Messaging Queues in the IoT under pressure

Stress Testing the Mosquitto MQTT Broker

Patrik Fehrenbach
Fakultät Informatik
Hochschule Furtwangen University
{patrik.fehrenbach}@hs-furtwangen.de

***Abstract*—MQTT is the *de facto* industry standard for publish-subscribe-based messaging, as it becomes widely adopted, the risk of it being attacked by cyber criminals rises. This paper tries to elaborate whether it is possible to exhaust the resources of a commonly used MQTT broker using a newly developed tool. The aim of the tool is to conduct several test cases with a continuous complexity in order to simulate a Distributed Denial of Service (DDoS) attack. The paper will additionally discuss the conducted test cases as well as the corresponding results and resulting statistics covering CPU, Processes and System performance. The test cases included the performance of TLS connections vs. non TLS connections, as well as the usage of different Quality of Service levels.**

I. Introduction

Nowadays the term *IoT* (Internet of Things) is a commonly used buzzword, it is about making various consumer electronic devices smarter by connecting them to the internet and to cloud services. MQTT (Message Queue Telemetry Transport) is today one of the most popular messaging protocols for use on the top of the TCP/IP protocol. It is convenient for real-time communication between embedded devices and services. MQTT was invented in 1999 by IBM and was initially only internally used by them. In 2010 IBM released version 3.1 of MQTT royalty free. Since then several companies have adopted MQTT to their needs. One of the most famous applications using MQTT is the Facebook Messenger. The MQTT protocol was able to tackle the most common problems the developers were facing such as bandwidth and battery lifetime [1]. The rising development lead to the introduction of numerous MQTT brokers such as the most popular and widest used MQTT brokers: HIVEMQ[2], RabbitMQ [3] and the open source Mosquitto [4].

In our digital and connected world IT-Security is preset every day, it doesn't matter whether you are purchasing something online, making a payment with online banking or accessing personal documents on the internet. The main idea of the Internet of Things (IoT) is to provide a connection to every single device of your daily use. While all of these devices can hold some sort of confidential and personal information, you do not want to share with everyone. Therefore it is important to focus on the protective pillars of information security: **confidentiality**, **integrity** and **availability**. [5]. In autumn 2016 news about the Mirai-Botnet what affected more than 49.657[6] Internet of Things (IoT) hosts appeared. Closed Circuit Television (CCTV) cameras besides of Digital Video Recorders (DVRs) and routers with low or none security configurations were the main target of the spreading malware. The Mirai malware spread itself by performing dictionary attacks [6] which is a subcategory of the Brute-Force method, to gain access to devices. After the malware gained access on the target system, the infected systems were used to start Distributed Denial of Service (DDoS) attacks against several IP ranges to force a shutdown of network system components due to a massive amount of requests sent over the Internet. Due to a research done by Securelist [7], DDoS attacks have been almost doubled in 2017. The most widely used techniques for those large scale attacks were syn (sending multiple TCP SYN Messages, so called SYN-Flood attack), TCP (Transmission Control Protocol) based attacks, UDP (User Datagram Protocol), HTTP (Hypertext Transfer Protocol) and ICMP (Internet Control Message Protocol) based attacks, however as the researchers stated: "complex attacks that can only be repelled with sophisticated protection mechanisms are becoming more frequent."[7] which likely predicts a new area of targeted, highly complex attacks. Since IoT and Industry 4.0 is on its way, or already there, DDoS attacks will likely adopt to IoT protocols and target large scale industry applications. Therefore it is inevitable to test the one of the main pillars of it-security, the **availability** of IoT based protocols. The aim of this paper is to provide a Proof of Concept tool (PoC) which tries to simulate a DoS (Denial of Service) attack on the common used open source MQTT broker Mosquitto. To perform a Denial of Service attack, a tool has been developed which tries to evaluate whether TLS connections and the QoS level have a significant overhead on the performance of the broker.

II. Related Work

In terms of availability and load testing for MQTT based Brokers there has been quite a lot of research over the last years. The main purpose of the current research however was the scalability and the number of concurrent client connections to a MQTT Broker. A recently published paper by Scaleagent [8] analysed five different MQTT brokers and their capabilities of handling concurrent connections (Number of publishers (connections) = delivered throughput (messages per second[8]). The researchers showed that the Mosquitto Broker was capable of handling up to 4000 publishers at a time, above that mark Mosquitto was not able to keep up

with the incoming messages. The RabbitMQ Broker [3] had similar results but with an exhaustive memory usage. The JoramMQ broker was capable of processing almost 50.000 messages per second. A publication by HiveMQ [9] analyzed the performance implication of TLS on the HiveMQ broker, and found out that TLS affects the performance significantly especially during the handshake of the TLS connection, after the clients have exchanged the certificates, the overhead is negligible [9]. For the Quality of Service level, S. Lee of the Dept. of Computer Engineering Keimyung University Daegu analyzed in the Paper On *Correlation analysis of mqttloss and delay according to QoS level* [10] the implications of the usage of various QoS levels to analyze end-to-end delays and message loss. In terms of available scalability measuring tools, the most common one is Malaria [11], the tool was considered to be used for this analysis, however TLS encrpytion is not yet supported by the project. Malaria is based on the Paho MQTT Implementation. Another application suitable of performance measuring is MQTTBox, in the testing environment the load testing was not stable and prone to errors/false positives. A stable too both capable of benchmarking MQTT and TLS support is *mqtt-bench* [12] developed by Takanori Suzuki of Acroquest Technology [12]. The tool is developed in GoLang language, unfortunately the implementation was slower than the Python library.

III. MQTT

The following chapter will give a brief overview about the basic components of the MQTT protocol. The first subsection will give a brief overview about the MQTT broker and it's capabilities. The second subsection will cover the message types supported by MQTT and the offered Quality of Service Levels.

A. MQTT Broker

The main part of the MQTT protocol is the so called MQTT Broker. Based on the implementation MQTT brokers are theoretically able of handling up to 45 thousands concurrent connected MQTT clients[6]. The main tasks for the MQTT broker are:

- receiving messages
- filtering messages
- sorting messages
- sending messages to clients
- Authentication/Authorization

The broker (in this case the Mosquitto Broker) is capable of receiving messages on a variety of channels. By default the Mosquitto broker listens for unauthenticated on port *1883*. Plaintext MQTT messages contain of a fixed-length header of 2 Bytes and an optional message-length header and message payload. These specifications make MQTT a reliable and fast protocol which is optimized for low bandwidth and unreliable networks. The MQTT TCP format, as can be seen in figure Figure 1.

The first eight bits are the fixed MQTT header, where the first four bits represent the Message Type, the fifth is

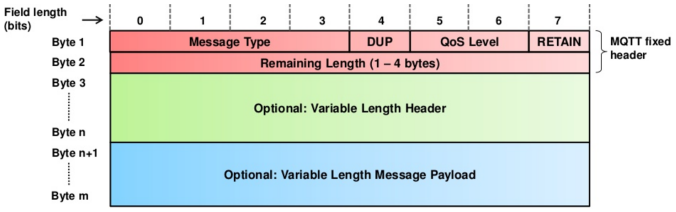

Figure 1. MQTT Message Header

responsible for the DUP and sixth and seventh for the QoS (Quality of Service) level, and the eighth for the so called retain message. The second 8 bits are reserved for the variable length header, and the optional header which can be used for TLS payloads.

B. MQTT Message Types

The following subsection will give a brief overview about the supported control packet types of MQTT.

The *Publish* message (as the name implies) is used by a sender to publish messages. Each message must contain a topic, which will be used by the broker to forward the message to interested clients. Each message typically has a payload which contains the actual data to transmit in byte format (s. Figure 1. On QoS 0 messages, this is the only payload sent. For all the remaining QoS levels, the *Puback* message is used to acknowledge the receive of a message by the broker (Especially for QoS1). A *Pubrec* Packet is the response for a QoS2 *Publish* Packet. It is the second packet of the QoS 2 protocol exchange as can be seen in Figure 2. A *Pubrel* Packet is the response to the perviously described *Pubrel* Packet, and is the third packet of the QoS 2 protocol exchange. The *Pubcomp* Packet is the response to a *Pubrel* Packet, and it is the fourth and final packet of the QoS 2 protocol exchange [13]. In addition to these control packets, there are several messages which indicates the status of the broker, subscribe/unsubscribe messages and disconnect notification. These message types however won't be covered in this paper.

C. Quality of Service

MQTT supports three levels of Quality of Services(QoS)[14]. The QoS ensures the reliability of messaging of the MQTT protocol as seen in figure Figure 2 messages with a QoS level of 0 only consist of a *Publish* message, with *QoS0* the message is submitted "at most once"[15]" and by default not stored on the server. The delivery of the packet is not acknowledged by the client or the server. Due to this "fire and forget"[15]" character, this is the fastest mode of transferring packets. The QoS Level 1 consist of a *Publish* and *Puback* message, which ensures that the message ist delivered at least one time. If there is a connection problem, the client will re-send the message until the *Puback* message of the server acknowledges the delivery of the packet, due to this the message is stored locally by

the client until the connection is finished. The highest level of QoS (*QoS2* consists of a *Publish*, *Puback* and additionally a *Purel* and a *Fubcomp* answer. The message is delivered exactly one time, similar to QoS1 the message must also stored locally by the sender, until the server confirms the delivery of the message. The QoS2 adds additional complexity to the protocol since the connection is only completed if the publish message is sent to the receiver. The additional handshake and acknowledge routine ensures that the message is only sent unique and no duplication of messages occur.

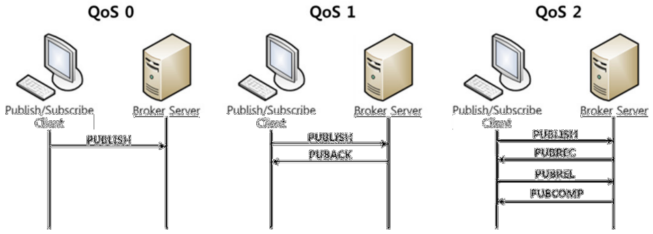

Figure 2. Quality of Service [10]

IV. MQTT Security

The following section introduces the main capabilities in terms of security of the Moqsuitto broker. The first subsection will cover the username/password authentication mechanism, where the second part will cover the TLS based encryption.

A. Username / Password Authentication

MQTT provides authentication using a Username and Password combination similar to basic authentication or crypt [16]. By default the authentication is sent in plaintext, security wise this means that an attacker with a MitM (Man in the Middle Connection) would be able to intercept the connection and obtain the credentials. Therefore it is recommended to supply a TLS based encryption on every username/password based MQTT authentication.

B. TLS Encryption

By enabling TLS encrpytion on MQTT it is possible to ensure that MQTT messages are send cryptographic encrypted. By providing a client with a signed digital certificate, the client is able to authenticate on the MQTT broker. By default, the MQTT Mosquitto Broker uses Port 8883 for encrypted communications. In theory MQTT supports two methods of encryption Pre-Sharey Keys (TLS-PSK) and Secure Socket Layer (TLS/SSL). The test cases of this paper will only focus TLS/SSL connections hence the used libraries do not have support for TLS-PSK. The TLS/SSL authentication is based on an asymmetric encryption using a public certificate for authentication, and a private certificate on the broker. For the authentication of the client a X509 certificate is used, this certificate ensures that the identity of the client can be ensured by broker.

V. Proof of Concept

The following section will cover the capabilities of the developed Python scripts and capabilities. The tools are divided in Publisher as seen in Section V-B and Subscriber in Section V-C.

A. Motivation

As already discussed in Section II, very few research has been conducted for stress testing / performance measuring of the MQTT protocol. As most of the found tools, didn't met the requirements for the upcoming test cases a standalone script has been developed. For the proof of concept, a Python based tool has been implemented. For the development of the tool the open source library Paho MQTT [17] has been used. The Paho Library provides all the necessary functionalities such as TLS encryption and the needed functions to send MQTT packets. The main aim of the tool is to send as many concurrent messages as possible, by the time paper was written the tool was capable of sending 8000 messages/second unencrypted. To provide a high complexity, the tool is capable of sending randomized messages, randomly selected QoS levels (0-2) and a randomly chosen MQTT client Identifier. Additionally, each publisher generates a new session session on the broker using the TLS connection, this ensures a realistic scenario. The current script connects once to the broker, sends (as defined in the for loop line 28) 1000 messages to the channel *test* with a random chosen message. The pseudo random message is generated using the random [18] library, and generates a 32-character hexadecimal string.

B. Publisher Script

```
1  import paho.mqtt.client as mqtt
2  import ssl
3  import uuid
4  import random
5
6  message = uuid.uuid4().hex
7
8  def on_connect(client, userdata, flags, rc):
9      m = "Connected flags"+str(flags)+"result code "\
10     + str(rc) + "client1_id  "+str(client)
11     print(m)
12
13 def on_message(client1, userdata, message):
14     pass
15
16 while 1:
17         try:
18             broker_address = "raspberrypi.local"
19             client1 = mqtt.Client(message)
20             client1.on_message = on_message
21             client1.on_connect = on_connect
22             client1.tls_set("ca.crt")
23             client1.tls_insecure_set(True)
24             client1.connect(broker_address,8883)
25             client1.loop_start()
26             counter = 0
27
28             for n in range(0, 1000):
29                     message = uuid.uuid4().hex
30                     qos = [0,1,2]
31                 client1.publish("test", message,
    qos=random.choice(qos))
32                     counter += 1
```

```
33              client1.disconnect()
34              client1.loop_stop()
35
36          except socket.gaierror, err:
37              print err
38              pass
39
40
41  def fname(arg):
42      pass
```

Listing 1. MQTT DoS Proof of Concept

The tool is capable of covering the following scenarios and test cases:

- TLS Encrypted Messages
- Random QoS Level
- Random Client Identifier
- Random Publish Message
- Fixed Channel

The tool was written in order to answer the following two hypothesis:

- Does the usage of TLS add an additional overhead to system resources?
- Does the level of QoS add an additional overhead to system resources

C. Subscriber Script

In addition to the publisher script, a subscriber script has been developed. The aim of the subscriber script is to create 100 publishers on the channel the publisher is publishing. For an additional complexity, each connection of the subscriber using TLS will generate a unique TLS session which should cause a higher load on the broker. As defined in line 1) the numbers of subscribers can be adjusted, for the test cases 100 concurrent subscribers are active.

```
1  for n in range(0, 100):
2
3    print "Connecting Senders \%d" \% n
4    broker_address = "192.168.0.39"
5    client1 = mqtt.Client(message)
6    client1.on_message = on_message
7    client1.on_connect = on_connect
8    client1.tls_set("ca.crt")
9    client1.tls_insecure_set(True)
10     client1.connect(broker_address,8883)
11   client1.publish("test", message, qos=0)
12   clients.append(client1)
```

Listing 2. MQTT Subscriber Proof of Concept

VI. Setup

The following section will describe the technical setup used to simulate a DDoS attack on the Mosquitto MQTT broker. The first part will focus on the used operating systems and environments, where the second part will cover the measurement of the CPU exhaustion and the used tools.

A. Technical Setup

For the technical setup a Raspberry Pi 3 with a Broadcom BCM2837 Chipset, a 4x ARM Cortex-A53, 1.2GHz CPU and 1GB of LPDDR2 (900 MHz) RAM with a 10/100 Ethernet network running Raspbian GNU/Linux 8 (jessie) and the mosquitto version 1.4.11 (build date Mon, 20 Feb 2017 22:47:27 +0000) has been used. On the operating system, no additional but Mosquitto were installed, additionally no Window manager has been installed to reserve as much CPU and RAM as possible. With a running but idling mosquitto server, 5.4% of the available RAM are used (86MB) and about 0.7% of the CPU are used for system tools. The setup consists of one or many sender (in this case up to 100) and a receiver, iMac late 2015) both connected via Ethernet to the router. The receiver (MQTT broker) is also connected via Ethernet to the router. The average Round-Trip-Delay (RTD) for the connection is 5ms. For the measurement of the CPU exhaustion on the broker, the tool *vmstat* has been used. While the sender is constantly sending packages, the overall performance of the CPU has been recorded every second for 60 seconds with `vmstat160`. The broker is running the mosquitto software with `sudomosquitto-c/etc/mosquitto/mosquitto.conf-v` the only additional changes made to the configuration of mosquitto was the additional usage of TLS and the corresponding port 8883 as can be seen on Listing 1 in Section IV. The sender script requires only on the paho mqtt library and the corresponding certificate for the TLS connection, additional the system limits of concurrent connections had to be increased as described in Section IX-A. The receiver PC subscribes to the channel *test* and listens for incoming connections using the MQTT tool `mosquitto_sub-t"test"-h192.168.0.39` where `-t` is the channel, and `-h` is the host. The total amount of sent messages is counted with the following channel provided by the broker. `\$SYS/broker/messages/received` The total number represents the messages of any type received since the broker started, additionally the amount of dropped messages has been monitored using the topic `\$SYS/broker/messages/publish/dropped`. To ensure all messages sent by the broker are received by the subscriber the topic `\$SYS/broker/messages/publish/sent` has been compared to the messages the script sent. The TLS configuration has been configured on the broker as can be seen on in Listing 2. For the X509 Certificate the SHA256 cipher has been selected.

```
1  listener 8883
2  cafile /etc/mosquitto/certs/ca.crt
3  certfile /etc/mosquitto/certs/raspberrypi.local.crt
4  keyfile /etc/mosquitto/certs/raspberrypi.local.key
5  require_certificate true
6  allow_anonymous true
```

Listing 3. Mosquitto MQTT over TLS/SSL configuration

VII. Test Cases

The following section will describe the conducted test cases. The first subsection describes the obtained QoS test cases,

where the second one covers the TLS based test cases.

A. QoS Test Cases

The first test case contained of one sender sending for each connected client 1000 Messages for 60 seconds to the broker. In the first three rounds, the QoS Level was increased from 0 to two. In an additional test a combination of randomly chosen QoS levels was conducted, this additional test tries to evaluate the assumption whether an attacker is capable of stressing the broker with a rapidly switching QoS level. All QoS related test cases were conducted with one sender, and n subscribers (in this test case, 100 concurrent subscribers) on the same network layout as described in Section VI-A.

B. TLS Test Cases

The first test contained one sender also sending 1000 Messages to the client for 60 seconds. The first round the messages were all sent un-encrypted with QoS level 0. In the second round the messages were sent encrypted with an QoS level of 0 again. Similar to Section VII-A, two receivers and one sender is involved in the setup.

VIII. Experimental Results

The following section will include the measured performance for QoS levels with TLS, and for QoS levels without TLS. The graphs have been created using the open source software [19] VMstatly with the performance data of the unix tool [20] vmstat. Table 1 and Table 2 represent the average values of the Idle time, the processes wait, interrupts and context switches of both TLS and non-TLS connection for each of the conducted tests.

A. Recorded Performance QoS TLS

As already discussed in section II the Mosquitto broker is capable of processing around 5000 messages/second without TLS. In this test case we have recorded a slightly higher number (8000) with QoS 0 and no TLS encryption, most likely due to the tweaks performed both on the broker and the publisher. The total amount of sent messages on QoS Level 0 was roughly 116840 confirmed messages with TLS. Due to the nature of QoS0 it was not possible to identify any packet loss during the test. For QoS level 1 and QoS level 2 the received messages were 17111 for QoS 1 and 14272 for QoS 2 with an average rate of ∼237 to ∼290 messages/second, for both levels no publish messages were dropped by the broker. For the combination of all three QoS level the amount of received messages was 23890 with an average rate of ∼400 messages/second, similar to the previous results, no packages were dropped by broker.

B. Processes, CPU and System performance for TLS Connections

The blue line represents the **in** value which shows the number of interrupts (explain what interrupts are) per second, including the clock size (clock rate typically refers to the frequency at which a chip like a central processing unit (CPU), one core of a multi-core processor, is running and is used as an indicator of the processor's speed). As can be seen in Figure 3 the QoS 0 level has an average interrupt of almost 30.000 where QoS2 and QoS1 only required almost 1/6th of the interrupts. The orange line represents the **cs** which shows the number of context switches per second, for QoS 0 the context switches were as high as the interrupt values (with about 25.000) in average.

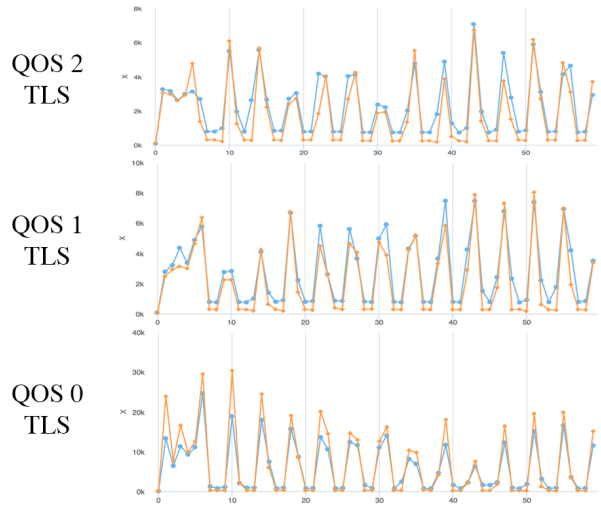

Figure 3. System Statistic

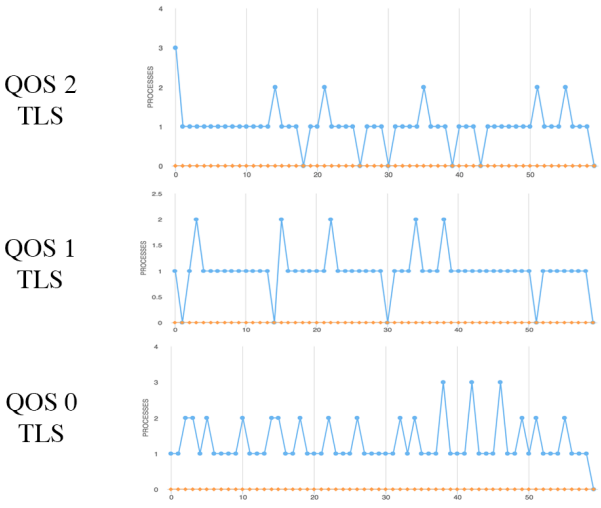

Figure 4. Processes Statistic

The blue **r** value represents the amount of processes waiting for CPU time, so to speak the occupied processes. As can be seen in Figure 4, for QoS 0 almost all of the processes were busy along the recorded time (three out of four). For QoS 1 there were a few peaks were three processes were busy, but on average only only two out of four were working. For QoS 2 the same behaviour could be noted, the average amount based on the figure was roughly two processes. On average Qos2 consumed (see Table 1) 1,01 processes on average.

As can be seen inf Figure 5 the green **id** value represents

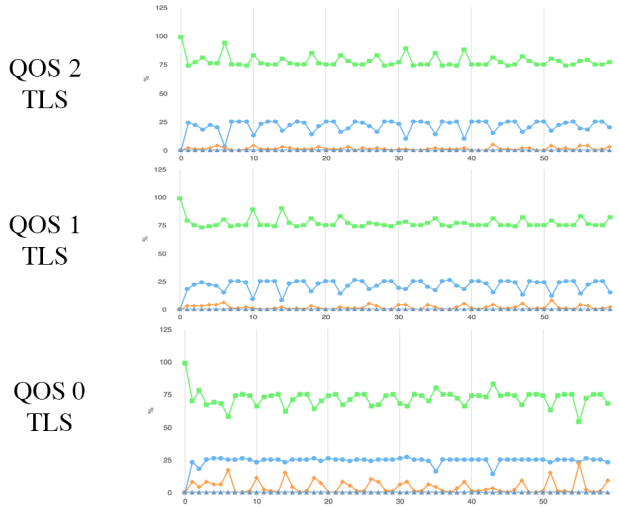

Figure 5. CPU Statistic

the CPU idle time in percentage, and indicates the overall exhaustion of the CPU. For QoS the exhaustion of the Idle time was almost at 1/4th of the available capacity. Compared to QoS1 ad QoS2 this value is slightly higher, QoS1 required almost the same capacity of idle time as QoS2. The blue **us** value represents the percentage of CPU used for running non-kernel code (user time, including nice time), for QoS 0 the values were at at 20% usage constantly. For QoS1 and 2 these values were almost the same, but with additional drops (likely due to the queueing system of the broker). The orange **sy** value represents the percentage of CPU used for running kernel code. (system time - network, IO interrupts, etc), for QoS 0 again, these values were much higher than for QoS1 and 2.

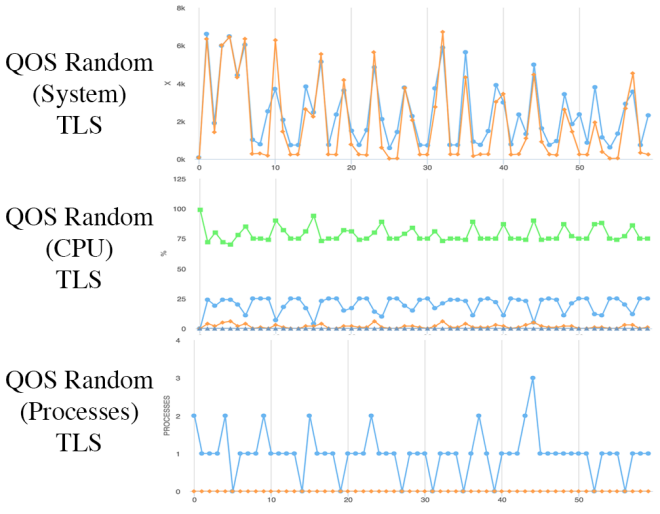

Figure 6. CPU Statistic

For the statistics of the random QoS level, all three measurements have been combined as can be seen in Figure 6. For the used QoS level, there is no technical way to measure the used levels by the library, however statistically every QoS level should have been used equally. Looking at the System values it is notable, that the interrupts of the system were way higher than the for conducted tests with QoS 1 and QoS 2 solely. The same could be observed for the context switches which were also significantly higher. The reason for this may be due to the fast switching of QoS levels and the corresponding operations. The average idle time of the CPU was about 10% lower than for QoS 1 and QoS2, still not as low as for QoS 0 tests. The waiting processes were on average as much as for the QoS 2 level. Looking at the processes exhaustion, on average at least 1 process was exhausted, with a maximum peak of three.

C. Average values for TLS Connections

QoS	idle (id)	Process w. (r)	Interrupts (in)	Context sw. (cs)
0	72,1%	1,31	5900	5900
1	77,05%	1	2748	2197
2	77,78%	1,01	2271	1788
Rand.	78,53%	1	2436	1875

Table I
AVERAGE IDLE, PROCESS WAITING, INTERRUPT AND CONTEXT SWITCHES FOR TLS CONNECTIONS

As can be seen in Table I the avergage idle values for QoS 0 is about 5% lower than for the QoS 1, QoS 2 and QoS random levels. It is suprising to see, that the diffrence of QoS 0 and QoS 1 is only 5% for TLS connections, but almost 13% for non TLS connections (see Table II).

D. Results non TLS Connections

The following subsection will discuss the results of the non-TLS connections, the Figure 5 shows the CPU statistics, the Figure 5 presents the Processes and Figure 6 the system values. For the random usage of QoS levels, Figure 6

E. Recorded Performance non TLS connections

The recorded performance for non TLS connections were slightly higher to the ones obtained in Section VIII-A. For QoS0 a total number of received messages of 529795 could be recorded with an average message speed of ∼9000 messages/second. For QoS level 1 and QoS level 2 the values were about 50% higher than for TLS connections with an average message count for QoS 1 with 112246 Messages and QoS 2 with 112222 messages, combined there were about ∼2000 messages/second set. For each of the non TLS test cases, no packets were dropped by broker for QoS 1 and QoS 2.

F. Processes, CPU and System performance for non TLS Connections

As shown in the Figure 7 the amount of processes was the most claimed by QoS 0, where in comparison QoS 1 and 2 had fewer active processes. In total the average process amount for QoS 0 was at 1.13, and slightly lower for QoS 1 and QoS 2, the random value surprisingly was the lowest value so far. If we compare the values to Table 1, the average process on non TLS connections are 30% less demanded than on TLS connections. The reason for this is more than likely the exchange of the certificates.

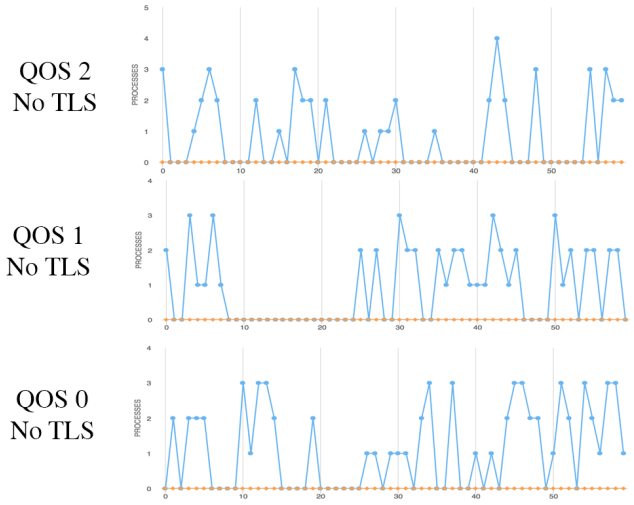

Figure 7. Process Statistic for non TLS connections

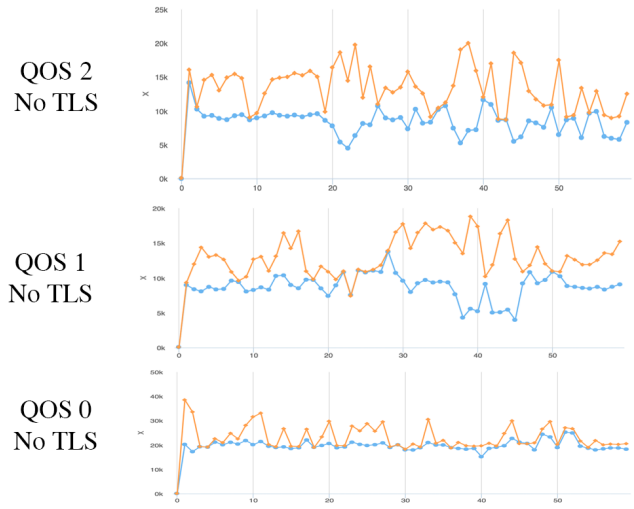

Figure 9. Sytem Statistic for non TLS connections

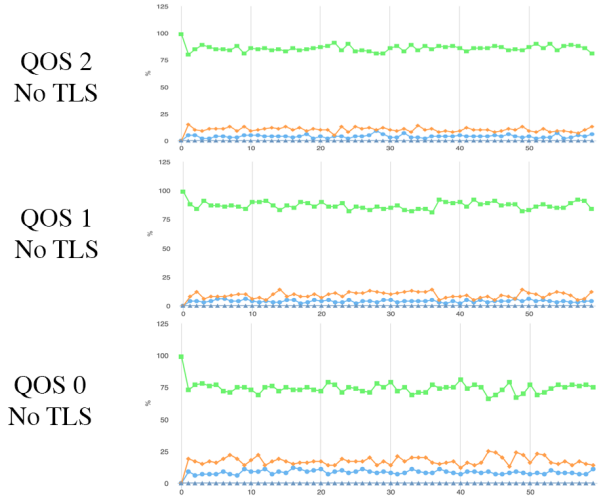

Figure 8. CPU Statistic for non TLS connections

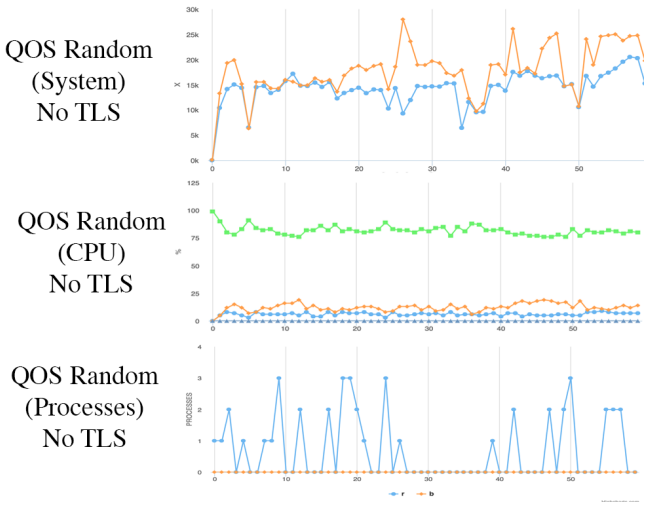

Figure 10. CPU Statistic for non TLS connections

As can be seen in Figure 8 the CPU usage without TLS was again on the highest value on QoS 0, same for the number of interrupts and context switches, if we compare those values to Figure 5, it can be seen that the amount of CPU running non-kernel code is drastically lower (by almost 20%) of all three QoS levels.

Looking on the statistics for System usage, it can be seen, that the context switches for Qos 0 are at almost 40.000 in the first three seconds. The reason for this may be due to the high amount of subscribers in the first couple of seconds. The value however drops slightly trough the runtime of the script, but the average amount of context switches stays with an average amount of 22933 almost twice as high as for QoS 1 (12948) and QoS 2 (13223) and similar to the random values (17823). The same can be observerd for the interrupt cycles, for QoS the average value is about 19500, again the values for QoS 1 and 2 are almost half.

For the random usage of QoS levels, the system values had higher values than for QoS 2 solely, still the most usage of had QoS0 on non TLS and TLS connections. The combined System values were tripled at No TLS connections, which is surprising given the fact that the processing should be fairly easier. One possible explanation could be the increased amount of messages. Looking at the processes, TLS connections required a higher load than non TLS connections, which underlines the results of [9].

G. Average values for non TLS Connections

QoS	idle (id)	Process w. (r)	Interrupts (in)	Context sw. (cs)
0	74,46%	1,13	19528	22933
1	87,01%	0,9	8642	12948
2	85,91%	0,83	8380	13223
Rand.	81,73%	0,71	14246	17823

Table II
AVERAGE IDLE, PROCESS WAITING, INTERRUPT AND CONTEXT SWITCHES FOR NON TLS CONNECTIONS

As can be seen in Table II, the combined average idle time of the CPU was on 74.46% for QoS 0 messages, with an average of 1.13 processes, 19.500 interrupt and 22.000 context switches. In conclusion this means, that it was possible to exhaust the CPU with about 25% using a single sender and a QoS 0 level. The values for QoS 1 and QoS 2 did only vary in roughly 1 % although especially QoS 2 messages should need a advanced CPU usage.

IX. Discussion

As can be seen on Table I on , the CPU idle time of the CPU could be exhausted by 33% and roughly 100.000 messages with QoS0 in one minute with the usage of TLS. For non TLS connections, the sender was capable of sending a increasingly higher amount of messages, (500.000) which may be related to the faster processing of messages due to no TLS usage. If we compare the idle time for non TLS and TLS connections, those only vary with an average of about 6%. Looking at the average waiting processes for non TLS connections (Table 2) the amount of waiting processes is fairly lower than compared to TLS connections (roughly 1 process). The interrupts of the processor didn't vary too much, same goes for the context switches. For QoS1 and QoS2 the performance of the received messages were almost only 1/6 of QoS0. The reasons for this may be a throttling mechanism offered by the broker, which ensures all messages are handled accordingly before a new connection is allowed. For QoS0 however, it seems that there is no limit whatsoever (apart from the operating system/technical limitations described in Section IX-A. This may due to the reason that QoS0 is sent in a "fire and forget" manner [15], if packets are gone, they are gone unnoticed.

A. Technical Limitations

One problem occurring on the Broker system while performing the test cases was *kernel: possible SYN flooding on port X.* Due to the high amount of incoming connections, the Kernel of the Raspberry Pi system supposed that there is a so called SYN flooding attack ongoing and as a result of that dropped the connections. The issue was mitigated by increasing the system backlog as described here [21]. Unix based systems have a so called soft and hard limit in terms of concurrent open files and connections. In order to achieve a higher rate the value can be increased using `/etc/security/limits.conf` and increasing the *hard* and *soft* values.

B. Further Work

The current state of the script leaves room for improvement. So far only the QoS and TLS test cases have been conducted, however it is possible to expand the current script to more complex tasks. Right now the size of the packets is 30 Byte each, however it is possible to send messages up to around 260MB. Additionally the numbers of concurrent subscribers could be increased to 1000/10000/100000 to provoke a higher exhaustion. It would also be possible to send variants of different encoding (SO 8859, UTF-8, UTF-32) etc. Another approach could be to fuzz the available message types of MQTT, a suitable tool has been developed by F-Secure in 2015 [22] which uses the the Radamsa[23] software to generate fuzzing payloads.

X. Conclusion

As shown in Section VIII the Mosquitto Broker was capable of handling all TLS QoS levels without a significant impact on the CPU, Process or System statistic. The same goes for the connections without TLS as seen in Table 2, an additional layer of TLS only impacted the CPU idle time by roughly 2%. Whats remarkable about the conducted tests is, the usage of higher QoS levels does not significantly impact the performance of the broker. As described in chapter III-C, QoS1 and QoS2 levels require a more complex exchange mechanism of messages, although the broker was only capable of handling fewer messages, the overall performance was not affected. On QoS level 0 however, a single sender was capable to exhaust the available CPU by an average of 33%. And about 500.000 messages per minute. The system tests have shown that the processes are way more exhausted (three times) with QoS 0 than the other levels. The random QoS tests did have a larger impact on the overall performance, but were not as significant as the QoS 0 tests. Looking at the least Idle time for TLS connections (Table I) and comparing it to the non-TLS version, the most exhaustion could be achieved using using the QoS level 0 with TLS. In conclusion this means, if a DDoS attack would be operated on a Mosquitto broker, it would be likely successful with a high amount of QoS 0 messages TLS enabled, rather than a higher complexity of the protocol mechanisms.

References

[1] "Mqtt used by facebook messenger — mqtt," http://mqtt.org/2011/08/mqtt-used-by-facebook-messenger, (Accessed on 06/18/2017).

[2] "Hivemq - enterprise mqtt broker," http://www.hivemq.com/, (Accessed on 06/07/2017).

[3] "Rabbitmq - messaging that just works," https://www.rabbitmq.com/, (Accessed on 06/12/2017).

[4] "An open source mqtt v3.1 broker," https://mosquitto.org/, (Accessed on 06/07/2017).

[5] "Introducing the mqtt security fundamentals," http://www.hivemq.com/blog/introducing-the-mqtt-security-fundamentals, (Accessed on 06/05/2017).

[6] "Nist sp 800-132, recommendation for password-based key derivation part 1: Storage applications," http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf, (Accessed on 06/12/2017).

[7] "Ddos attacks in q1 2017 - securelist," https://securelist.com/78285/ddos-attacks-in-q1-2017/, (Accessed on 06/12/2017).

[8] "Benchmark_mqtt_servers-v1-1.pdf," http://www.scalagent.com/IMG/pdf/Benchmark_MQTT_servers-v1-1.pdf, (Accessed on 06/12/2017).

[9] "How does tls affect mqtt performance?" http://www.hivemq.com/blog/how-does-tls-affect-mqtt-performance/, (Accessed on 06/24/2017).

[10] S. Lee, H. Kim, D. k. Hong, and H. Ju, "Correlation analysis of mqtt loss and delay according to qos level," in *The International Conference on Information Networking 2013 (ICOIN)*, Jan 2013, pp. 714–717.

[11] "Github - remakeelectric/mqtt-malaria: Attacking mqtt systems with mosquittos (scalability and load testing utilities for mqtt environments)," https://github.com/remakeelectric/mqtt-malaria, (Accessed on 06/20/2017).

[12] "Github - chirino/mqtt-benchmark: A benchmarking tool for mqtt servers," https://github.com/chirino/mqtt-benchmark, (Accessed on 06/20/2017).

[13] “Small packet overhead - mqtt client user guide - the hcc documentation portal,” https://doc.hcc-embedded.com/display/MQTTCLIENT/Small+Packet+Overhead, (Accessed on 06/25/2017).
[14] S. Behnel, L. Fiege, and G. Muhl, “On quality-of-service and publish-subscribe,” in *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW’06)*, July 2006, pp. 20–20.
[15] “Paho mqtt c client library: Quality of service,” http://www.eclipse.org/paho/files/mqttdoc/MQTTClient/html/qos.html, (Accessed on 06/19/2017).
[16] “mosquitto_passwd,” https://mosquitto.org/man/mosquitto_passwd-1.html, (Accessed on 06/25/2017).
[17] “paho-mqtt 1.2.3 : Python package index,” https://pypi.python.org/pypi/paho-mqtt, (Accessed on 06/19/2017).
[18] “9.6. random — generate pseudo-random numbers — python 2.7.13 documentation,” https://docs.python.org/2/library/random.html, (Accessed on 06/25/2017).
[19] “Vmstatly,” http://jsargiot.github.io/vmstatly/, (Accessed on 06/25/2017).
[20] “Linux performance messungen mit vmstat – thomas-krenn-wiki,” https://www.thomas-krenn.com/de/wiki/Linux_Performance_Messungen_mit_vmstat, (Accessed on 06/25/2017).
[21] “”kernel: Possible syn flooding on port x. sending cookies” is logged - red hat customer portal,” https://access.redhat.com/solutions/30453, (Accessed on 06/22/2017).
[22] “Github - f-secure/mqtt_fuzz: A simple fuzzer for the mqtt protocol,” https://github.com/F-Secure/mqtt_fuzz, (Accessed on 06/25/2017).
[23] “Github - aoh/radamsa: a general-purpose fuzzer,” https://github.com/aoh/radamsa, (Accessed on 06/25/2017).