SAPIENZA
UNIVERSITÀ DI ROMA

Analisi delle problematiche di sicurezza del protocollo MQTT

Facoltà di Ingegneria dell'informazione, informatica e statistica

Classe Informatica L-31

Candidato

Edoardo Di Paolo
Matricola 1728334

Relatore

Prof. Angelo Spognardi

Anno Accademico 2019/2020

Analisi delle problematiche di sicurezza del protocollo MQTT
Relazione di tirocinio. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: dipaolo.1728334@studenti.uniroma1.it

Alla mia famiglia.

Indice

Capitolo 1

Introduzione

Negli ultimi decenni il numero di dispositivi collegati ad Internet è cresciuto esponenzialmente. Ormai, nel 2020, non si hanno più solamente computer o cellulari in rete ma anche elettrodomestici, macchine industriali e strumenti medici, tutti dispositivi che qualche anno fa erano offline. Tutto ciò fa riferimento all'IoT, l'*Internet of Things*.
Parallelamente allo sviluppo di queste nuove tecnologie, sono stati studiati nuovi protocolli affinché i dispositivi potessero essere utilizzati in maniera efficiente. Ad esempio ci sono alcuni sensori i quali permettono di misurare temperatura ed umidità di una stanza e funzionano attraverso l'uso di una semplice pila; dunque è necessario andare a ridurre il costo energetico della connessione così da aumentare la durata di utilizzo del dispositivo.
Alcuni esempi di protocolli possono essere: MQTT, CoAP, AMQP e WebSocket. Ovviamente tutti i protocolli hanno la possibilità di essere integrati con TLS (*Transport Layer Security*) così da poter garantire una maggiore sicurezza nello scambio dei dati; questo, però, potrebbe andare a gravare sui consumi del dispositivo poiché dovrebbero essere effettuati più calcoli affinché avvenga il trasporto dei dati. Inoltre, con l'aumento di questi nuovi dispositivi sono aumentate anche le possibili minacce relative all'IoT. Un esempio è il malware Mirai [11] che, nel 2016, ha infettato milioni di dispositivi rendendoli parte di una botnet la quale, successivamente, ha attaccato attraverso un DDoS il fornitore di servizi DNS Dyn così da rendere inaccessibili milioni di siti web. A causa, anche, di questa tipologia d'attacco, sempre più comune, i protocolli sviluppati devono presentarsi sicuri e robusti.

Negli ultimi anni l'utilizzo del protocollo MQTT è sicuramente cresciuto di gran misura.

Figura 1.1. Numero dispositivi MQTT 2018.

La Figura 1.1 rappresenta il numero di dispositivi che una ricerca della parola "MQTT" produceva nel 2018 sul sito Shodan [3]. Se effettuiamo la stessa ricerca adesso, abbiamo un numero maggiore come si vede dalla figura sottostante.

Figura 1.2. Numero dispositivi MQTT 2020, statistiche di Shodan [8].

La maggior parte di questi dispositivi girano sulla porta standard di MQTT, la 1883; inoltre molti di questi non richiedono alcuna autenticazione per poter connettersi. Effettuando una ricerca ancora più specifica attraverso la stringa "*port:1883*", abbiamo ancora più risultati come si vede dalla Figura 1.3.

Figura 1.3. Numero dispositivi MQTT 2020, port:1883.

In questa relazione viene descritto lo studio effettuato su uno dei maggiori protocolli nell'IoT: MQTT. Nello specifico durante l'attività di tirocinio siamo andati alla ricerca di possibili anomalie e/o violazioni del protocollo da parte dei broker server i quali dovrebbero rispettare ogni standard definito per MQTT.

Nel **Capitolo 2** sarà analizzato il protocollo nel dettaglio assieme alle sue principali caratteristiche e funzionalità.

Nel **Capitolo 3** viene analizzata la libreria che è stata scritta ai fini del tirocinio. In questo capitolo sono anche descritti i differenti broker e il dispositivo fisico su cui sono stati effettuati i test e vengono riportati i risultati di alcuni.

Nel **Capitolo 4** sono riportate le conclusioni del lavoro svolto durante il tirocinio.

Capitolo 2

MQTT

2.1 Descrizione del protocollo

MQTT, acronimo di *Message Queuing Telemetry Transport*, è un protocollo di tipo *publish-subscribe* il quale permette il trasporto di messaggi tra diversi dispositivi tramite TCP/IP. Il protocollo è molto utilizzato in ambito IoT per la sua semplicità e anche per la banda la cui richiesta è davvero bassa.

MQTT presenta un modello di architettura differente, ad esempio, dal tipico client/server del protocollo HTTP; infatti adotta il meccanisco *publish-subscribe* (pub/sub) attraverso l'utilizzo di un *broker*. Il funzionamento del modello pub/sub avviene attraverso la pubblicazione e la sottoscrizione da parte del client a diversi topic, che possono essere paragonati a dei canali di comunicazione. In MQTT il publisher e il subscriber non comunicano mai direttamente e non sono neppure consapevoli della presenza dell'uno e dell'altro: il collegamento è gestito dal broker il cui compito è quello di filtrare i messaggi che riceve e distribuirli ai vari subscribers. In questo capitolo sono analizzate le principali caratteristiche che il protocollo MQTT mette a disposizione.

2.1.1 Topic

Nel protocollo MQTT un topic non è altro che una stringa codificata in UTF-8 che il broker utilizza per filtrare i messaggi da inviare successivamente ad ogni client sottoscritto a quel topic. I topic sono case-sensitive, quindi bisogna prestare attenzione alle lettere maiuscole o minuscole, e la lunghezza minima dev'essere di un carattere. Inoltre, MQTT mette a disposizione del client dei *wildcard*, i quali permettono di connettersi simultaneamente a più topic; uno è rappresentato dal simbolo +, mentre l'altro dal simbolo #. Il primo è chiamato

single-level wildcard, mentre il secondo *multi-level* wildcard. Ci sono dei topic, a cui non si può pubblicare alcun messaggio, riservati allo stato del sistema e sono quei topic che cominciano per $SYS/.
Alcuni esempi di topic validi possono essere i seguenti:

1. *casa/luci/sala* - topic specifico per le luci della sala;
2. *casa/+/sala* - include tutti i topic dei dispositivi che fanno riferimento alla sala (luci comprese);
3. *casa/#* - include tutti i topic che fanno riferimento alla casa.

2.1.2 Connessione al broker

Come scritto nell'introduzione di questo capitolo, la connessione avviene solamente fra client e broker. Per client intendiamo un qualsiasi dispositivo il quale permette di gestire una connessione ad un broker, che si comporta come un server.

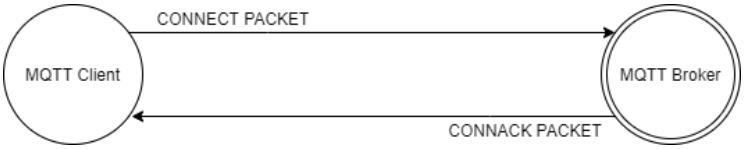

Figura 2.1. Flusso connessione al broker MQTT.

Come si può vedere dalla Figura 2.1, la connessione avviene tramite lo scambio di due pacchetti: *connect*, inviato dal client al broker, e *connack* inviato dal broker al client. Il pacchetto connect è così strutturato:

Tabella 2.1. Struttura pacchetto connect.

Parametro	Descrizione
clientId	rappresenta l'identificativo del client che chiede di connettersi
cleanSession	valore booleano il quale specifica se la connessione è persistente o meno
username	rappresenta l'username necessario affinché avvenga la connessione
password	la password associata all'username
willRetain	viene letto solo se *willFlag* è true
willQos	viene letto solo se *willFlag* è true
willFlag	permette di notificare un messaggio specificato dal client nel payload in caso di disconnessione anomala
keepAlive	rappresenta in secondi l'intervallo massimo in cui broker e client possono non inviarsi messaggi

Il pacchetto connack, invece, è così strutturato:

Tabella 2.2. Struttura pacchetto connack.

Parametro	Descrizione
sessionPresent	flag riferita al cleanSession del pacchetto connect
returnCode	stato della connessione

Per quanto riguarda il *returnCode* descritto nella Tabella 2.2, questo è un valore che va da 0 a 5:

- 0: connessione accettata;
- 1: connessione rifiutata, versione del protocollo non valida;
- 2: connessione rifiutata, clientId non valido;
- 3: connessione rifiutata, server non disponibile;
- 4: connessione rifiutata, username o password errati;
- 5: connessione rifiutata, non autorizzati.

2.1.3 Sottoscrizione e pubblicazione

Come scritto più volte, MQTT è un protocollo che adotta un meccanismo chiamato *publisher/subscribe*; perciò le due azioni fondamentali che un client può compiere sono *publish* e *subscribe*.

Subscribe

L'azione subscribe permette al client di sottoscriversi a un determinato topic passato nel pacchetto.

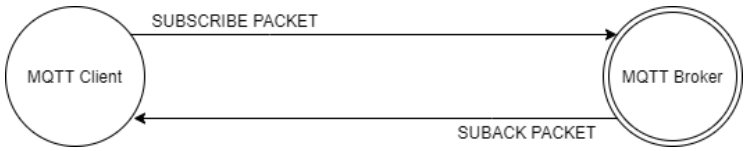

Figura 2.2. Flusso subscription ad un topic in MQTT.

Come si nota dalla Figura 2.2, la sottoscrizione avviene semplicemente con lo scambio di due pacchetti: *subscribe*, dal client al broker, e *suback*, dal broker al client. Il primo pacchetto rappresenta la richiesta di sottoscrizione ad un

determinato topic, il secondo invece conferma la sottoscrizione a quel topic. La struttura del subscribe packet è la seguente:

Tabella 2.3. Struttura pacchetto subscribe.

Parametro	Descrizione
packetId	l'identificatore del pacchetto
payload	una lista di QoS e topic a cui sottoscriversi

Il pacchetto suback, invece, è così strutturato:

Tabella 2.4. Struttura pacchetto suback.

Parametro	Descrizione
packetId	l'identificatore del pacchetto a cui rispondere
returnCode	lista di *returnCode* che rappresentano lo stato della sottoscrizione

Anche in questo caso il *returnCode* può assumere diversi valori, riassunti in questa lista:

1. 0: successo, massimo QoS 0;
2. 1: successo, massimo QoS 1;
3. 2: successo, massimo QoS 2;
4. 128: sottoscrizione fallita.

All'azione di sottoscrizione corrisponde anche un'azione di disiscrizione dal topic che avviene attraverso il pacchetto *unsubscribe*. Quest'ultimo è così strutturato:

Tabella 2.5. Struttura pacchetto unsubscribe.

Parametro	Descrizione
packetId	l'identificatore del pacchetto a cui rispondere
topics	lista di topic da cui disiscriversi

Publish

La pubblicazione di un messaggio in MQTT è più complicata rispetto alla fase di sottoscrizione al topic. Prima di tutto, bisogna introdurre il concetto di QoS, *Quality of Service*. Il QoS è paragonabile ad un contratto che viene stipulato tra client e broker; esistono tre diversi livelli: QoS 0, QoS 1, QoS 2.

Nel Quality of Service di livello 0 il payload del pacchetto viene pubblicato immediatamente. In questo caso non c'è garanzia sull'effettiva pubblicazione del messaggio.

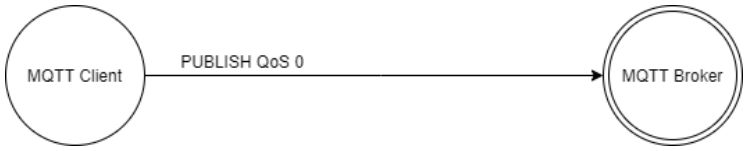

Figura 2.3. Flusso publish con QoS 0.

Nel caso del Quality of Service di livello 1 viene garantita la pubblicazione ad almeno un destinatario.

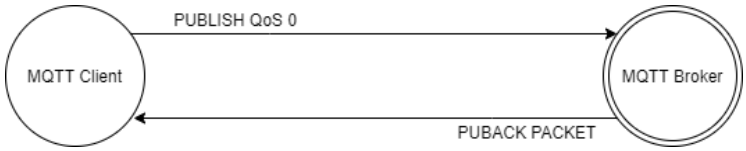

Figura 2.4. Flusso publish con QoS 1.

Come si vede dalla Figura 2.4, il client riceve dal broker il pacchetto *puback* solo dopo aver pubblicato il pacchetto publish. Il pacchetto puback è così strutturato:

Tabella 2.6. Struttura pacchetto puback.

Parametro	**Descrizione**
packetId	l'identificatore del pacchetto publish

Questo meccanismo permette al client di inviare nuovamente il pacchetto publish nel caso in cui, dopo un certo intervallo di tempo, non abbia ancora ricevuto il puback dal broker; in questo caso, nel pacchetto publish, il campo *dup* sarà impostato come true. Inoltre, il client mantiene in memoria il pacchetto publish finché non riceve il puback.

Infine abbiamo il QoS di livello 2 il quale dà maggiore affidabilità ma è più lento rispetto ai precedenti livelli.

Come si vede dalla Figura 2.5 abbiamo più messaggi scambiati fra client e broker. Questo meccanismo assicura che il messaggio sia ricevuto solamente una volta dai destinatari; il livello 2, perciò, dà un maggiore livello di sicurezza al client. Nel caso in cui il pacchetto vada perso, dopo un certo intervallo di tempo il client trasmetterà nuovamente il pacchetto e resterà in attesa della risposta da parte del broker.

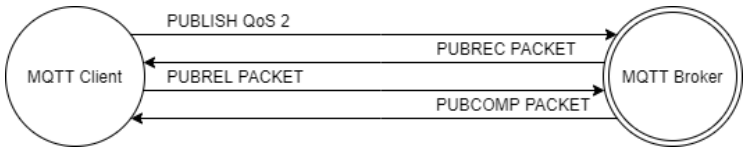

Figura 2.5. Flusso publish con QoS 2.

La struttura dei pacchetti è la stessa che si può vedere nella Tabella 2.6.

Adesso che sono stati analizzati i differenti QoS, si può analizzare nel dettaglio il publish, il pacchetto più complicato del protocollo. Il pacchetto ha i seguenti campi:

Tabella 2.7. Struttura pacchetto publish.

Parametro	**Descrizione**
packetId	l'identificatore del pacchetto
topic	il topic a cui si vuole pubblicare il messaggio
qos	il Quality of Service del pacchetto
retainFlag	booleana per il *retained*
payload	il contenuto del messaggio
dup	booleana che indica se il messaggio è un duplicato

Dopo aver ricevuto il pacchetto il broker processa il messaggio in base al QoS; una volta processato invia il pacchetto a tutti i client sottoscritti al topic.

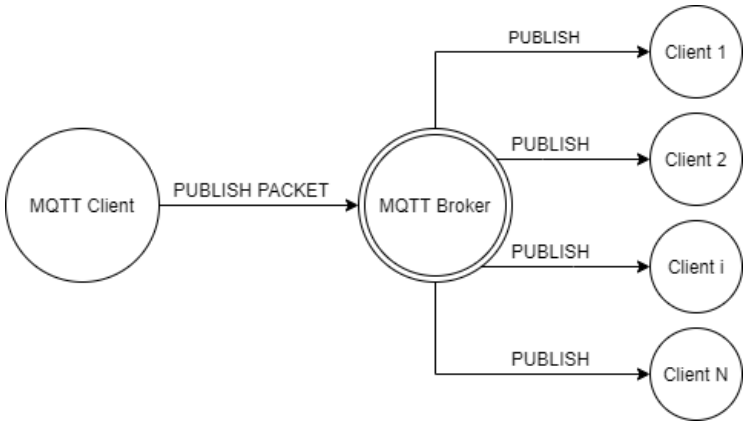

Figura 2.6. Esempio di pubblicazione di un pacchetto.

2.1.4 Altre caratteristiche

Retained message

Un *retained message* è semplicemente un pacchetto che ha impostato il valore true della *retainFlag*. Di solito il broker una volta che riceve un messaggio da pubblicare su un topic, in cui nessun client è iscritto, lo scarta; questo avviene quando il retain è *false*. Nel caso di un *retained message* il broker salva in memoria il pacchetto che ha ricevuto per poi pubblicarlo una volta che un client si sottoscrive al topic.

Last and will testament

Questo meccanismo, indicato anche con *LWT*, ci permette di gestire meglio le disconnessioni improvvise da parte di un client. Quando il dispositivo si connette al broker, nel pacchetto connect (Tabella 2.1), viene specificato il payload nel caso in cui avvenga una disconnessione imprevista; questo payload viene salvato dal broker il quale lo utilizzerà in caso di imprevisto.

Clean session o persisted session

Questa funzionalità viene specificata dal client nel momento della connessione; infatti nel pacchetto *connect* (Tabella 2.1) viene specificata una variabile *cleanSession*. Nel caso di una persisted session il client riceverà dal broker tutti i messaggi pubblicati sui topic a cui si era sottoscritto che sono stati inviati mentre era offline.

MQTT 5

Questa versione del protocollo è stata rilasciata nel 2018 e non è ancora utilizzata da tutti i dispositivi. Rispetto alla precedente versione, sono stati aggiunti i *reason codes* i quali riportano il tipo di errore relativo al protocollo che è avvenuto. La *clean session* non è più presente e al suo posto c'è la flag *clean start*. È stato aggiunto anche un nuovo pacchetto, l'*AUTH* packet, utilizzato ad esempio per le autenticazioni *OAuth*. Il pacchetto *disconnect* in MQTT 5 è bi-direzionale: il broker invia un pacchetto di tipo *disconnect* prima di chiudere la connessione. In questo modo il client può intercettare il motivo della disconnessione ed agire di conseguenza.

Capitolo 3

Implementazione ed esperimenti

Per ricercare possibili anomalie prodotte dalle librerie broker MQTT, c'è stata la necessità di implementare una libreria client la quale potesse permettere di gestire a basso livello i pacchetti e il loro flusso; infatti le librerie come *paho* per python e *mqtt.js* per nodejs non permettono di lavorare a basso livello.

3.1 Implementazione del protocollo

La libreria client è stata scritta in python e utilizza la libreria *twisted* [12]. Sono stati implementati tutti i pacchetti che MQTT mette a disposizione, inclusi tutti i livelli QoS.
Un esempio di pacchetto costruito è quello publish; i vari campi sono descritti nella Tabella 2.7, qui possiamo vedere il *fixed header* del pacchetto.

Tabella 3.1. Fixed header publish packet.

bit	7	6	5	4	3	2	1	0
byte 1	Packet				Dup	QoS level		Retain
	0	0	1	1	x	x		x
byte 2	Remaining Length							

Oltre a questo header, abbiamo il *variable header* che contiene il nome del topic a cui si sta pubblicando e l'id del pacchetto. Infine c'è campo per il *payload*, ovvero il contenuto del messaggio da pubblicare.

Il codice python per la gestione di questo pacchetto è il seguente:

```
def publish(self, topic, message, dup=False, qos=0, retain=False,
                                    messageId=None):
    header = bytearray()
    varHeader = bytearray()
    payload = bytearray()

    header.append(0x03 << 4 | dup << 3 | qos << 1 | retain)
    varHeader.extend(_encodeString(topic.encode("utf-8")))

    if qos > 0:
      if messageId is None:
        varHeader.extend(_encodeValue(random.randint(1, 65535)))
      else:
        varHeader.extend(_encodeValue(messageId))

    payload.extend(_encodeString(message.encode("utf-8")))
    header.extend(_encodeLength(len(varHeader) + len(payload)))

    self.transport.write(header)
    self.transport.write(varHeader)
    self.transport.write(payload)
```

Come si può vedere, il codice rispecchia la costruzione del pacchetto come definito nell'OASIS standard [7]. Innanzitutto vengono inizializzati i tre differenti campi: header (fixed header), varHeader (variable header) e infine payload. Si può notare come il messageId, che è l'identificatore del pacchetto, venga utilizzato solamente nel caso in cui il qos sia diverso da 0; in questo caso, infatti, dopo il publish dovrebbe esserci un ulteriore scambio di pacchetti tra broker e client che coinvolge l'id del pacchetto.
Infine viene aggiunto il payload, codificato in utf-8, e viene esteso il fixed header aggiungendo la lunghezza totale tra varHeader e payload. Il trasporto vero e proprio del pacchetto dal client al broker, invece, è gestito grazie alla libreria twisted attraverso il metodo *write*.

In maniera del tutto analoga sono stati implementati tutti gli altri pacchetti. Attraverso questa libreria client si ha accesso alla costruzione del pacchetto, cosa non possibile nelle librerie come paho; ciò ha permesso di creare dei test particolari che hanno dato risultati differenti in base al broker utilizzato.

Per facilitare l'esecuzione dei test da eseguire, si è creato il programma in modo che potesse prendere dei file *json* i quali contenevano il flusso che la

libreria doveva seguire. Questo ha permesso di poter studiare in maniera più dettagliata i diversi comportamenti da parte dei server.
Un esempio di test è il seguente:

```
[
    {
       "type": "subscribe",
       "params": {
           "topic": "test/topic"
       }
    },

    {
        "type": "publish",
        "params": {
            "topic": "test/topic",
            "message": "pacchetto \#1",
            "qos": 0,
            "dup": false,
            "retain": false,
            "packetId": 1
        }
    }
]
```

In questo caso la libreria client gestiva due pacchetti: il primo di tipo subscribe al topic *test/topic*, il secondo, invece, di tipo publish al topic *test/topic* con QoS 0 e packet id 1. Ovviamente prima veniva inviato il pacchetto di tipo subscribe e solo successivamente quello di tipo publish.

3.2 I broker studiati

Sono stati presi come caso di studio differenti broker, che saranno introdotti molto brevemente in questa sezione. Ricapitolando, il broker ha la funzionalità di filtrare i messaggi che riceve dai differenti client e successivamente inviarli ai destinatari.

3.2.1 Mosquitto

Mosquitto [4] è uno fra i broker più famosi ed utilizzati per quanto riguarda MQTT. È leggero, open source e si adatta molto bene sia con dispositivi di poco consumo che con normali server. Supporta tutte le versioni del protocollo.

3.2.2 EMQ X

Anche EMQ X è un broker molto utilizzato. È scritto in *Erlang* ed è open source; permette di gestire milioni di connessioni simultanee anche con un unico server. Oltre a supportare MQTT, supporta anche CoAP, MQTT-SN (*MQTT for Sensors Networks*) e LwM2M.

3.2.3 HiveMQ Community Edition

HiveMQ Community Edition è un altro broker scritto in Java ed open source. Supporta MQTT a partire dalla versione 3.x fino all'ultima rilasciata che è MQTT 5. È molto utilizzato anche nei sistemi di automazione e in macchinari industriali che necessitano di un sistema real-time.

3.2.4 Moquette

Moquette [5] è un broker meno conosciuto scritto in java e open source. Supporta tutti i livelli di QoS e tutte le versioni del protocollo.

3.2.5 Aedes

Aedes [6] è un broker non troppo conosciuto scritto in NodeJS. Non supporta l'ultima versione di MQTT, ma è pienamente compatibile con la versione 3.1 e la 3.1.1. Ha molte librerie col quale può essere integrato.

3.3 Esperimenti effetuati

La tecnica utilizzata per eseguire i test sulle librerie broker è stata simile al *fuzzing*. Di per sé il fuzzing consiste in un'automazione dei test, i quali forniscono al SUT (*software under test*) dati non validi o input inaspettati. Così facendo si può vedere il comportamento del software e si possono analizzare eventuali crash, buffer overflows e memory leaks.

I test che sono stati effettuati sulle librerie consistevano in file *json* che vengono passati come input alla libreria client. In queta sezione sono riportati i file di alcuni test e i risultati prodotti che possono rappresentare un'anomalia di gestione da parte del broker. Alcuni test effettuati sono stati creati prendendo spunto da questo articolo [9] e i risultati ottenuti sono del tutto simili a quelli descritti nell'articolo.

3.3.1 Publish QoS 2 e 1

```
[
    {
       "type": "subscribe",
       "params": {
           "topic": "test/topic"
       }
    },
    {
        "type": "publish",
        "params": {
            "topic": "test/topic",
            "message": "pacchetto #1",
            "qos": 2,
            "dup": false,
            "retain": false,
            "packetId": 1
        }
    },
    {
        "type": "publish",
        "params": {
            "topic": "test/topic",
            "message": "pacchetto #2",
            "qos": 1,
            "dup": false,
            "retain": false,
            "packetId": 1
        }
    },
    {
        "type": "pubrel",
        "params": {
            "packetId": 1
        }
    }
]
```

In questo test vengono inviati due pacchetti al topic "*test/topic*" con lo stesso id ma differente livello di QoS. Ovviamente, prima di inviarli, la libreria client effettua una sottoscrizione al topic di pubblicazione.

Risultati ottenuti

La tabella sottostante riassume i risultati ottenuti dai diversi broker.

Tabella 3.2. Risultati test pubblicazione QoS 2 e 1.

Broker	Comportamento
Mosquitto	In questo caso il broker pubblicava il primo pacchetto ricevuto, cioè quello di QoS 2. Il secondo pacchetto, invece, viene perso e non viene pubblicato.
EMQ X	Diversamente da Mosquitto, il broker effettuava tutto ciò che era relativo al primo pacchetto publish ricevuto; quindi il client riceveva il pubcomp e successivamente la pubblicazione. Infine veniva pubblicato il secondo pacchetto, che quindi non viene perso.
HiveMQ	Il comportamento di questo broker è simile a quello di EMQ X; entrambi i pacchetti vengono pubblicati nell'ordine prestabilito, tuttavia, per quanto riguarda il primo, il client riceve la pubblicazione e solo successivamentene il pubcomp.
Moquette	Anche in questo caso il broker pubblica il primo pacchetto, pur ricevendo dopo il pubcomp, e successivamente il secondo.
Aedes	In questo caso il broker si comporta in maniera leggermente differente dai precedenti. Pur pubblicando entrambi i pacchetti nell'ordine prestabilito, riceve il pubcomp come ultimo pacchetto anche dopo la pubblicazione del secondo.

3.3.2 15000 subscription

In questo test sono stati inviati 15000 pacchetti di tipo subscribe, in un intervallo di tempo molto piccolo, al broker per vedere come si comportasse. La struttura del file *json* è la seguente:

```
[
  {
    "type": "subscribe",
    "params": {
      "topic": "random_topic_subscription_5858",
      "packetId": 5314
    }
  }
]
```

Il topic e il packet id sono stati generati randomicamente; nel file ci sono, in realtà, 15000 pacchetti di questo tipo.

Questo genere di test è volto più a verificare il comportamento del broker nel caso in cui riceva un flood di pacchetti.

Risultati ottenuti

Anche in questo caso i broker si sono comportati in maniera differente e i risultati sono riassunti nella tabella sottostante.

Tabella 3.3. Risultati test sottoscrizione di 15000 pacchetti.

Broker	Comportamento
Mosquitto	Il client perdeva la connessione dopo aver inviato circa 3800-4000 pacchetti al broker.
EMQ X	In questo caso l'invio dei pacchetti non comporta alcuna problematica; infatti tutti i pacchetti sono gestiti correttamente dal broker e il client non perde la connessione.
HiveMQ	Come in Mosquitto, il client perdeva la connessione dopo più o meno 4000 pacchetti inviati.
Moquette	Anche in questo caso il client perdeva la connessione dopo più o meno 4000 pacchetti inviati.
Aedes	Questo broker si è comportato come EMQ X. È riuscito a gestire correttamente tutti i subscribe che sono stati inviati dal client non producendo alcuna disconnessione.

3.3.3 Topic lungo

Nello standard di MQTT la lunghezza massima della stringa che rappresenta il topic è di 65536 caratteri. Tuttavia nelle source del broker EMQ X [2] si può vedere che la lunghezza massima è di soli 4096 caratteri, così è stato testato il subscribe ad un topic lungo 4097 caratteri.

Risultati ottenuti

I risultati sono riassunti nella tabella che segue.

Tabella 3.4. Risultati test topic lungo 4097 caratteri.

Broker	Comportamento
Mosquitto	La sottoscrizione avviene con successo.
EMQ X	Disconnessione del client.
HiveMQ	Viene tagliato il nome del topic.
Moquette	Sottoscrizione avvenuta con successo.
Aedes	Disconnessione del client, crash del server.

Come si può vedere dalla Tabella 3.4, i broker si comportano differentemente. Si può notare che *Mosquitto* e *Moquette* sembrano accettare senza problemi il topic; *HiveMQ*, invece, effettua un controllo sulla lunghezza del topic e nel caso sia troppo lungo taglia dei caratteri affinché il client non venga disconnesso. *Aedes* sembra addirittura crashare scrivendo nella console *"Too many words"*.

3.3.4 Publish QoS 2 e 0

```
[
    {
       "type": "subscribe",
       "params": {
           "topic": "test/topic"
       }
    },
    {
        "type": "publish",
        "params": {
            "topic": "test/topic",
            "message": "pacchetto #1",
            "qos": 2,
            "dup": false,
            "retain": false,
            "packetId": 1
        }
    },
    {
        "type": "publish",
        "params": {
            "topic": "test/topic",
            "message": "pacchetto #2",
            "qos": 0,
            "dup": false,
            "retain": false,
            "packetId": 1
        }
    },
    {
        "type": "pubrel",
        "params": {
            "packetId": 1
        }
    }
]
```

Questo test è identico a quello con QoS 2 e 1, però qui abbiamo il secondo pacchetto inviato con QoS 0. Il test è interessante poiché ha prodotto risultati inaspettati in alcuni casi.

Risultati ottenuti

I risultati ottenuti sono riassunti nella tabella che segue.

Tabella 3.5. Risultati Publish test QoS 2 e 0.

Broker	Comportamento
Mosquitto	Il broker pubblica prima il pacchetto con QoS 0 e successivamente gestisce, correttamente, tutto il flusso per quanto riguarda il primo pacchetto, che ha QoS 2.
EMQ X	Diversamente da Mosquitto, EMQ X pubblica gestisce il primo pacchetto inviato e lo pubblica. Successivamente avviene la pubblicazione del pacchetto con QoS 0.
HiveMQ	Il broker si comporta esattamente come EMQ X, quindi avviene prima la pubblicazione del pacchetto con QoS 2 e solo successivamente quella con QoS 0.
Moquette	Anche in questo caso viene pubblicato prima il pacchetto con QoS 2 e dopo quello con QoS 0. Piccola anomalia: il pubcomp riferito al primo pacchetto arriva al client solo dopo la pubblicazione del secondo pacchetto.
Aedes	Il broker si comporta similmente a Mosquitto; infatti viene pubblicato prima il secondo pacchetto, quello con QoS 0, e successivamente quello con QoS 2. Il client, anche in questo caso, sembra ricevere il pubcomp riferito al primo pacchetto dopo la sua pubblicazione.

3.3.5 Doppio publish QoS 2

In questo test vengono inviati due pacchetti di tipo publish con lo stesso QoS e lo stesso id.

```
[
    {
       "type": "subscribe",
       "params": {
          "topic": "test/topic"
       }
    },
    {
        "type": "publish",
        "params": {
            "topic": "test/topic",
            "message": "pacchetto #1",
            "qos": 2,
            "dup": false,
            "retain": false,
            "packetId": 1
        }
    },
    {
        "type": "publish",
        "params": {
            "topic": "test/topic",
            "message": "pacchetto #2",
            "qos": 2,
            "dup": false,
            "retain": false,
            "packetId": 1
        }
    },
    {
        "type": "pubrel",
        "params": {
            "packetId": 1
        }
    },
    {
        "type": "pubrel",
        "params": {
            "packetId": 1
        }
    }
]
```

Risultati ottenuti

Anche in questo caso i broker hanno prodotto risultati differenti e in alcuni casi anche particolari.

Tabella 3.6. Risultati Publish test doppio QoS 2.

Broker	Comportamento
Mosquitto	Pur inviando, correttamente, il doppio pubrel, il broker in questo caso pubblica solamente uno dei due pacchetti. Infatti il client sembra ricevere indietro, nell'ordine, il pubcomp relativo al primo pacchetto, successivamente la pubblicazione di quest'ultimo e infine il pubcomp relativo al secondo pacchetto che, però, non viene pubblicato; perciò il broker perde un pacchetto.
EMQ X	In questo caso il broker si comporta esattamente come Mosquitto; viene perso il secondo pacchetto.
HiveMQ	A differenza di EMQ X e Mosquitto, questo broker pubblica entrambi i pacchetti nell'ordine prestabilito, ricevendo indietro prima i relativi pubcomp; perciò in questo caso non c'è alcuna perdita di pacchetti.
Moquette	Il broker si comporta in maniera del tutto simile ad HiveMQ; infatti pubblica entrambi i pacchetti, tuttavia il client riceve indietro i pubcomp solo dopo la pubblicazione dei pacchetti.
Aedes	Il broker ha un comportamento molto strano. Vengono effettuati due publish, ma entrambi riguardano il primo pacchetto; quindi il secondo, in realtà, viene perso.

3.3.6 Altri esperimenti

Oltre ai test riportati in questa relazione, ne sono stati effettuati anche altri. Ad esempio, sono stati eseguiti dei test riguardanti la codifica del topic; tutti i broker studiati in questo caso si son comportati bene senza far notare alcuna anomalia. Anche per quanto riguarda il *client id* contenente caratteri non codificati in utf-8 non si son verificati problemi.

Ulteriori esperimenti sono riassunti nella lista sottostante assieme ad una breve descrizione del risultato ottenuto:

- intervallo del *keepAlive* stringa: in tutti i broker è stata registrata la disconnessione del client a causa del pacchetto di connessione malformato;

- sottoscrizione (o pubblicazione) ad un *wildcard* non valido: in tutti i broker è stata registrata la disconnessione del client a causa del topic non valido;
- *wildcard* codificato in: *utf-16*, *zzlib*, *bz2* e *base64*. Negli ultimi tre casi, il pacchetto subscribe è stato inviato correttamente e gestito altrettanto dal broker; per quanto riguarda la prima codifica utilizzata, l'utf-16, è sempre avvenuta la disconnessione del client;
- flood pacchetti publish QoS 0: tutti i broker hanno gestito correttamente il flood dei pacchetti senza andare in crash;
- versione (o nome) del protocollo non validi nel pacchetto di connessione: tutti i broker hanno disconnesso il client;
- invio di un pacchetto pubrel riferito ad un pacchetto publish mai inviato: i broker, tranne *Aedes*, hanno risposto mandando indietro al client il pubcomp relativo; in *Aedes*, invece, avviene la disconnessione del client.
- payload pacchetto di pubblicazione di 100MB: solamente *Mosquitto* ha gestito il test correttamente, mentre in tutti gli altri casi è avvenuta la disconnessione del client.

3.4 Librerie client

Parallelamente ai test eseguiti sulle librerie broker, ne sono stati effettuati alcuni anche su quelle client disponibili in rete. Sono state prese in esame tre librerie: *paho*, *mqttools* e *mqtt.js*; le prime due sono entrambe scritte in *python*, mentre l'ultima è in *javascript*.

I test eseguiti non hanno rivelato particolari anomalie, anche perché sono librerie open source mantenute da community attive. Alcuni degli esperimenti effettuati sono riassunti nella lista sottostante:

- livello QoS non valido: tutte le librerie riportano l'errore che il QoS specificato per la pubblicazione del messaggio non è valido, bloccandone così l'invio. Da notare che in *mqtt.js* avviene il crash del client a causa di un errore riferito agli indici di un *array*;
- sottoscrizione ad un wildcard non valido: *mqtt.js* fa notare nella console l'errore, generando il log *"Invalid topic"*. Le altre librerie, invece, sembrano andare in *timeout*;

- *client id* non codificato in utf8: per quanto riguarda *mqttools* il client non riesce a connettersi al broker; in *paho* il client si connette con successo al broker, mentre in *mqtt.js* viene generato un errore e, quindi, non viene stabilita la connessione;
- topic con più di 65536 caratteri: in tutt'e tre le librerie avviene la disconnessione del client.

3.5 Dispositivo fisico

Nella domotica, il protocollo MQTT è molto utilizzato dato che la maggior parte dei dispositivi "intelligenti" lo supporta.

Figura 3.1. Dispositivo Shelly.

Nella Figura 3.1 si può vedere il dispositivo fisico sul quale sono stati eseguiti alcuni test che hanno confermato i risultati precedentemente ottenuti. Questa lampadina supporta la comunicazione attraverso il protocollo MQTT; infatti è possibile accenderla, spegnerla e cambiarle la luminosità e il tipo di luce da remoto.

La configurazione del protocollo, in questo dispositivo, avviene in una semplice interfaccia web, disponibile sulla rete al quale è connesso, in cui bisogna specificare l'indirizzo e la porta del server su cui gira il broker MQTT. È possibile anche scegliere il livello di QoS sul quale far viaggiare i pacchetti e alcuni valori come *min reconnect timeout*, *max reconnnect timeout* e *keepalive*. Inoltre è possibile specificare username e password nel caso in cui la connessione richieda l'autenticazione dell'utente. Infine permette di specificare la flag della

cleanSession e del *retain*.

Questo dispositivo mette a disposizione dell'utente diversi topic con cui lavorare; ecco quelli più importanti:

- *shellies/announce*: in questo topic viene pubblicato un messaggio dal dispositivo che contiene informazioni utili come: *id*, *model*, indirizzo mac, indirizzo ip, *new_fw* e *fw_ver*. Gli ultimi due valori si riferiscono al firmware del dispositivo, il primo è una variabile booleana la quale indica se c'è un aggiornamento da eseguire, il secondo indica la versione del *firmware*;
- *shellies/ShellyBulbDuo-D0CC77/info*: fornisce alcune informazioni del dispositivo come ad esempio la rete al quale è connesso, lo stato della memoria e aggiornamenti firmware;
- *shellies/ShellyBulbDuo-D0CC77/light/0/command*: permette di accendere o spegnere il dispositivo passando come payload "on" o "off";
- *shellies/ShellyBulbDuo-D0CC77/light/0/set*: permette di modificare lo stato del dispositivo passando semplicemente un payload *json*;
- *shellies/ShellyBulbDuo-D0CC77/light/0/status*: riporta le informazioni dello stato attuale del dispositivo.

Il server broker scelto per far collegare il dispositivo è *Mosquitto*. Dai test effettuati, si è visto che il dispositivo, di per sé, non ha un "anti-flood" per quanto riguarda i pacchetti ricevuti; infatti, teoricamente, è possibile, ad esempio, spegnere ed accendere la lampadina in maniera ripetuta e veloce attraverso l'invio di un publish sul topic specifico.

Interessanti sono stati i risultati dei test precedentemente effettuati sul broker. Ad esempio, il test nella sezione 3.3.1 è stato eseguito di nuovo, ma questa volta sul dispositivo stesso.
Il contenuto del file di test, ovviamente, è leggermente diverso da quello presente in 3.3.1; in questo caso pubblichiamo due pacchetti publish dove l'unica differenza è il parametro *turn*, che sul primo è "on" mentre sul secondo "off".

```
[
    {
        "type": "publish",
        "params": {
            "topic": "shellies/ShellyBulbDuo-D0CC77/light/0/set",
            "message": "{\"brightness\":50,\"white\":50,\"temp\":2700
                                                ,\"turn\":\"on\"}",
            "qos": 1,
            "dup": false,
            "retain": false,
            "packetId": 1
        }
    },
    {
        "type": "publish",
        "params": {
            "topic": "shellies/ShellyBulbDuo-D0CC77/light/0/set",
            "message": "{\"brightness\":50,\"white\":50,\"temp\":2700
                                                ,\"turn\":\"off\"}",
            "qos": 1,
            "dup": false,
            "retain": false,
            "packetId": 1
        }
    },
    {
        "type": "pubrel",
        "params": {
            "packetId": 1
        }
    }
]
```


Come ci si poteva aspettare, viene pubblicato solamente il primo aggiornamento e quindi la lampadina non esegue il secondo comando, che equivale a quello di spegnimento.

Sono stati effettuati anche altre tipologie di test, come un payload codificato stranamente o payload pesanti; in tutti i casi il dispositivo ignora il messaggio malformato non producendo alcun aggiornamento.

Una delicata parte di questi dispositivi è sicuramente il *firmware*. Quest'ultimo è un programma implementato direttamente nel componente elettronico; nel caso di *Shelly*, il firmware non è *open source*, per cui non si hanno le

sorgenti. I dispositivi IoT, per natura, non hanno grandi risorse: poca memoria e processori poco performanti; di conseguenza anche i software devono essere piuttosto leggeri e quindi potrebbero esserci falle di sicurezza. Gli attacchi basati su *firmware* in IoT sono abbastanza comuni e rappresentano un bel problema che non è facile da arginare. In [1] sono descritte 5 categorie di attacchi basati su firmware:

- *reverse engineering*: consiste nell'estrare il firmware ed analizzarlo, andando alla ricerca di possibili vulnerabilità;
- *firmware modification*: consiste nell'inserire codice malevolo all'interno del firmware;
- *obtaining access authorization*: se il dispositivo comunica con altri dispositivi, previa autenticazione, l'attaccante può attaccare anche quest'ultimi;
- *installing unauthorized firmware*: l'attaccante può installare un firmware malevolo per prendere il controllo del dispositivo e, ad esempio, farlo così partecipare ad una botnet;
- *unauthorized device*: consiste nell'utilizzare il firmware del dispositivo valido su uno non valido; possibili problemi di privacy.

Attraverso queste possibili vulnerabilità, l'attaccante può prendere il controllo del dispositivo e farlo partecipare a possibili attacchi. Ad esempio, una tipologia nuova di attacco, basato su MQTT, è descritto in [10]. L'idea alla base è quella di inizializzare un alto numero di client col server fino ad arrivare alla saturazione del pool di connessioni del broker.

Oltre agli attacchi, ovviamente, un dispositivo vulnerabile può causare una violazione della privacy non di poco conto. Ad esempio, una volta che l'attaccante ha preso il controllo remoto di un dispositivo nella rete potrebbe effettuare un attacco di tipo *man in the middle* per visualizzare il flusso dei messaggi dagli altri dispositivi collegati. In aggiunta a questo, l'attaccante potrebbe anche avere accesso a tutti i dispositivi domotici collegati in casa e, paradossalmente, potrebbe prendere il controllo di quest'ultima.

Iniziano ad essere diffusi anche i malware di tipo *ransomware* nei confronti dei dispositivi IoT. Un *ransomware* è un malware che limita l'accesso al dispositivo e, affinché si riprenda il controllo, viene richiesto un riscatto da

pagare, di solito in bitcoin (inoltre non si è sicuri di poter ri-ottenere il controllo).

Capitolo 4

Conclusioni

In questa relazione è stato descritto tutto il lavoro di studio durante l'attività di tirocinio. Inizialmente si sono cercate possibili vulnerabilità in MQTT stesso ma essendo un protocollo piuttosto semplice non è stato trovato alcunché. Invece, dai test effettuati sui differenti broker presi in esame, si son notate alcune anomalie e differenze nella gestione di alcuni test effettuati. In effetti non tutti i broker si son comportati secondo gli standard definiti per MQTT e questi esempi potrebbero essere un punto d'inizio per iniziare a rendere i comportamenti dei server uguali.
Per quanto riguarda il dispositivo fisico provato, si è visto che gestisce bene alcuni test strani, tuttavia manca, a mio parere, un controllo sul numero di pacchetti ricevuti in un intervallo di tempo. Il vero problema di questi dispositivi, probabilmente, è il *firmware*; quest'ultimo è possibile aggiornarlo anche tramite MQTT, pubblicando un messaggio particolare (nel caso dello shelly in *announce/command*).
Come visto nell'introduzione, il protocollo sta prendendo sempre più piede nelle piccole realtà; questo può preoccupare. Molti broker presenti in rete, infatti, non hanno alcuna protezione per l'accesso e, perciò, ci si può connettere senza problemi e rimanere in ascolto su quei canali.

Ringraziamenti

Bibliografia

[1] Bettayeb, M., Nasir, Q., and Talib, M. A. Firmware update attacks and security for iot devices: Survey. In *Proceedings of the ArabWIC 6th Annual International Conference Research Track*, ArabWIC 2019. Association for Computing Machinery, New York, NY, USA (2019). ISBN 9781450360890. Available from: https://doi.org/10.1145/3333165.3333169, doi:10.1145/3333165.3333169.

[2] EMQX. Source length topic. https://github.com/emqx/emqx/blob/8e658edb767040f699a42d751b1be1098c31329a/src/emqx_topic.erl (2020).

[3] Hron, M. Are smart homes vulnerable to hacking? https://blog.avast.com/mqtt-vulnerabilities-hacking-smart-homes (2018).

[4] Light, R. A. Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, **2** (2017), 265. Available from: https://doi.org/10.21105/joss.00265, doi:10.21105/joss.00265.

[5] Moquette. Moquette. https://github.com/moquette-io/moquette (2020).

[6] MoscaJS. Aedes. https://github.com/moscajs/aedes (2020).

[7] Open, O. OASIS Open. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Table_3.3_- (2020).

[8] Shodan. Shodan website. https://shodan.io (2020).

[9] Sochor, H., Ferrarotti, F., and Ramler, R. 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). ISBN 9781450388337. Available from: https://doi.org/10.1145/3407023.3407078, doi:10.1145/3407023.3407078.

[10] Vaccari, I., Aiello, M., and Cambiaso, E. Slowite, a novel denial of service attack affecting mqtt. *Sensors*, **20** (2020), 2932. doi:10.3390/s20102932.

[11] Wikipedia. Mirai (malware) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Mirai%20(malware) (2020).

[12] Wikipedia. Twisted (software) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Twisted%20(software) (2020).