet size could reduce the impact of the message queueing attack by a large factor, while still allowing for significant fluxuation of the size of legitimate packets.

mosquitto offers the max_queued_bytes option, which allows the broker to limit how many bytes worth of packets can be queued at a time. This defaults to no limit[Light,]. Instead of limiting the queue to 1000 packets by default, it would be more prudent to limit the amount of bytes in the queue as the amount of bytes has the greater potential of causing disruption. A cap on the amount of queueable bytes may result in fewer queueable messages than the 1000-message cap, however if devices on the network are relying on megabytes or gigabytes of queued messages to be delivered on connection, it could be said that a lightweight message protocol is not a suitable platform for the system.

HiveMQ does not seem to offer any limit on the amount of bytes queued, only the number of queued messages[HiveMQ-v4.5, 2021a]. Based on the sample data gathered from the smart home devices, MQTT packet payloads can vary from just a few bytes to multiple kibibytes, and the publishing interval of devices varies. A queue of 1000 messages could be filled in a few

minutes by a device that publishes small packets every few seconds, whereas a byte-limited queue could retain many of these publications. Allowing the broker to have a byte-capped queue size is not only a safer option, but likely a more usable option.

6 Future Work

To better define the values of safe MQTT broker defaults, more information gathering should be done on the packet sizes and packet queueing requirements of various devices or software that use MQTT to communicate.

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, 2021a] HiveMQ-v4.5 (2021a). HiveMQ User Guide / General Configuration Information. `https://www.hivemq.com/docs/hivemq/4.5/user-guide/configuration.html`. [Online; accessed 23-March-2021].

[HiveMQ-v4.5, 2021b] HiveMQ-v4.5 (2021b). 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].

[Light,] Light, R. mosquitto.conf. https://mosquitto.org/man/mosquitto-conf-5.html. [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].

[Saint-Andre et al., 2011] Saint-Andre, P., Cisco, Hodges, J., and PayPal (2011). RFC 6125 Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). https://tools.ietf.org/html/rfc6125#page-28. [Online; accessed 31-March-2021].

[van der Meulen, 2013] van der Meulen, N. (2013). DigiNotar: Dissecting the First Dutch Digital Disaster. *Journal of Strategic Security*, 6:46–58. [Online; accessed 25-March-2021].

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

TABLE 2: Expected TLS Error Alerts for Various Certificate Configurations

CA Certificate	Server Certificate	Server Verification Certificate	Client Certificate	Expected TLS Error Alert
valid	valid	not required	not required	none
valid	expired	valid	not required	certificate_expired
valid	unrecognized CA	valid	not required	unknown_ca
valid	unresolvable CN	valid	not required	certificate_unknown
valid	valid	valid	required; none	handshake_failure
valid	valid	valid	required; expired	certificate_expired
valid	valid	valid	required; unrecognized CA	unknown_ca

TABLE 3: Error Messages Generated by mosquitto(broker) Due to Certificate Errors

CA Certificate	Server Certificate	Server Verification Certificate	Client Certificate	Error Message
valid	valid	not required	not required	none
valid	expired	valid	not required	OpenSSL Error[0]: error:14094415:SSL routines: ssl3_read_bytes:sslv3 alert certificate expired
valid	valid	incorrect CA	not required	OpenSSL Error[0]: error14094418:SSL routines: ssl3_read_bytes:tlsv1 alert unknown ca
valid	unresolvable CN	valid	not required	OpenSSL Error[0]: error14094438:SSL routines: ssl3_read_bytes:tlsv1 alert internal error
valid	valid	valid	required; none	OpenSSL Error[0]: error:1417C0C7:SSL routines: tls_process_client_certificate: peer did not return a certificate
valid	valid	valid	required; expired	OpenSSL Error[0]: error:1417C086;SSL routines: tls_process_client_certificate:certificate verify failed
valid	valid	valid	required; unrecognized CA	OpenSSL Error[0]: error:1417C086;SSL routines: tls_process_client_certificate:certificate verify failed

TABLE 4: Error Messages Generated by mosquitto_pub & mosquitto_sub Due to Certificate Errors

CA Certificate	Server Certificate	Server Verification Certificate	Client Certificate	Error Message
valid	valid	not required	not required	Error: A TLS error occurred[1]
valid	expired	valid	not required	Error: A TLS error occurred
valid	valid	incorrect CA	not required	Error: A TLS error occurred
valid	unresolvable CN	valid	not required	Error: A TLS error occurred
valid	valid	valid	required; none	Error: The connection was lost
valid	valid	valid	required; expired	Error: The connection was lost
valid	valid	valid	required; unrecognized CA	Error: The connection was lost

TABLE 5: Error Messages Generated by HiveMQ CE Due to Certificate Errors

CA Certificate	Server Certificate	Server Verification Certificate	Client Certificate	Error Message
valid	valid	not required	not required	none
valid	expired	valid	not required	SSL handshake failed
valid	valid	incorrect CA	not required	SSL handshake failed
valid	unresolvable CN	valid	not required	SSL message transmission failed
valid	valid	valid	required; none	SSL handshake failed
valid	valid	valid	required; expired	none
valid	valid	valid	required; unrecognized CA	SSL handshake failed