

Security analysis of protocols used in IoT solutions

Andrea Olivari

July 5, 2018

Contents

1	Introduction to IoT	3
1.1	Structure of an IoT application	4
1.2	Structure of the thesis	6
I	MQTT	7
2	Protocol Overview	8
2.1	Architecture and basic concepts	9
2.1.1	Topics and subscription	10
2.1.2	Quality of Service Levels (QoS)	16
2.1.3	Retained messages	17
2.1.4	Persistence	18
2.1.5	Last Will and Testament (LWT)	18
2.1.6	Wildcards	19
3	Security Overview	20
3.1	MQTT combined with TLS	23
3.1.1	Session resumption	23
3.2	Client Authentication	25
3.2.1	Client Id	25
3.2.2	Username and Password	26
3.2.3	X.509 Client certificates	27
3.3	Authorization	29
3.4	Simple Data Encryption	30
3.5	Data Integrity	33
3.6	MQTT combined with an SDP	34
3.6.1	What is an SDP?	34

3.6.2	How does an SDP work?	35
3.6.3	How can this architecture be combined with MQTT?	37
3.7	Best practices	40
3.8	Homemade secure MQTT system	42
3.8.1	Hiding resources	43
3.8.2	SPA mechanism implementation	44
4	Real-world examples	49
4.1	Case 1 - Real-world MQTT-based industrial system	50
4.1.1	Factory	52
4.1.2	Connected Machines	54
4.2	Case 2 - Every aspect is important	56
II	ZigBee	58
5	Protocol Overview	59
5.1	Introduction	59
5.2	Architecture and basic concepts	60
5.2.1	ZigBee network topologies	61
5.2.2	Protocol Stack	64
5.2.3	ZigBee communication modes	65
6	Security Overview	67
6.1	Introduction	67
6.2	Security assumptions	68
6.3	Security models	69
6.4	Protocol Stack Security	74
6.4.1	802.15.4 Security	74
6.4.2	ZigBee Security features	76
6.5	ZigBee vulnerabilities	81
6.6	Considerations and best practices	84
7	Conclusion	86

Chapter 1

Introduction to IoT

Internet of Things (IoT) is a neologism, introduced by Kevin Ashton [1] in 1999, which has become popular in recent years due to the growing interest from companies and because nowadays it is possible to produce small devices with a reasonable computational power. Often, these small devices are embedded into bigger objects in order to improve them by making them, in some way, *smart*.

IoT has a variety of different applications ranging from everyday things, like watches that measure how many kilometers we have walked during the day, to smart homes that control the air conditioning system on the basis of temperature and pollution in the air, or even smart cities where semaphores adjust their times on traffic conditions.

But these are only three examples, IoT extends also to other fields like: health care, industrial monitoring, self driving cars and agriculture.

IoT is also changing the way software is developed: in order to produce good IoT systems, companies are brought to cooperate with each other, moving from a one-company-does-it-all perspective to a lets-work-together approach [2]. This means that proprietary systems are no more a good choice and companies should embrace open systems in order to cooperate better.

From the financial point of view, IoT seems like a very profitable market: it is esteemed that the global IoT market will grow to \$457B by 2020 [3]. Assuming this forecast is on target, it means that lots of devices will be de-

ployed in the network, exposing a device on the Internet also means exposing it to threats.

An interesting case of insecure IoT system is given by a luxury hotel in Austria [4]: the electronic key system was attacked by hackers that were able to lock out (or in, it depends) guests their rooms, until they paid a ransom; the system lacked a fundamental security component necessary to avoid intrusion.

Long story short: after the attack, the hotel reverted to physical keys.

We can imagine other problematic situations that do not imply an attack; for instance: if a blackout occurs, is a guest still able to access his well paid room?

What if the content of the electronic key is not encrypted and it is possible to change its content in order to access other guests' room?

This event should also make us think if we really need an IoT system for every aspect of our life. An electronic key system is convenient, since you can configure the electronic key as you want in few seconds, but on the other hand it is also easy to corrupt its content, while it is not so simple to steal and duplicate a physical key.

The security aspect of an IoT application is not related only to possible external attacks by criminals but it also involves reliability; just think about a power plant monitored by an IoT system, if it is not well designed and for some reason it does not take the proper counter measures to an overload, a blackout can occur and the consequences are bad, mostly on the financial point of view.

These are the reasons why I am interested in analyzing the security aspect of IoT applications, considering that in near future IoT could be integral part of our life I think it is crucial to develop secure system in order to avoid problems.

1.1 Structure of an IoT application

Most of times an IoT system is a large scale application that put in place different protocols, hardware and software.

From the point of view of the protocols, there are lots of choices, such as: *HTTP*, *MQTT*, *WebSocket*, *ZigBee*, *CoAP*, etc; each of them has its own

advantages and disadvantages, so it is crucial to choose the one that fits our business logic; also considering that some protocols like MQTT or CoAP are used in sensors-server communication, while HTTP and WebSocket are used for client-server communications.

While dealing with IoT we can bump into different types of hardware: from constrained devices with low computational power to high performance cluster machines. Basically the “dirty work” is taken from low power devices to the server, which elaborate received data or forward them to the cloud. Talking about software, instead, we can work with a variety of programming languages: from a low level *Assembly* to high level languages like *C#* or *JavaScript*.

The development of an IoT system firstly requires a deep knowledge of the different parts of an application, hence the development team should consist of experts with different skills and specializations. This little overview should help us to understand how difficult could be to design and develop an IoT application; Figure 1.1 illustrates the schema of a possible architecture of an IoT application.

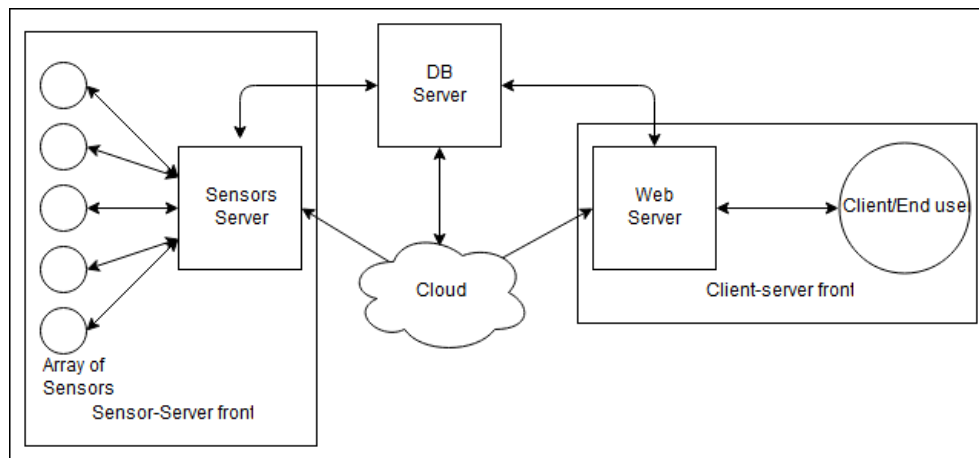


Figure 1.1: Possible structure of an IoT application

1.2 Structure of the thesis

In this thesis I am going to analyze and discuss two of the most important and widespread IoT protocols: *MQTT*, in part I, and *ZigBee*, in part II. More in details, we will start looking at their architecture and basic concepts, then we will discuss their security features, weaknesses and possible fixes. Finally we list the best practices to follow in order to build a a secure communication environment.

With regard to the MQTT protocol, we discuss two interesting real use cases on which I had the opportunity to work during my internship, in order to see how much theory is respected in practice and how a real-world production system should be designed.

Unfortunately, I had not have the opportunity to work on industrial ZigBee projects, during my internship, since, as we will see, it is a protocol mainly aimed to home applications rather than industrial ones.

Part I

MQTT

Chapter 2

Protocol Overview

MQTT (*Message Queue Telemetry Transport*) [5] is a simple, open and light-weight standardized messaging transport protocol, built on top of TCP/IP, invented by *Andy Stanford-Clark* of *IBM* and *Arlen Nipper* of *Cirrus Link Solutions*[6] in 1999, with the following goals:

- minimal battery loss
- minimal bandwidth usage
- simple to implement
- good quality delivery service

All these aspects make this protocol particularly suitable for very constrained devices, which are often used in IoT services.

Nowadays MQTT is used in many different fields of applications:

- Sensors communications (*light, temperature, humidity, magnetic fields, pressure, intrusion detectors, etc*)
- Health monitoring devices (*e.g. blood pressure, insulin, heartbeat*)
- Fitness devices (*e.g. fitbit*)
- Location services
- Home automation kits used in *smart homes*

- Inventory tracking (*systems designed to track movements of inventory*)
- Automotive telematics
- Instant messaging applications (*for instance, Facebook Messenger*)

Internet of Things requires a *real-time event-driven communication model*, and it must be possible to send information in *one-to-many* modality efficiently.

Since we mainly deal with sensors sending lots of data continuously, we need to reduce as much as possible the size of data packets to be sent, reliably.

HTTP is not suitable in this case because it is based on the request/reply paradigm and has a big network overhead. We must always remember that IoT deals with low-power devices whose connectivity can be really poor and extremely expensive.

MQTT is simple and light-weight, and can manage a huge number of communications as well as granting the delivery and reception of messages, without having to poll sensors.

2.1 Architecture and basic concepts

MQTT protocol exploits the modern publish/subscribe pattern, sometimes called pub-sub, which is a valid alternative to the traditional client-server model, in which a client communicates directly with an endpoint.

This design pattern provides asynchronous communication between clients. Nodes are arranged around a component, usually called *broker* or *dispatcher*, in a star topology and, as we can see in Figure 2.1, they can talk to each other by publishing and subscribing to topics; we discuss them in the section 2.1.1.

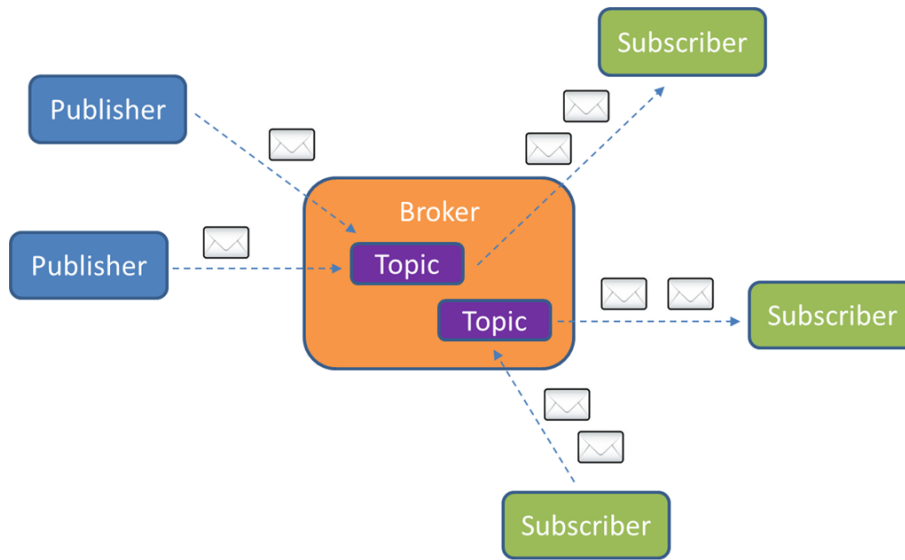


Figure 2.1: MQTT Architecture

The sender of a message is called *publisher*, while who is interested in receiving it is called *subscriber*.

So, a broker is nothing but a kind of message container, known by both the publisher and subscriber, able to filter all incoming messages and distribute them accordingly.

2.1.1 Topics and subscription

Clients publish or subscribe to particular topics, which are simply message subjects having a syntax similar to file pathnames. For instance, assuming we want to name two topics related to sensors placed in our kitchen and bedroom, we may opt for the following names:

'sensors/home/lights/kitchen'

'sensors/home/lights/bedroom'

MQTT brokers use these topics to decide to who forward the messages to.

It is important to notice that publishers and subscribers do not know about the existence of one another, but they can talk anyway. This behaviour is part of a process called *decoupling of publisher and subscriber*, which can be split in three dimensions:

- *space decoupling*: publisher and subscriber do not know each other; to be more clear, they do not know the IP address and port used by the other one
- *time decoupling*: publisher and subscriber do not have to be synchronized and may run at different times
- *synchronization decoupling*: operations on both components are not suspended while publishing or receiving messages; they are independent

Figure 2.3 shows a schematic example of the publish/subscribe pattern combined with topics subscription, but, in order to better understand it, let's keep a reference list of the existent MQTT messages we are going to deal with. Figure 2.2 lists all of them.

Name	Value	Direction of flow	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Client request to connect to Server
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Client to Server or Server to Client	Publish message
PUBACK	4	Client to Server or Server to Client	Publish acknowledgment
PUBREC	5	Client to Server or Server to Client	Publish received (assured delivery part 1)
PUBREL	6	Client to Server or Server to Client	Publish release (assured delivery part 2)
PUBCOMP	7	Client to Server or Server to Client	Publish complete (assured delivery part 3)
SUBSCRIBE	8	Client to Server	Client subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Unsubscribe request
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response
DISCONNECT	14	Client to Server	Client is disconnecting
Reserved	15	Forbidden	Reserved

Figure 2.2: MQTT messages

Message types' names are self-explanatory, anyway we will describe some of them more in details along the way.

Now, let's analyze the example we were talking about.

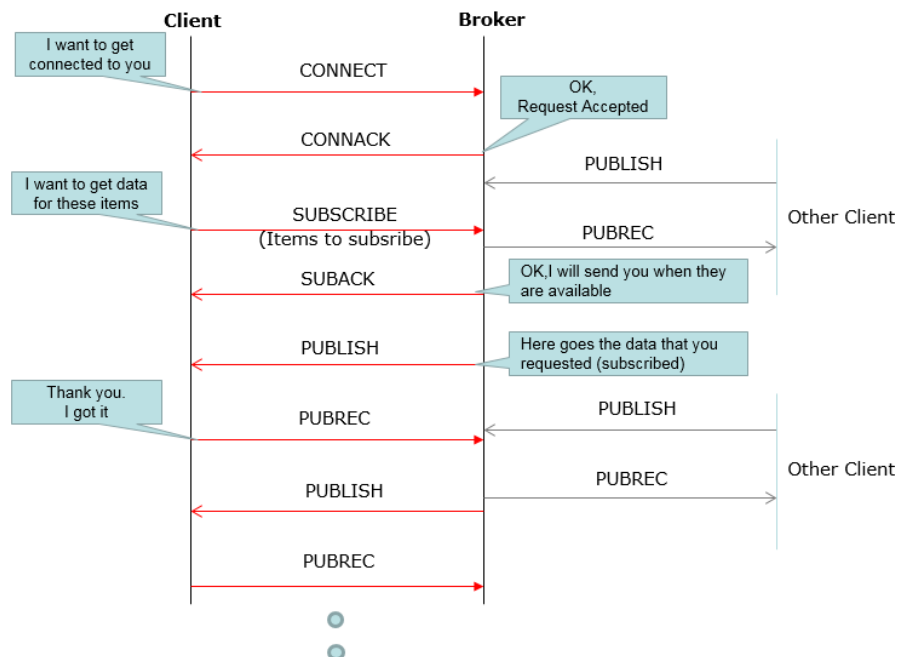


Figure 2.3: Publish/Subscribe combined with Topics

1. A client wants to connect to the broker, and sends the proper **CONNECT** packet, whose structure is visible in Figure 2.4.

MQTT-Packet:	
CONNECT	
contains:	Example
clientId	"client-1"
cleanSession	true
username (optional)	"hans"
password (optional)	"letmein"
lastWillTopic (optional)	"/hans/will"
lastWillQos (optional)	2
lastWillMessage (optional)	"unexpected exit"
lastWillRetain (optional)	false
keepAlive	60

Figure 2.4: CONNECT packet

where:

- *clientId* is a identifier, unique per broker, each client must have. *If you do not need a state to be held by the broker, in MQTT 3.1.1 it is possible to send an empty value.*
- *clean session* is used to establish persistent connections
- *username* and *password* are used to authenticate the user in a password-protected broker. These credentials are sent in plain-text, we will see later how to face this issue.
- *keep alive* is nothing but a time interval used by the client to commit regular **PING Request** messages to the broker (that will reply with a **PING Response** message)

2. Once the broker has received a connection request, it replies with a **CONNACK** packet, containing a field called *return code*. This field can assume six different values, where the first one (Connection accepted) indicates the connection has been accepted, while the others indicate that connection has been refused for some reason (Unacceptable protocol version, Invalid identifier, Server unavailable, Not Authorized, Bad username and password).

3. Assuming that the connection has been accepted by the broker, then the client can choose to publish a message or subscribe to a topic.

In the first case it will send a **PUBLISH** packet, whose structure is visible in figure 2.5.



Figure 2.5: PUBLISH packet

where:

- *packetId* is a reference identifier to the current packet (it is unique per client)
- *topicName* is the topic we want to publish on
- *qos* and *retainFlag* will be explained in details later
- *payload* is the actual content of the message
- *dupFlag* is a flag used to warn receivers that this message might have been already received

The broker will reply sending a PUBREC packet, used to let the client know that its message has been correctly received.

The broker will send the message to subscribers as soon as possible, asynchronously.

In the second case it will send a SUBSCRIBE packet, which contains multiple pairs (*topic,qos*), since a single packet can be used to subscribe to multiple topics.

Clearly, the broker will reply with a SUBACK packet, containing the return code for each topic the client is interested to.

4. If a client wants to unsubscribe a topic it simply sends an UNSUBSCRIBE packet to the broker (as in the SUBSCRIBE case, the client can specify multiple topics).

5. Finally, the user should disconnect gracefully from the broker sending a `DISCONNECT` message.

Now that we have an idea of how MQTT communications work, in the section 2.1.2 we discuss aspects regarding the quality of the service. The content of the sections 2.1.2 to 2.1.6 is mainly referred to the *MQTT Essentials* series published by *HiveMQ* [7].

2.1.2 Quality of Service Levels (QoS)

We have briefly introduced the structure of the PUBLISH and SUBSCRIBE packets in section 2.1; both of them contain a field called *QoS*, acronym of *Quality of Service*.

This field allows clients to set the desired QoS level, according to how much reliability they expect from message delivering. Clearly, being QoS level an agreement between the client and the broker, this last one has to honor the contract. There are three different QoS levels:

- *QoS0 (at most once delivery)*: messages are sent at most once, so their delivery is not guaranteed
- *QoS1 (at least once delivery)* messages are certainly delivered to the client at least once, but they might be also delivered more times.
- *QoS2 (exactly once delivery)* messages are sent exactly once by using the following 4-way handshake mechanism. Of course, the exchange of 4 packets increases the overhead and afflicts performances, therefore this level should be used only in critical scenarios that cannot afford to lose messages or receive duplicated messages.

Figure 2.6 wraps up the three levels we have just discussed.

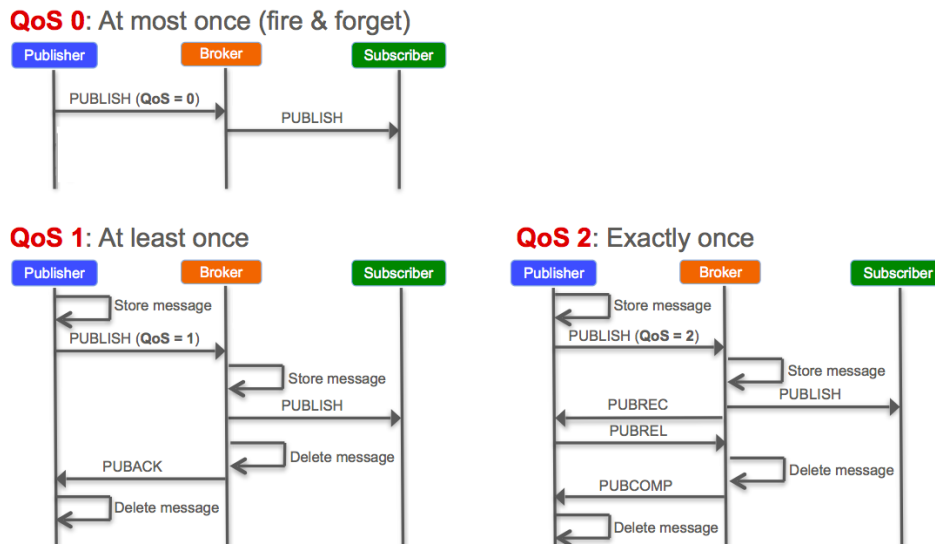


Figure 2.6: MQTT QoS levels

The following real-world examples help to better understand when we should choose one level rather than the others.

- if we have a series of solar panels that publish every second some measures on a specific topic, we can accept with no worries that some messages are not delivered correctly, so we may use QoS0.
- if we have an alarm sensor which publishes an alert message on a topic only if something bad happened in our house, then we want to be sure that such message will be correctly delivered, even more than once, so we may opt for QoS1.
- if a diabetic person has an auto-injecting insuline device applied to his body, which expects to receive an MQTT message when an extra injection is needed. It is necessary to receive one and only one message, so we must choose QoS2.

2.1.3 Retained messages

In MQTT, a retained message is a normal message having the retained flag set to true. MQTT brokers always store the last retained message received, in order to send it to new subscribers, immediately after subscribing to the corresponding topic.

What are retained messages useful for?

We must remember that MQTT clients are autonomous, therefore they do not know the presence of each other; this means that when a subscriber subscribes to a topic, it has no way of discovering how much time it will take before receiving a message.

This problem is solved using retained messages.

Example scenario:

if we have a certain device which updates his online status (true/false) on the topic '/devices/devicename/status' only when it changes, and a new subscriber arrives, then he would probably want to be notified immediately about the status of the device.

2.1.4 Persistence

MQTT also supports persistence of messages.

This feature is very useful because each time a client connects to the broker, a new session for subscribing or publishing to topics is started; if, for any reason, the connection is lost, the process starts all over again with the client establishing a new session.

Clearly, this behaviour afflicts the performances of the system, especially when clients have low power and poor connectivity (for instance, intermittent connectivity).

A client can request a persistent session setting the *cleanSession* flag within the **CONNECT** packet to false.

Let us notice an important aspect of this feature that we are going to discuss later: to resume a session, the broker has to recognize whether the user is the same as before, and it can do that by using their unique identifier; once the broker finds the match between the *clientId* and an available persistent session, that session is immediately reassigned to that client.

2.1.5 Last Will and Testament (LWT)

It is not uncommon for clients to get disconnected ungracefully from the broker.

Therefore, there exist scenarios in which it may be helpful to notify other clients that a specific client has been suddenly disconnected.

Thanks to the LWT feature provided by MQTT brokers, clients can choose a message that will be published to previously chosen topics when they get disconnected.

As already seen, a client can specify the LWT message as part of the **CONNECT** packet, then the broker will store the message till it detects that the client has been abruptly disconnected, finally topic's subscribed clients will be notified.

Let's clarify that “ungracefully disconnected” means that a network failure happens, or the client fails to communicate within the keep-alive time, or the client closes the connection without sending a `DISCONNECT` message or the server faced a protocol error.

In the real world LWT is often used in combination with retained messages to keep the state of a client updated on a specific topic.

For instance, when a client has connected to a broker, it will send a retained message to a topic, named something similar to “*clientname/status*”, with the payload “*online*”. After that, the client sets the LWT message on the same topic to “*offline*” and, clearly, mark it as retained.

2.1.6 Wildcards

A wildcard is simply a shortcut to allow a user to subscribe to multiple topics without having to specify each of them individually.

Just to make an example: if the user wants to check the humidity of all his house's rooms, he could simply subscribe to:

'sensors/home/ + /temperature'

to automatically subscribe to topics like the following ones:

'sensors/home/livingroom/temperature'

'sensors/home/kitchen/temperature'

'sensors/home/bathroom/temperature'

This first kind of wildcard is called “*single-level*” since it allows the user to avoid to specify one single path level.

There is another type of wildcard, called “*multi-level*”, which allows the user to avoid to specify multiple path levels.

For instance, both the topics *'sensors/home/livingroom/temperature'* and *'sensors/garage/temperature'* can be subscribed by simply subscribing to *'sensors/#/temperature'*.

Chapter 3

Security Overview

MQTT is probably the most used IoT protocol, and experts think it will play an important role in the coming years, therefore it is necessary to talk about its security aspects.

Just to begin, we can take a look at its real-world usage in order to understand how much this protocol is widespread, and how many times its security is completely ignored.

To get this data we use a search engine, called *Shodan* [8].

Shodan, developed in 2009, lets the user find specific IoT “things” (webcams, routers, servers, sensors, printers, traffic lights, etc) connected to the Internet, providing a huge variety of filters.

Thanks to its dangerousness and power, many people call it “*the hackers’ search engine*”.

Its searching mechanism is based on the so-called *service banners*, which are nothing but metadata sent back to clients by servers: service banners contain a lot of information describing the kind of server/device you are talking to.

By default, MQTT brokers listen on port 1883; we can simply find the current visible servers listening on that port by using Shodan and searching for “*port:1883*”.

On 3rd June 2018 this search produced 160393 results, as shown in figure 3.1.

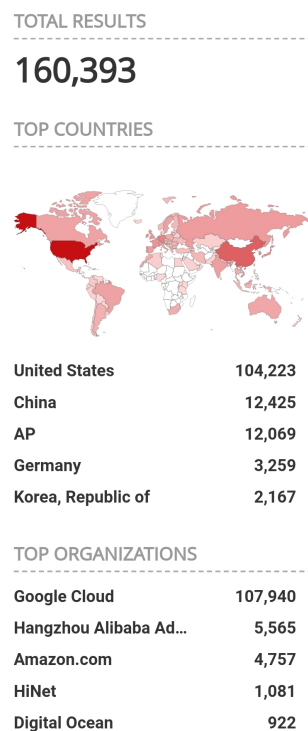


Figure 3.1: MQTT brokers found by using Shodan

We found, all over the world, more than 160 thousand devices listening on port 1883, many of them owned by important organizations like Google, Amazon, Microsoft and Alibaba. While this is already a big number, a more interesting thing is that such a number is growing up very quickly: just two weeks before the same research returned almost 146 thousand devices.

The port 8883 is standardized for a secured MQTT connection, whose name is “secure-mqtt”, and this port is exclusively reserved for MQTT over TLS. The first clue that leads us to think that security is neglected is that, if we search again on Shodan, this time filtering on port 8883, we get only 56 servers, so statistically only one broker in a thousand opt for the secure version of the protocol.

Besides, I found a very interesting article on a reliable blog I have been vis-

iting for several years, whose content is strongly related to cyber security¹. An easy-to-implement Python scanner script, exploiting a paid Shodan API key, has been implemented to search for servers listening on port 1883 and then try to connect to them by sending a **CONNECT** message with no authentication. The strategy of this attack is pretty simple: the broker we are trying to connect to will reply, as we have already seen, with a **CONNACK** packet containing the return code; if the return code is equal to 0 it means that the connection has been accepted, therefore the broker leaves the door open to anyone.

An analysis conducted on 800 found servers, has led to the following results.

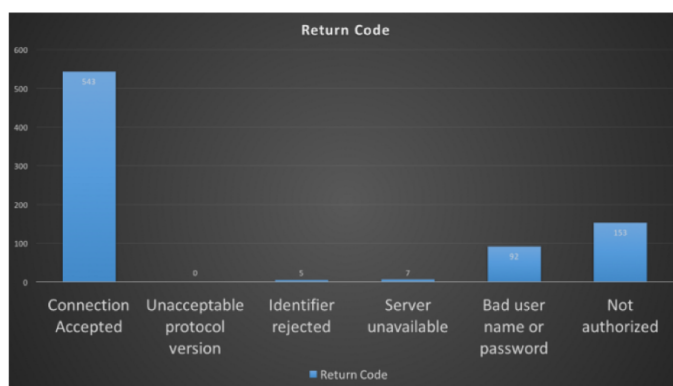


Figure 3.2: MQTT brokers return code statistics

543 brokers out of 800, which is almost 67%, do not use authentication at all!

¹<https://morphuslabs.com/>

3.1 MQTT combined with TLS

Before starting to talk about the authentication methods used by MQTT, we must always remember that MQTT relies on TCP as transport protocol, which means that by default the protocol itself has no built-in encryption. This choice was made purposely by developers to keep it simple, fast and lightweight; therefore, everything is sent in plain text.

We know that TLS uses an handshake mechanism to negotiate various parameters necessary to establish an encrypted and secure communication which cannot be read or altered by third parties.

Usually servers provide their *X.509 certificates*, signed by a trusted authority, to clients to prove their identity.

The negotiation of the parameters brings to a communication overhead afflicting performances, requiring more work for the CPUs. This may turn out to be a problem for very constrained devices.

Let's suppose to have a constrained device which establishes a new connection every time it has to communicate with the broker; this will bring a huge overhead because the handshaking messages will be the most significant portion of content to send in each communication (considering that usually MQTT devices send simple text messages of few bytes, and the handshake ones are a few KB).

Therefore, it is recommended for constrained devices to use long-living MQTT connections rather than short-living ones in order to drastically improve the TLS performances. This idea is implemented in a technique, called *Session resumption*.

3.1.1 Session resumption

TLS session resumption techniques allow to reuse an already negotiated TLS session after reconnecting to the server; doing this the client does not have to perform the entire TLS handshake again.

Unfortunately, some TLS libraries do not implement session resumption, although most of them do.

There are two session resumption techniques:

- **Session Ids:** the server simply stores the secret Session Id associated to a client, so that when this last one reconnects, it provides the Session Id and the session is resumed.
- **Session tickets:** the servers send to clients a secret ticket encrypted with a secret key known only by the server.
When a client reconnects, it sends its state to the server and if this last one is able to decrypt it, the session is resumed with the state contained in the secret ticket.

Despite everything, very constrained devices do not have enough computational power to support TLS anyway. In this case, clients might think to use TLS-PSK cipher suites (not supported by all TLS libraries), since they avoid CPU intensive operations, or totally avoid TLS and use other strategies discussed later.

To conclude this TLS section, we list some best practices:

- Clients should always use TLS or, more generically, encrypted communication channels, if they can afford it
- Use the newest version of TLS, since older ones might have known, and now fixed, bugs [9]
- Always use certificates from trusted Certification Authorities. Using self-signed certificates or allow-all approach is not recommended

In section 3.2 we will talk about client authentication techniques supported by MQTT.

3.2 Client Authentication

There are three ways used by MQTT brokers to verify the identity of a client:

- Client Id
- Username and Password
- X.509 Client certificates

Let's analyze each of them in details.

3.2.1 Client Id

We have already seen that clients must include within the **CONNECT** message a field called *clientId* to identify themselves.

This procedure can be, in some way, already considered as a form of authentication since each client must have a unique identifier.

It is common practice to use 36-character long identifiers, or some unique information available to the client like his MAC address, or the serial number of the device. Clearly, this is not secure because someone can easily pretend to be you, just by connecting to the broker specifying your clientId.

Depending on the broker's configuration, if the real client has already connected to the broker, the attacker's connection request would be discarded or the real client would be disconnected, but if not, the connection would be certainly accepted.

How can the attacker obtain this information? He can easily sniff that, since communication is not encrypted.

We have discussed in section 2.1.4 MQTT persistent sessions, saying that the broker restores the persistent session of a client after he proves its identity specifying its Client Id during connection.

This mechanism is not secure by default, in fact an attacker, who have sniffed the unique identifier of the client, can use it to pretend to be him, reads the

new messages from his subscribed topics and those sent while it was offline, publishes messages, or simply disable persistence making the real client losing possible important information.

Another vulnerability is that the broker stores as much information as possible for clients who have requested persistence, so it stores all the messages from their subscribed topics until it finish its available memory.

This behaviour gives an attacker the possibility to connect the broker, requiring a persistent connection, subscribe to as many topics as possible (for instance, subscribing to '#'), then disconnect; doing this, the broker will store a huge number of messages to be sent to the client when it reconnects.

Moral of the story: *never rely on clientId to secure your system.*

3.2.2 Username and Password

There are two main problems related to this authentication method:

1. Since, by default, no encryption system is used, sending credentials is totally pointless, in fact even a trivial *Man In The Middle* attack [10] would be enough to sniff them without effort, as we can see in figure 3.3.

	17	3.227036	192.168.1.6	192.168.1.7	MQTT	141	Connect Command
	19	3.227328	192.168.1.6	192.168.1.7	MQTT	90	Publish Message

▼ Connect Flags: 0xc2

- 1... = User Name Flag: Set
- .1. = Password Flag: Set
- ..0. = Will Retain: Not set
- ...0 0... = QoS Level: At most once delivery (Fire and Forget) (0)
-0.. = Will Flag: Not set
-1. = Clean Session Flag: Set
-0 = (Reserved): Not set

Keep Alive: 60

Client ID Length: 23

Client ID: paho/1ACECFDE977A1F352D

User Name Length: 15

User Name: statusTopicUser

Password Length: 19

Password: statusTopicPassword

0000	08	00	27	c8	a4	ea	7c	04	d0	c0	9c	4e	08	00	45	00	..'...N..E.
0010	00	7f	00	00	40	00	40	06	b7	1b	c0	a8	01	06	c0	a8	...@.e.
0020	01	07	cf	6c	07	5b	04	16	2a	90	c2	22	5a	07	80	18	...l.[.. *. "Z...
0030	10	15	25	36	00	00	01	01	08	0a	45	07	78	ac	a1	45	..%6.... ..E.x..E
0040	19	06	10	49	00	04	4d	51	54	54	04	c2	00	3c	00	17	...I..MQ TT...<..
0050	70	61	68	6f	2f	31	41	43	45	43	46	44	45	39	37	37	paho/1AC ECFDE977
0060	41	31	46	33	35	32	44	00	0f	73	74	61	74	75	73	54	A1F352D..statusT
0070	6f	70	69	63	55	73	65	72	00	13	73	74	61	74	75	73	opicUser ..status
0080	54	6f	70	69	63	50	61	73	73	77	6f	72	64				TopicPas sword

Figure 3.3: MQTT credentials sent in plain text

2. Authentication is optional, so a huge number of topics are easily accessible from the whole world.

The only way to use credentials safely is to combine them with an encrypted communication tunnel (better if TLS).

3.2.3 X.509 Client certificates

Another possible way to identify clients is using their X.509 certificates, which will be sent during the TLS handshake.

We are used to servers providing their X.509 certificates to clients to prove their identity, and clients authenticating in the application layer.

In this case, however, we perform *mutual authentication*, since client and server authenticate each other at the same time using X.509 certificates.

The broker will use the information in the certificate for application layer authentication after the TLS handshake has completed with success.

This authentication method is very secure but rarely used because it requires the provisioning and managing of the certificates to the clients, besides this double negotiation afflicts even more the performances.

Now, let's take a look at pros and cons of using client certificates.

- Pros:
 - only valid clients are allowed to establish a secure connection
 - authentication of clients at transport level, so invalid clients are discarded before **CONNECT** messages are sent.
This is very useful to save resources on broker's side, because authentication mechanism can be expensive (a common case is that credentials are stored in a database, which must be interrogated every time someone wants to connect) and using client certificates makes them useless.
- Cons:
 - Managing a huge number of client certificates may be a serious burden, therefore it is important to have a solid certificate provisioning process. In other words, clients' certificates must be kept

updated, and owners should have a way to manage their lifecycle. It is suggested to use certificates, only when clients are under their strict control.

- It could happen that a client’s certificate get leaked and, consequently, cannot be trusted anymore since some attacker may use it, so it is necessary to have a way to revoke invalid certificates.

There are two ways to obtain that:

1. *Certificate Revocation Lists (CRLs)*: they are simple lists of certificates that have been revoked before their expiration. These lists are created and continuously updated by the related Certification Authority. A server hosting a broker can have its own local CRL to consult, but clearly it has to get updates from the CA, from time to time.

This is not a very convenient procedure.

2. *Online Certificate Status Protocol (OCSP)*: it is a protocol used for asking information about specific client certificates, more specifically to know if they are still trustable.

In order to use OCSP there must be a so-called *OCSP Responder*, which is an HTTP server waiting for revocation-check requests, provided by the certification authority. This is a better and faster solution since you do not have to download the entire, and possibly huge, list every time before checking a certificate.

Let’s see a simple example [11] of how it works:

- * Alice and Bob want to communicate and have their certificates issued by a certification authority, called Carol
- * Alice starts the transaction by sending her certificate to Bob
- * Bob, who suspects that Alice’s private key might have been compromised, sends an *OCSP request*, containing Alice’s certificate serial number, to Carol
- * Carol’s OCSP responder search in its database the certificate’s serial number received from Bob
- * At this point, Carol’s OCSP responder returns a signed successful *OCSP response* to Bob if the certificate is still valid, otherwise will reply telling Bob that the certificate

is not valid anymore

- * Depending on the response received by Carol, Bob will decide whether carry on, or not, the communication with Alice.

To sum up, this kind of authentication is secure but expensive, therefore only suited to a small number of clients which need high security.

In section 3.3 we will discuss authorization, another important concept related to security.

3.3 Authorization

Authorization and authentication go hand in hand, but they are very different security concepts.

While authentication is used to recognize the identity of a client, authorization allow us to create policies in order to restrict access to certain resources (*in this scenario resource means topic*).

In MQTT, without authorization, every authenticated client could do whatever he wants, so publish or subscribe to any existent topic.

More in details, the resources to protect are the common actions: publishing, subscribing, setting a LWT or retained message, requiring a persistent session.

How can the broker restrict permissions on topics?

Brokers can limit clients' permissions implementing some authorization policies called *topic permissions*, having, at least, the following information:

- Allowed topic
- Allowed operations (publish, subscribe, LWT, retained messages)
- Allowed quality of service levels (0, 1, 2, all)

What does it happens when a client tries to access a restricted resource without having enough permissions?

If the client tries to publish, the broker can disconnect it or simply reply to it normally, but without relaying the message to subscribers; instead, if the client tries to subscribe, the broker will reply with a standard **SUBACK** message containing a return code that tells the client that its subscription was denied, nothing more.

In section ?? we discuss another form of authentication/authorization mechanism usually implemented in other protocols, like HTTP, and we will see whether and how it is possible to combine it with MQTT.

3.4 Simple Data Encryption

In this section we discuss a workaround to grant secrecy of messages without involving TLS: encrypting only sensitive data, such as the payload in **PUBLISH** messages.

Using this strategy sensitive data, let us assume the payload in our examples, will be encrypted, but other metadata remains unencrypted.

Notice that the broker has no decryption key, so it will relay the messages to subscribers as is, without knowing their real content.

Clearly, this mechanism expects subscribers to have the proper key to decrypt the received payload.

Figure 3.4 explains this scenario, called *End-to-End*, perfectly.

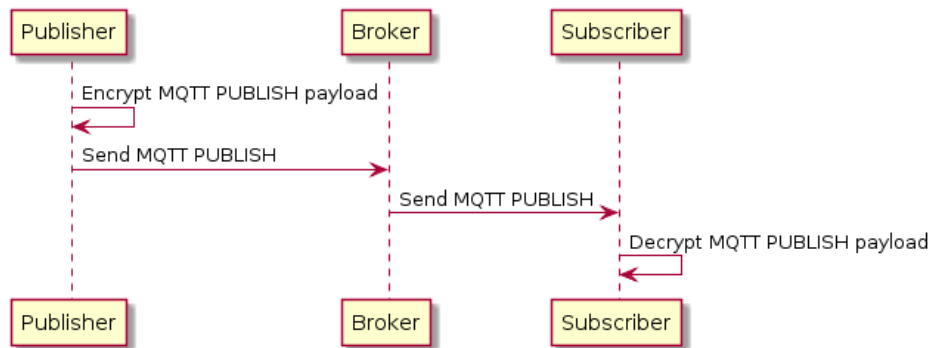


Figure 3.4: MQTT communication using End-to-End encryption

Now, a question arises: if we encrypted meta data, like client's credentials in the `CONNECT` packet, how could the broker decrypt and validate them? This is a very uncommon practice, because it requires to write an additional plugin for the broker in order to make it able to decrypt our data. This second scenario is called *Client-to-Broker*.

In this case, the broker will send the decrypted message to subscribers, which will not have to know any key, as we can see in the following figure 3.5.

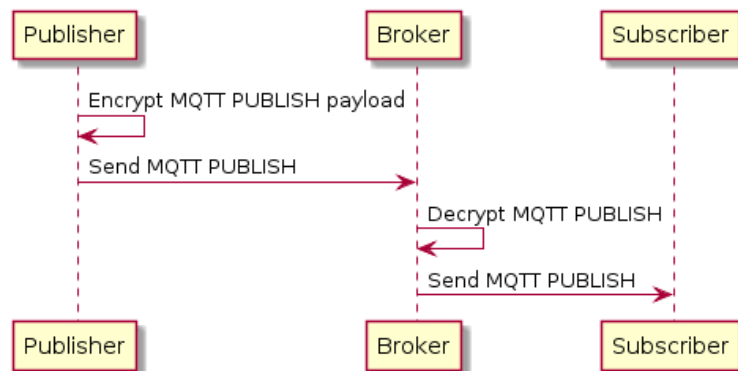


Figure 3.5: MQTT communication using Client-to-Broker encryption

Should we use an asymmetric or symmetric encryption scheme to encrypt the payload in a `PUBLISH` message? We should

- prefer asymmetric cryptography when we have many, possibly untrusted, publishers and few trusted subscribers, because these last ones

know the private key which is used to read the messages in the subscribed topics. In this way, only trusted clients can decrypt the topic's messages.

- use symmetric cryptography when we have a trusted environment, in fact we must remember that in this case there is only one key to encrypt and decrypt, so we can share it with everyone.

Certainly, using a single symmetric key is simpler to implement, but less flexible.

We conclude this section wrapping up the pros and cons of data encryption:

- encrypting only specific data is a method to add a security layer to communication, and can be an option for constrained devices which cannot rely on TLS.
- encrypting and decrypting every message can be intensive for constrained devices, in fact it is fundamental to choose a fast encryption algorithm
- we must remember that encrypting only specific data prevents attackers to sniff our sensitive information, but man in the middle attacks, as well as replay attacks, are still possible.
Just to make an example: an attacker could intercept a valid message and modify some parts of it, like the topic
- keys management adds a layer of complexity, for instance it is important to establish a secure provision of them.

3.5 Data Integrity

We previously described tokens and how their signatures are used to make sure they have been generated by a trusted source, like the Authorization server; now, we see, in general, how brokers can validate received messages. Data integrity is fundamental because it is the only way to assert that the content of a message is authentic and has not been altered by third parties. There are three different approaches for message integrity checks:

- *Checksum*: a value calculated, with a series of calculations that converts the message payload into a string of digits.

This value is sent with the payload, so the receiving application has just to recalculate the checksum to verify the integrity of the message. This approach guarantees that data was not modified by a transmission error, but it cannot assure that it comes from a trusted sender.

- *Message Authentication Code (MAC)*: this authentication code is calculated using a *HMAC* combined with a cryptographic key. Since only trusted clients know the secret key, we can discover if a message comes from a trusted client or not.

HMAC is very fast compared to digital signatures.

Clearly, this second authentication way involves a key, so there must be a provisioning way.

- *Digital signature*: it is a code generated by a public key encryption algorithm. More in details, the client signs the message with its private key and the broker validates the signature using the public key of the client.

This approach is the most secure because the only way to sign a message properly is knowing the private key, which is known by the real client only, but at the same time the broker has not to know it too.

So, this approach guarantees the same things of the previous one plus the certainty that a specific user sent the message.

The difference with respect to the previous approach is that the secret key was shared to all clients, while in this case each client has its own private key, but the broker is still able to decrypt their messages using their public key.

Secrecy combined with integrity gives the system a robust security: secrecy prevents attackers to see our things, and integrity prevents attackers to alter them without being discovered.

3.6 MQTT combined with an SDP

In this section we describe a sophisticated solution to implement a very secure MQTT system; since this architecture's implementation seems to be a bit difficult, it should be taken into consideration only if your system needs to be very protected.

3.6.1 What is an SDP?

An *SDP* (*Software Defined Perimeter*), also known as *Black Cloud*, is an emerging security architecture, a bit difficult to implement and manage, that restricts accesses and connections among allowed devices.

Once, companies used to define a local physical network perimeter in order to protect their applications from external threats.

Nowadays, digital progress has significantly broadened these boundaries, and IoT is the biggest proof of that, in fact having sensors scattered in different places of the world make it impossible to define a local physical network as it used to be.

SDP relies on a strategy based on “*hidden-resources*”, or “*non-visibility*”, to provide secure access to devices and applications.

A traditional internal network separated by the external world has a well-defined perimeter and makes use of some firewall functions denying the access to external users, but allowing internal ones to communicate with the external world.

Instead, an SDP is a scalable solution able to provide a secure and authorized access to resources in a perimeter in continuous evolution.

We may think to a network protected by an SDP like a private and exclusive society distributed all over the world, where everybody would like to access, but to which only authorized persons will access. [12]

It is important to say that SDPs are not strictly connected to IoT technologies, but they can be used in different fields.

3.6.2 How does an SDP work?

Devices trying to access are first identified, then, once they have been accepted within the protected network, they are subjected to a further verification process, an authentication with credentials, and finally are assigned permissions based on their authorizations.

Let's see more in details how it works.

First of all, an SDP expects a secure connection, therefore encrypted, based on a *“need-to-know”* logic, that allows only identified, authenticated and authorized clients to access network resources.

There exist many different SDP structures, as we will detail in the following, but they all refer to the idea of *“hide to protect”*.

Figure 3.6 shows an example of a possible SDP structure.

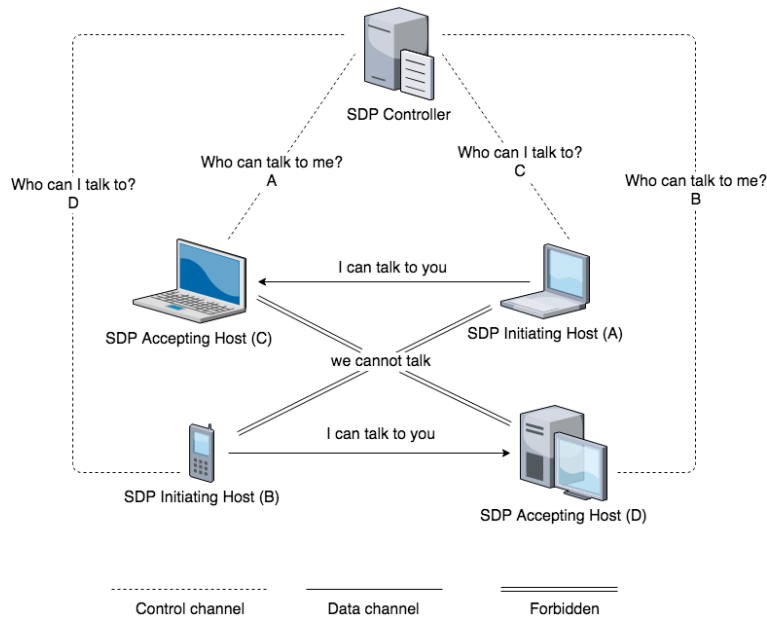


Figure 3.6: Example of SDP structure

There are two components: *SDP Hosts* and *SDP Controllers*; SDP Hosts can either initiate or accept connections, and these actions are managed by interactions with the SDP controllers through a secure control channel. This is a possible SDP workflow:

1. One or more Controllers are brought online and connected to the appropriate authentication/authorization services
2. Hosts, after being brought online, connect and authenticate to the controllers and, for the moment, they do not accept to communicate with any other host
3. Controllers determine a list of Accepting Hosts to which the Initiating Hosts are authorized to communicate
4. Controllers instruct the Accepting Hosts to accept connection from the Initiating Hosts and gives these last ones the list of Accepting Hosts. Since we said that communication is encrypted, policies containing information about encryption are sent to both Accepting and Initiating hosts.

5. Now Initiating Hosts can communicate with Accepting Hosts.

3.6.3 How can this architecture be combined with MQTT?

We have seen that MQTT allows clients to specify username and password while connecting to a broker.

By using an SDP, the pair username/password could be replaced using the so-called *Single Packet Authorization (SPA)* [13].

What is a Single Packet Authorization? [14]

SPA is a technique to securely get access to certain service ports, communicating across closed firewall ports.

So, the initial idea is to make services inaccessible to the outside keeping all the server's ports closed, or at least those ones where services to protect are listening on.

SPA is a kind of evolution of an older system called *Port Knocking*, in which we are able to open some closed firewall ports from the outside by sending connection attempts to a pre-established sequence of closed ports; then the firewall updates its own rules in order to allow the sending host to connect to the desired port.

This old system is basically flawed and vulnerable to replay attacks, as we can see in Figure 3.7, in fact an attacker can sniff the attempts and replay them, obtaining the access to the service's port.

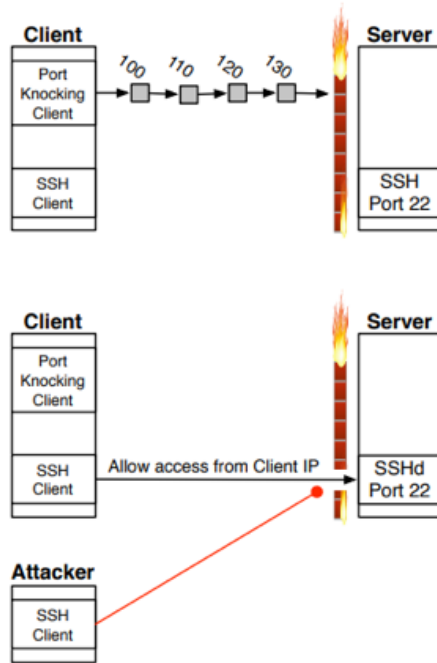


Figure 3.7: Port Knocking Replay attack

SPA and Port Knocking have the same goal but quite different delivery mechanisms: in this newer system, as the name suggests, all of the necessary information, the so-called “knock”, is encoded in a single *Authorization Packet*. This AP can be sent using TCP, UDP or ICMP, but usually these last ones are preferred.

An AP has the following simplified structure:

- **Timestamp:**1523357581
- **Client IP:** 2.225.190.3
- **Password:** somethingsecure

Then, what is actually sent to the server is a resulting MD5 hash calculated in this way:

`MD5($timestamp:$client IP:$password) = 35b45e73c99905b675ffb05b78714eb9`

Once the server received the packet, a SPA daemon will recalculate the hash starting from the password, which must be known from both the client and

server, the IP address of the client and the current timestamp.

If the resulting hash matches the received one, then the firewall allows that specific client IP to connect to the desired port, otherwise no action is performed.

So, we have some advantages using SPA:

- we do not have to respect a certain packet delivery order.
It does not seem to be such a big problem, but it is not so unlikely to make it wrong since the server does not send any acknowledge packet back while receiving knock
- the timestamp and client IP within the AP prevent replay attacks
- the use of the password increases security, but clearly there must exist a way to share it with authorized clients
- encoding the packet using MD5 makes almost impossible for attackers to steal the secret password, when the password is properly chosen

All these precautions make attackers' life very hard.

Coming back to MQTT, we said that username and password could be replaced with an SPA.

Applying what we have just discussed to MQTT security, we may think to hide devices, so that they cannot be seen by attackers, and let them communicate just with pre-established brokers.

On the other side, we will let brokers communicate only with clients who send a valid Authorization Packet.

To make it a bit more secure, we may think to give the broker a list of authorized IP addresses, and reject the communication with any IP not contained in this list, even if able to send a proper AP.

Clearly, the bad aspect of this restriction is that it is less scalable and IP addresses can be spoofed.

Once a (valid) client have sent a proper AP, the server hosting the broker will allow it to connect to the MQTT dedicated port, and finally the broker will receive messages from it.

The MQTT port will remain open for a certain (little) period of time, after

which the client will have to send another AP to get authorization.

We notice that this system is robust even without TLS, and this might be the proper choice to authenticate devices unable to support it, but the problem persists after the authentication: it is true that only messages coming from trusted clients are received by the broker, but the messages following the authentication are still vulnerable to MITM attacks.

As always, using TLS is the best solution, but if not emplyable we may face this problem with a workaround, but keeping in mind that *it should be considered only if TLS is absolutely impossible to use*: since authorized users already know the secret password, using it as input for a key derivation function to obtain a cipher key might be an idea.

3.7 Best practices

Combining all the things we have discussed so far, we can try to list some best practices to consult before creating a secure MQTT system.

- Protect communication using encryption; use TLS whenever possible, otherwise encrypt sensitive data and grant data integrity using one of the techniques we have previously described.

If we use TLS, we must remember that commonly used TLS is already secure enough, but it only proves the identity of the server to the client using a X.509 certificate, while the authentication of the client to the server is left to the application layer.

If we want to strengthen even more this mechanism, we may think to use mutual authentication, so, as already discussed, the client will send its own X.509 certificate to the server to prove its identity.

This choice can afflict performances.

- Request clients authentication.
- Disallow client authentication through clients identifiers.
- Disallow access to system topics. They can contain information about the connected clients, including their identifiers.

- Define properly the authorizations to assign to clients, such that once a server accepted a client connection, this last one must have access only to authorized resources, that are brokers, specific topics and, even deeper, actions allowed on these ones (publish, subscribe, LWT, retained messages).
- Use *Throttling*, that means limiting the total incoming bytes per second on a global or a per-client basis, in order to spare bandwidth.
- Think about a good load balancing: a large MQTT system may distribute the traffic to multiple MQTT brokers rather than just one, to avoid work overloads.

There are many tools to automatize this mechanism, called load balancers and most of them are also able to detect unusually high traffic and block it.

- Define the maximum message size (the default one is 256 MB)
- Choose the proper QoS level according to your needs.
- Check you server and its operating system.
- Keep software and libraries updated.

If you also plan to combine MQTT with an SDP, you should:

- Hide devices and limit access to resources/services using SPA with a secure enough password (in this case, secure enough means not vulnerable to brute force and dictionary attacks: attackers must not be able to get accepted from the server trying a huge number of generated passwords or taken by a list)
- Define properly who can communicate with whom: specifically, servers hosting brokers must accept only connections from clients able to send a proper AP. As already said, if our goal is to have a very strong system, the server can keep a list of accepted client IPs, and immediately truncate connections from unregistered ones, even if valid APs are sent; the same is true for clients, so each client must have a list of servers with which is allowed to communicate.

- Define an authentication password to specify within the APs, and how to share it with registered clients at the beginning
- Use tools to detect brute force/dictionary attacks and lock out attackers.

3.8 Homemade secure MQTT system

This example can be seen as a kind of practical guide of how to build a *simplified* homemade secure MQTT system.

Before starting, let's see a list of the technologies we are going to use:

- OS: Ubuntu 16.04 Desktop
- Programming languages: Python, NodeJS, Bash
- Redis, Iptables, openssl, Nmap

By using this minimal set of technologies, we can build a quite secure MQTT system. In details, our goals are:

- creating a simple MQTT broker
- making server's resources/services hidden to everyone and accessible only by registered, authenticated and authorized client IPs, by using the SPA strategy we have discussed before
- implementing a bruteforce/dictionary attack detector, able to block attackers
- making clients talk only with the server
- encrypting MQTT messages with TLS
- looking at an example of publish/subscribe specifying the desired QoS level and retained messages

Clearly, *this is just a simplified demo*, therefore we will make some assumptions along the way.

Let's start understanding how to hide resources.

3.8.1 Hiding resources

The first step for the server to avoid connections from/to undesired client IPs is to properly configure its firewall, adding the correct constraint rules. Assuming our tiny system is composed by the following actors:

- Server (*192.168.1.126*)
- Client 1 (*192.168.1.6*)
- Client 2 (*192.168.1.7*)

Let's start creating a simple text file containing the list of registered client IPs; let's call it "`trusted_client_ips.txt`".

Then, we move on blocking all the incoming and outgoing connections, except for the ones from/to the registered client IPs on the SPA dedicated port.

To do that, we simply write two bash scripts to properly configure the server and clients' firewall using nothing more than iptables.

```
1 #!/bin/sh
2 iptables -F
3 iptables -P INPUT DROP
4 iptables -P OUTPUT DROP
5 file="trusted_client_ips.lst"
6
7 while IFS= read line
8 do
9     iptables -A INPUT -s $line -p tcp --dport 62015 -j
        ACCEPT
10    iptables -t filter -A OUTPUT -d $line -p tcp -j
        ACCEPT
11
12 done < "$file"
13
14 iptables -nL
```

Code Listing 3.1: Server's firewall configuration script

The code above shows the bash script used to configure the server's firewall; the client's script is almost identical, but *OUTPUT* becomes *INPUT* and viceversa, finally the filename changes to "*trusted_servers.lst*".

The dedicated port is the only open port on the server, running the Python daemon responsible to accept only registered IPs messages and verify if they are valid APs.

3.8.2 SPA mechanism implementation

The SPA daemon is a simple Python server which accepts only connections from authorized client IPs and discard all the others.

Once the daemon receives an AP, it simply uses the pre-established password, the IP address of the sender and the current timestamp to generate the expected MD5, then it compares it with the received one, and if they are equal the received AP has to be considered valid.

All of this stuff can be done in few lines of code:

```
1
2 service_port=8883
3 log_filename="SPA_daemon.log"
4 SPA_passwd="supersecretpassword"
5
6 def grantAccessToClient(client_ip):
7     cmd1="iptables -I INPUT -s %s -p tcp --dport %s -j ACCEPT"
8         %(client_ip,service_port)
9
10    cmd2='echo "iptables -D INPUT -s %s -p tcp --dport %s
11        -j ACCEPT"|at now+2 minute;' %(client_ip,service_port
12        )
13
14    os.system(cmd1)
15    os.system(cmd2)
16
17 def checkAP(received_md5,client_ip):
18     timestamp = str(time.time())
19     timestamp = timestamp[0:timestamp.find('.')] # avoid ms
20     str_to_encode="%s:%s:%s" %(timestamp,client_ip,SPA_passwd)
21     expected_md5=hashlib.md5(str_to_encode).hexdigest()
22
23     if(expected_md5==received_md5):
24         grantAccessToClient(client_ip)
25     else:
26         log_file.write("[%s] %s - Invalid MD5 received\n"
27             %(timestamp,client_ip))
28
29     log_file.seek(0)
```

The idea is pretty simple: if the daemon receives a valid AP, it adds an Iptables rule to allow the client to access the service port, otherwise it does nothing.

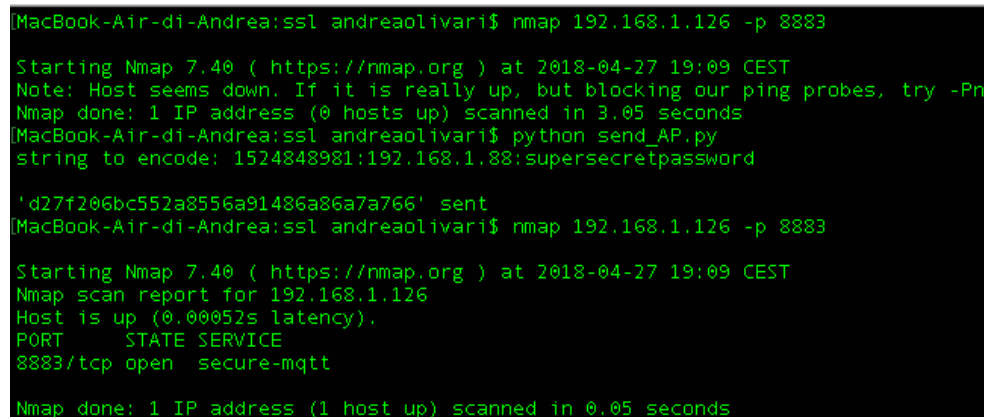
Note: since we want the access to the service to be limited in time, we run the command “at” which gives us the opportunity to run a command after a certain period of time (in the example it is 2 minutes, but in a real scenario it should be larger). Another thing we should notice is that we log only the negative cases, and we do this because there is another script which makes use of this log file to detect and block bruteforce/dictionary attacks in real-time.

This detector implements a very simple but effective detection strategy: when it reads a new line of the daemon’s log file, it takes the client IP address and checks if the number of invalid APs received by it in the last `TIME_RANGE` minutes is larger than `MAX_REQUESTS`; if so, that client IP is locked out for `BAN_TIME` hours, and to do that it simply adds an iptables rule again. This last rule will expire automatically using once again the command “at” to delete it from Iptables.

Of course, it is important to define properly these constants: let’s say, for instance, that we may accept at most 5 invalid APs for IP in the last 30 minutes and ban the user for 6 hours.

We skip the analysis of the SPA client because it is nothing but a Python script sending well-formed requests to the server’s daemon through sockets.

Instead, we can see in Figure 3.8 a proof of how this SPA mechanism works well, using Nmap to scan the service port, which is the 8883, on the server before and after sending a valid AP.



```
[MacBook-Air-di-Andrea:ssl andreaolivari$ nmap 192.168.1.126 -p 8883
Starting Nmap 7.40 ( https://nmap.org ) at 2018-04-27 19:09 CEST
Note: Host seems down. If it is really up, but blocking our ping probes, try -Pn
Nmap done: 1 IP address (0 hosts up) scanned in 3.05 seconds
[MacBook-Air-di-Andrea:ssl andreaolivari$ python send_AP.py
string to encode: 1524848981:192.168.1.88:supersecretpassword

'd27f206bc552a8556a91486a86a7a766' sent
[MacBook-Air-di-Andrea:ssl andreaolivari$ nmap 192.168.1.126 -p 8883
Starting Nmap 7.40 ( https://nmap.org ) at 2018-04-27 19:09 CEST
Nmap scan report for 192.168.1.126
Host is up (0.00052s latency).
PORT      STATE SERVICE
8883/tcp  open  secure-mqtt
Nmap done: 1 IP address (1 host up) scanned in 0.05 seconds
```

Figure 3.8: Nmap scan before and after sending a valid AP

Of course, we have to run a broker on the port 8883, and I have done that using an easy-to-use nodeJS module, called *Mosca* ².

²<https://github.com/mcollina/mosca>

Mosca gives the user the opportunity to use TLS as well as credentials and two storing methods: MongoDB or Redis.

I opted for *Redis* (*REmote DIctionary Server*), which is nothing but a key-value store with optional durability, often used to cache server content that needs to be accessed quickly.

We skip a deep analysis of the broker's code since it is very similar to the example you can find in the module's page, except for some security constraints we added:

```
1 var moscaSettings = {
2   port: 8883,
3   backend: listener,
4   persistence: {
5     factory: mosca.persistence.Redis
6   },
7   secure: {
8     keyPath: "broker_cert.key",
9     certPath: "broker_cert.pem"
10  }
11 };
12
13 var server = new mosca.Server(moscaSettings);
14
15 server.on('ready',function() {
16   console.log("MQTT broker is up and running");
17   server.authenticate = function(client,username,
18     password,callback) {
19     callback(null,(username===broker_username &&
20       password.toString('ascii')===broker_password))
21   }
22 });
```

Code Listing 3.2: Mosca broker security settings

We have added the authentication control and enabled TLS specifying two files, *broker_cert.key* and *broker_cert.pem*, where the first one contains the broker's certificate private key, while the second is the server certificate to be sent to the client, containing the public key of the server.

Clearly, in order to use this certificate, we first have to create it: it is possible

to create self-signed X.509 certificates using the Linux command “*openssl*”.

In this demo our MQTT broker uses a self-signed certificate generated locally and I properly configured clients to consider it as trustable, but *in a real scenario we have to rely only on trusted certification authorities, so this is absolutely forbidden.*

At this point, MQTT clients can be implemented using Python once again, and more in details its library *paho.mqtt* (available also in Java and other programming languages), to simply connect to the broker and publish/subscribe to topics.

This library is easy to use, and allows us to set TLS, credentials, QoS levels and retained messages in no more than one line of code for each feature:

```
1 def publishMsg(client,topic,message,qos_level,retain_msg):
2     client.publish(topic,payload=message,qos=qos_level,
3                     retain=retain_msg)
4
5 client = mqtt.Client()
6 ssl_version = ssl.PROTOCOL_TLSv1_2
7 client.username_pw_set(brk_username,password=brk_password)
8 client.tls_set("broker_cert.pem",cert_reqs=ssl.CERT_REQUIRED,
9               tls_version=ssl_version)
10 publishMsg(client,topic_pub,"online",1,True)
```

Chapter 4

Real-world examples

I did my internship working for four months with a company called ABO DATA, which deals with IoT technologies.

The company enables customers to customize and manage IoT business applications: most of the times we talk about customers' devices/sensors which send their measurements to a cloud or a centralized system, then these measures are computed and exploited to do something else related to customers' business.

So, during my internship I often had to deal with tasks related to receiving data from a source and their subsequent forwarding to a destination, like a bridge, typically using two protocols: HTTP(s) and MQTT.

In Section 4.1, I would like to discuss two real-world cases I have worked on, which should make us reasoning about how a production system should be thought and implemented.

4.1 Case 1 - Real-world MQTT-based industrial system

This is a real-world scenario involving an important company, whose name cannot be disclosed for reasons of privacy, partner of several automotive, pharmaceutical, writing tools and watchmaking industries, that produces and distributes automated systems as well as high-precision and efficient cutting tools.

This company recently decided to build a reliable monitoring and management system for their machines using a *cloud* architecture.

The requirements were to collect data from machines' sensors in order to create a so-called:

- *Local Control Room*, which allows the company's customers to check the performances, real use and health status of their machines, in real-time
- *Centralized Control Room*, which allows the company itself to monitor their resources operating with different customers in order to collect historical data, useful to make analysis and comparisons

These monitoring systems collecting tons of data are often used by companies and industries to drastically change and improve their maintenance approach. A *Time-based* maintenance approach is the simplest one, and unfortunately the most used, so when a machine breaks down, the customer contacts the support company to receive assistance.

The use of a monitoring system introduces a new maintenance approach, called *Condition-based*, thanks to which the corrective maintenance is considerably reduced; in fact, the continuous monitoring of the health status of the machines allows to intervene before the failure occurs.

Finally, in the last few years a new approach, called *Predictive*, was born; this approach is based on the possibility of predicting possible failures in a smart way, using the huge amount of data received as input for Machine Learning algorithms.

Clearly, both the second and third approaches can be seen in two different ways. The customer uses data from its

- Local Control Room, and when he (or his ML algorithms) detects an imminent breakdown, he contacts the company assistance
- Local Control Room for other purposes, and let the company predicting possible failures using Centralized Control Room's data.

Now that we know what a monitoring system can be useful for, let's see the structure of the system mentioned above.

This company provides two different monitoring systems for two different possible scenarios:

1. *Factory*
2. *Connected machines*

4.1.1 Factory

This first scenario expects the customer to be an industry, having its own private network to which the company's machines (equipped with internal sensors) are connected.

As we can see in Figure 4.1, the customer runs a MQTT broker on a local server, on which the company's sensors publish their data.

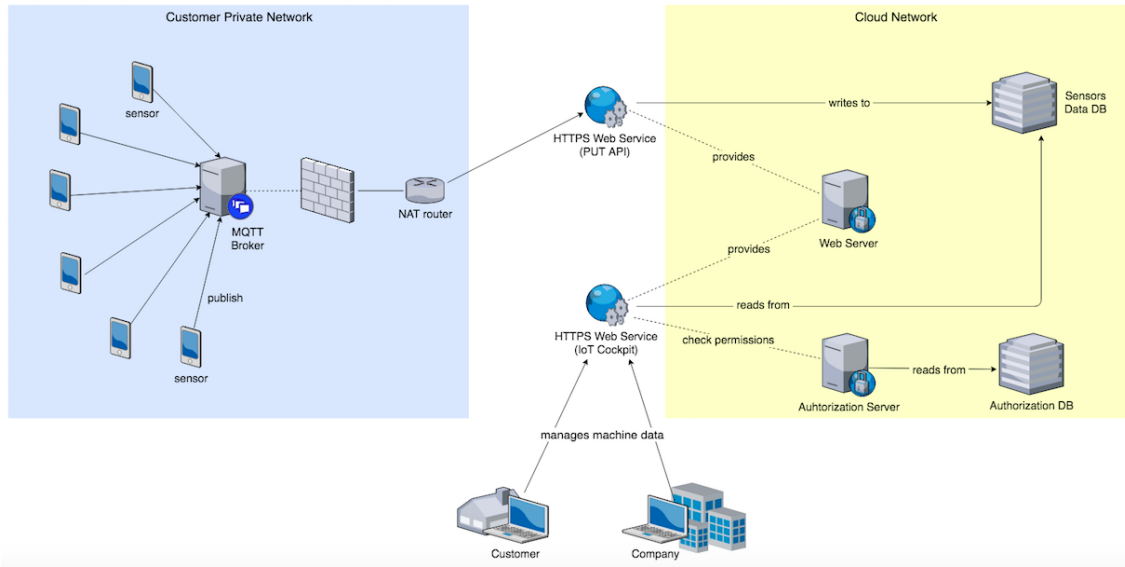


Figure 4.1: Factory Architecture

This MQTT broker is developed and provided by a third company, which is the owner of the cloud network as well as the provider of the monitoring service, and in our case we are talking about *SAP*¹.

In this specific scenario, MQTT communication is not encrypted and there is no required authentication system; simply, there are several open topics, one for each sensor, which publishes its measures at regular intervals of time. It is easy to understand that this is a risky choice for an industrial system architecture, that should be considered only if we can totally trust all the devices connected to the internal network.

¹<https://www.sap.com/>

Clearly, to trust all of them we have also to be sure that the internal network protection is secure enough: having a wired network definitely helps, since an attacker must have physical access, but of course this is not always possible, especially when devices belonging to the same network are not located in the same place.

In that case it is necessary to create a secure wireless network protected by WPA2 and a strong password.

As shown in Figure 4.1, the customer private network increases its security putting a firewall in front of the server hosting the MQTT broker.

This firewall is properly configured to hide all the devices from the external world, blocking all the incoming and outgoing connections.

Actually, there is only one outgoing connection allowed, that is the one that allows the server, on which the broker runs, to send the collected data by the sensors to the cloud.

The outgoing connection uses an *HTTPS API* (simply a web service), provided by a webserver belonging to the Cloud network, so to the SAP network, whose job is to receive data from sensors and write them on the database. Having to do with a secure HTTPS connection we know that data is transferred securely, therefore we can assume that the customer-side is properly protected.

There is just one remaining risk to be removed: only trusted clients' requests must be accepted by the *PUT API*, hence those coming from the company customers' broker servers.

To do that, mutual authentication is employed, so when a customer's broker sends sensors' data to the API it will send also its own X.509 certificate to the remote webserver to prove its identity.

Note: this is the solution chosen and implemented by SAP, but an alternative one could be to include authentication credentials in the POST content, avoiding the mutual authentication mechanism, which, as already seen, can afflict performances.

We should notice that the Factory architecture brings a great benefit in terms of certificates management: since all the customer's machines/sensors belong to the same private network and only the server is allowed to establish outgoing connections, then just one X.509 client certificate is necessary for all of them.

One last thing to notice is that, considering what has been said so far, it seems that the customer has no way to verify if the data sent to the API has been correctly received and stored on the Cloud.

This is not true, in fact the customer, as well as the company itself, can verify the health status of their machines using another secure web service, called *IoT Cockpit*, provided by the same webserver.

This service simply provides an intuitive dashboard to the user, by which he can stay updated and check their machines and measures; the difference from the other API is that this last one can be accessed by everyone but it requires a user authentication.

4.1.2 Connected Machines

This second system architecture deals with a different, and more difficult to protect, scenario.

In this case, there are several customer's machines located in different places, probably far from each other, therefore no common and secure network protecting them, but simply standalone machines and their related sensors.

We can see this architecture in Figure 4.2.

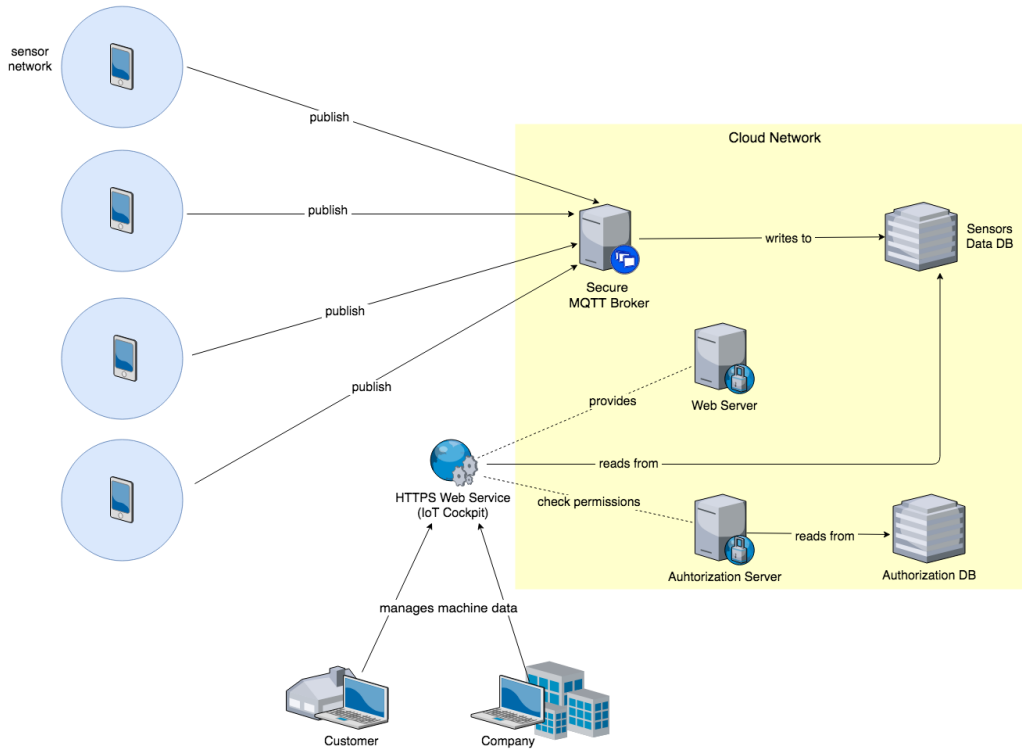


Figure 4.2: Connected Machines Architecture

It is obvious that this fact changes everything: it would not make sense to give each machine a local server running an MQTT broker to which send data, and then forward it to the cloud through it as before.

This procedure should be considered only when there are several machines close to each other which can share a private network in order to create a singular exit node towards the cloud network.

Having customers' sensors spread all over the world is a very common scenario; just think about sensors applied on transports, for instance sensors used to check the health status of car tires.

So, for all these cases there is only one way to communicate: having a remote MQTT broker on the cloud and let each sensor publishing its own data on it.

Clearly, this time it is necessary to use the secure version of MQTT, therefore

relying on TLS.

As before, it is also necessary to verify if sensors' data actually come from trusted senders; we notice that in this case MQTT client authentication would be enough, but practically mutual authentication is employed again.

Here we have a practical example of what we have discussed before in Section 3.8: I recommended you to trust only certificates approved by reliable certification authorities and never trust self-signed certificates.

This last architecture helps us to understand that, in addition to the fact that it is not good to trust self-signed certificates, forcing the server to keep a potentially huge number of certificates is not convenient and totally pointless; therefore, the right way is to trust client's certificates which are trusted by certification authorities we trust.

After having seen these two real-world architectures, we can conclude that there is no unique perfect solution covering all cases, but it is necessary to find the most proper one for the specific scenario we are dealing with, considering all the relevant factors, such as networks topologies, performances, secrecy, integrity, users' needs, maintenance.

4.2 Case 2 - Every aspect is important

This second case is certainly less detailed than the previous one, but I would like to quickly discuss it because it helps to show how every aspect of a system should be considered important as the others, and as trivial errors can lead to serious consequences.

In the previous case we have seen that the monitoring system was provided by SAP; we can say that, currently, the company relies on the SAP platform for most of its projects, and SAP structure is very solid and secure, paying attention to be up to date with the newest security measures to take. For more information, you can consult their official documentation at <https://www.sap.com/corporate/en/company/security.html>.

This is a real case, but again, for obvious privacy reasons, I cannot disclose to the customer I am going to talk about.

In this project I had to take some real-time data, stored on a remote server belonging to the customer, and send them to another remote server through a secure MQTT channel.

I started Wireshark to sniff data and see if it was possible for an attacker to alter the measures, but without success as the MQTT channel used TLS.

What we did not say is that the protocol used to recover data from the source server was FTP, which is known to be extremely vulnerable since it sends data in plain-text.

So, the moral of this brief story is that it is fundamental to consider every security aspect of the system before developing it, because all the efforts involved in the successful implementation of a communication channel can be undermined by a very bad implementation of another communication channel.

Part II

ZigBee

Chapter 5

Protocol Overview

5.1 Introduction

ZigBee is a low-cost protocol developed by ZigBee Alliance, for low-power wireless personal area networks that interconnect devices primarily for personal uses within a short range (10-100 meters).

This communication system is less expensive than Wi-Fi and it has a larger range than Bluetooth, besides its range can also be extended using routers.

Nowadays, ZigBee is used in industrial automation, where industries exploit it to manage and monitor critical equipments, but in particular this protocol is perfectly suited for controlling home appliances remotely. We may think to TVs, surveillance devices, locks, thermostats, lighting system, garage door, windows, blinds, speakers, as well as any other home system that can be controlled remotely.

Currently, ZigBee is mainly used to build *Smart Homes*.

In section 5.2 we start to analyze the architecture and the basic concepts of this protocol. Security issues are described in the next chapter.

5.2 Architecture and basic concepts

As we can see in Figure 5.1, a ZigBee system consists of three types of devices:

- *Coordinator*: they manage the overall network and they are responsible for starting it, allowing other devices to join it, keeping track of them and configuring the security level of the network.

A coordinator is considered as the *Trust Center (TC)*, which performs security control of the network, storing and distributing the keys.

For all these reasons, the coordinator never sleeps, instead it must be continuously powered.

- *Routers*: they are intermediary devices responsible for routing packets between end devices or between an end device and the coordinator. Routers, as well as end devices, need to be accepted in the network by the coordinator.

Routers have two things in common with the coordinator: 1) they cannot sleep, 2) they can give permissions to new devices to join the network.

- *End devices*: they are low-power, or battery-power, devices (usually sensors) that can communicate only through their parent nodes.

Unlike routers, end devices cannot route traffic and they can go to sleep to reduce power consumption.

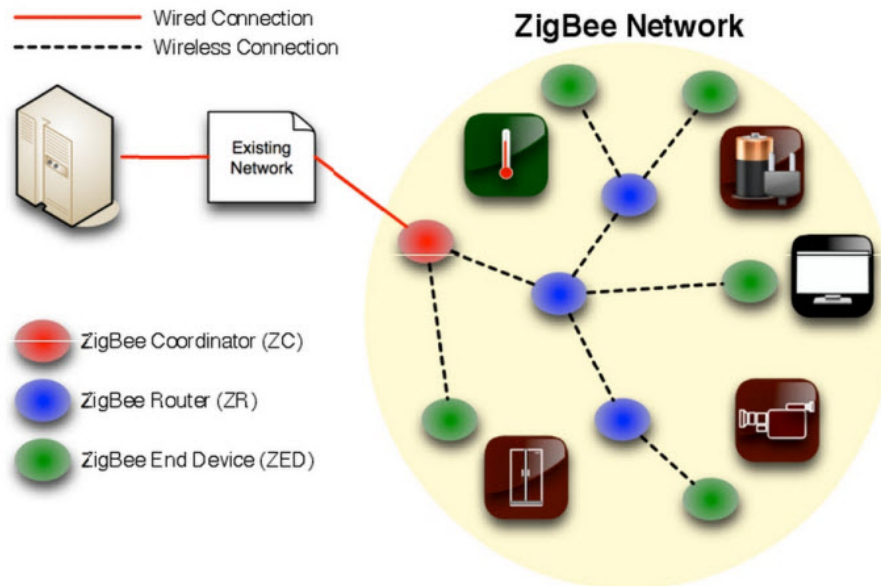


Figure 5.1: ZigBee Architecture

There is no required number of specific components for a ZigBee network, instead the number of coordinators, routers and devices depends on the network we want to build and its topology (*star, cluster tree or mesh*).

5.2.1 ZigBee network topologies

Figure 5.2 shows the three possible ZigBee network topologies: star, tree and mesh.

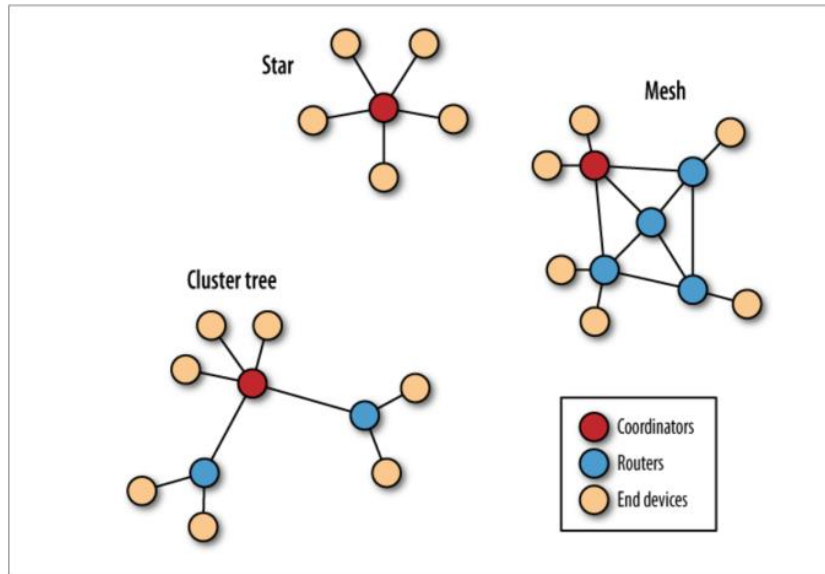


Figure 5.2: ZigBee Topologies

- *Star*: in this topology there is only the coordinator responsible for initiating and managing the network devices; since every device talks directly with the coordinator, we consider the other devices as end devices, therefore no routers in this topology.

This topology is mainly used when all the endpoints need to communicate with a central controller, so it is common in industries.

Of course this topology is very simple, but its simplicity is also its greatest weakness, in fact the coordinator is a single point of failure, because if it breaks down the whole network goes down.

- *Tree*: this topology has a root node, represented by the coordinator, which is responsible for establishing the network and choosing the proper key network parameters.

This time we have even routers, which can have either the coordinator or another router as parent node, and they are responsible for routing data packets through the network using hierarchical routing strategy.

This solution is not so better than the previous one because we still have a single point of failure, given by the coordinator, besides children nodes become unreachable if their parent node breaks down.

- *Mesh*: this topology has, as usual, a single coordinator, several

routers to extend the network and, of course, end devices.
Once again, the coordinator establishes the network and is responsible to choose the network parameters.
There is no more a single point of failure because in this case we have multiple paths to reach a node, therefore a more robust protection from link failures.
The only drawback of this topology is that is complex and more difficult to set up than the previous ones.

Mesh topologies are more robust than the others, therefore they are the most used.

5.2.2 Protocol Stack

The protocol is based on the IEEE 802.15.4 standard, which provides the physical and MAC layers, while the above layers are given by ZigBee. Each layer provides a set of services exposed to the upper layer.

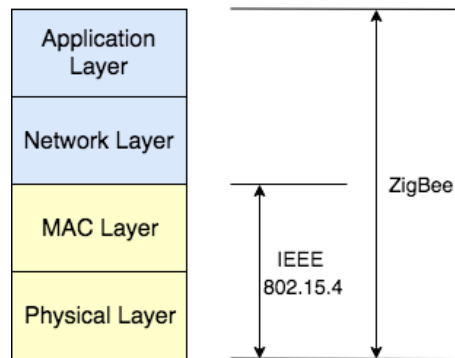


Figure 5.3: ZigBee Protocol Stack

Let us briefly see them:

- *Physical Layer*: it provides the basic radio communication capabilities, such as channels, modulation, demodulation, transmission rate, etc.
- *Medium Access Control layer (MAC)*: it manages data transmissions between neighbor devices (point to point). It includes services like transmission retry and acknowledgment management [15].
- *Network Layer*: it adds routing capabilities that allows data packets to traverse multiple devices (multiple hops) to route data from source to destination, in order to create more complex topologies.
- *Application Layer*: it provides device and service discovery features, applications supporting layer primitives used by devices, capabilities to manage the security policies and configuration of a device. Finally, it collaborates with the Network layer to establish and manage cryptographic keys, and provides primitives for their management.

Chapter 6

Security Overview

6.1 Introduction

As already said in the MQTT part, IoT will play a primary role in the coming years and for this reason it is fundamental to take care about its security.

While MQTT security is mainly aimed to help companies and industries to transfer their production data reliably, ZigBee has more to do with personal area networks that interconnect devices primarily for personal uses, hence high personalized and sensitive data.

Just think to a smart home based on ZigBee: a lot of user sensitive information can be leaked by an attacker if the security aspect is neglected.

Data theft is not the only problem, in fact an attacker would cause a lot of problems to the user if he could control his household appliances.

Just think about a thief, good enough to find and exploit a ZigBee vulnerability and turn off the surveillance devices.

In the next sections, we are going to discuss the security measures offered and adopted by the ZigBee standard; more in details, we will analyze its security models, keys (types, generation and management), authentication mechanisms and protection from some famous attacks.

6.2 Security assumptions

Being ZigBee a simple as well as low-cost communication protocol, it relies on a symmetric-key cryptography to protect network messages and devices, precisely AES 128 bits, avoiding heavier encryption system, like the asymmetric one; therefore, both the sender and recipient of a protected transaction need to share the same key.

All the ZigBee security features we discuss are strictly related to the cryptographic (symmetric) keys used by the protocol; in order to talk about them, we must first take a look at the following list of assumptions [16] on which they are based:

- communication between different stack layers on the same device is not encrypted
- cryptographic keys are securely stored in tamper proof hardware devices
- keys are always transmitted encrypted, so never revealed
- cryptographic mechanism and security policies are properly implemented
- availability of almost perfect random number generators

6.3 Security models

ZigBee provides two network security architectures, or models, visible in Figure 6.1: *centralized* and *distributed*.

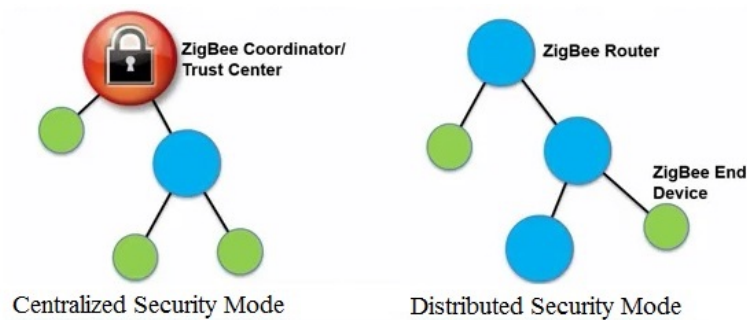


Figure 6.1: ZigBee Security Models

The difference between them concerns only the way they accept new joining devices into the network and how messages are protected.

Each network has a key used to encrypt broadcast messages, called *network key*, and a key shared by any two devices to communicate, as well as to encrypt and provide the network key, called *link key*.

The idea of the *distributed security model* is pretty simple: there is no coordinator, but only routers and end devices.

In this model, we totally rely on routers to accept new routers and end devices within the network as well as to generate and distribute network keys.

This security model expects all the devices to share the same network key and to be pre-configured with the same link key that will be used to send the encrypted message containing the network key.

All the devices will talk to each other encrypting messages with the same network key, therefore everyone accepted within the network can read everyone else's data.

The *centralized security model* is a bit more complex, but even more secure. In this case we have routers, end devices and the coordinator, that from now on we will call *Trust Center (TC)*.

This time the responsible for the management and acceptance of routers and end devices joining the network as well as the creation and sharing of the keys is the TC.

Besides, devices have not to be all pre-configured with the same link key and they have not to communicate with each other using the network key to encrypt messages.

The TC also creates and provides a new network key periodically, in order to protect the network from possible attackers, who can try to steal it.

While the first model is certainly simpler it is less secure, and almost unused, so we focus on the second one.

We have already mentioned that a ZigBee network exploits two main types of keys to communicate securely: *network keys* and *link keys*.

Let us see them in details:

- the *network key* is a 128-bit key generated and distributed by the TC to all the network devices in order to allow them to communicate in broadcast securely.

It is clear that the TC cannot send the network key in plain-text, otherwise an attacker could sniff that, making it useless. Sending a key without encrypting the message is against one of the assumptions we have seen in section 6.2, but unfortunately there is one exceptional case in which this happens.

There are two types of network keys, *standard* and *high security*, which differ from each other in the way they are distributed to joining devices. A high-security network key is sent encrypted, while the other one in plain-text.

More specifically, the standard network key is sent in plain in those cases where the end device that wants to join the network has no pre-configured link key. Some TCs decide to deny the access to such devices, while others are more permissive and allow them to join the network, risking to compromise the security of the whole network.

It is important to notice that joining devices can be also pre-configured with a network key, but this is a uncommon practice, since, as already said, in a centralized model the network key is updated periodically by the TC.

- the *link key* is a 128-bit key shared by two devices, so between the TC and another node, or between two nodes.

More in details, we have three different types of link key [16]:

- *Pre-configured Global Link key*: it exists for one reason only, that is encrypting the network key in order to transfer it securely from the TC to the end devices.

As the words *global* and *pre-configured* suggest, this key is the same for all the network nodes and pre-installed in the devices that want to join the network.

The global link key can be defined by the manufacturer, or, even much simpler, by ZigBee.

- *Pre-configured Unique Link key*: same use of the previous one, but different for every node.

- *Trust Center Link Key (TCLK)*: like the last one, this key is used to allow the TC and another node to communicate securely, but the difference is that this time it is not pre-configured.

The TC generates it randomly or derive it from the network key (or from the pre-configured unique link key, if existent) using an hash function.

Without entering in too many details about this hash function, we just say that it takes two 128-bit keys as input and return one 128-bit key as output; the second input key of the function is known only by the TC.

Of course, this generated key must be sent to the interested node, and to do that the TC uses once again the network key or, much better, the pre-configured unique link key, if existent.

A reasonable question is “*why should we generate a TCLK if we already have a pre-configured link key?*”; I did not find a specific answer to this question, but I guess that the reason is that, being the pre-configured link key rarely updated, changing the link key

we use to communicate and avoiding to use our pre-configured link key helps us to protect it and give less chance to an attacker to steal it.

- *Application Link Key*: similar to the previous one, but used by pairs of nodes.

The generation and provisioning of the key is quite intuitive: one of the two devices requests the key to the TC, which is responsible to generate and send it to both of them using their respective pre-configured unique link key or, if not existent, the network key.

Now that we have analyzed the ZigBee security keys and their provisioning, I would like to report, just for completeness, that there exists a (rarely used) extension of the traditional key-transport, which involves certificates. Security keys can be distributed using the *Certificate-Based Key Establishment protocol (CBKE)* [17], which implements a mechanism to allow devices to negotiate symmetric unique keys with the TC, starting from a certificate, signed by a Certification Authority, that both devices must have.

The CBKE mechanism allows the device to start communicating, but even the Trust Center to securely identify it.

The key establishment procedure involves the following four steps:

1. Exchange static data for certificate validation
2. Generate the key
3. Derive a MAC (*Message Authentication Code*) key
4. Confirm the key using the MAC

The key generation and the MAC derivation use the *Elliptic Curve MQV*¹ and a key derivation function.

At the end of this process, both the Trust Center and the device i share a new Link Key LK_i , which is used to encrypt the TCLK they are going to use to communicate.

Now, if two devices i and j have obtained their TC Link Key, and want to communicate with each other, they can obtain their Application Link key by asking it to the TC, as usual, which will generate a random Application Link

¹<https://www.ecmqv.com/>

Key LK_{ij} and send it to both the devices using their respective Trust Center Link Key.

6.4 Protocol Stack Security

In this section we discuss the security measures adopted by the ZigBee protocol stack, which involves the MAC, Network and Application layers.

The IEEE MAC Layer implements security services which are used by the ZigBee protocol in the network and application layers. IEEE 802.15.4 establishes the encryption algorithm to use (AES) when the data has to be transmitted, but the standard does not specify the keys or security levels to use. This is a task concerning the upper layers, hence ZigBee is responsible for that.

6.4.1 802.15.4 Security

Security Control Header

Let's start taking a look at Figure 6.2, representing the structure of a MAC data frame [18].

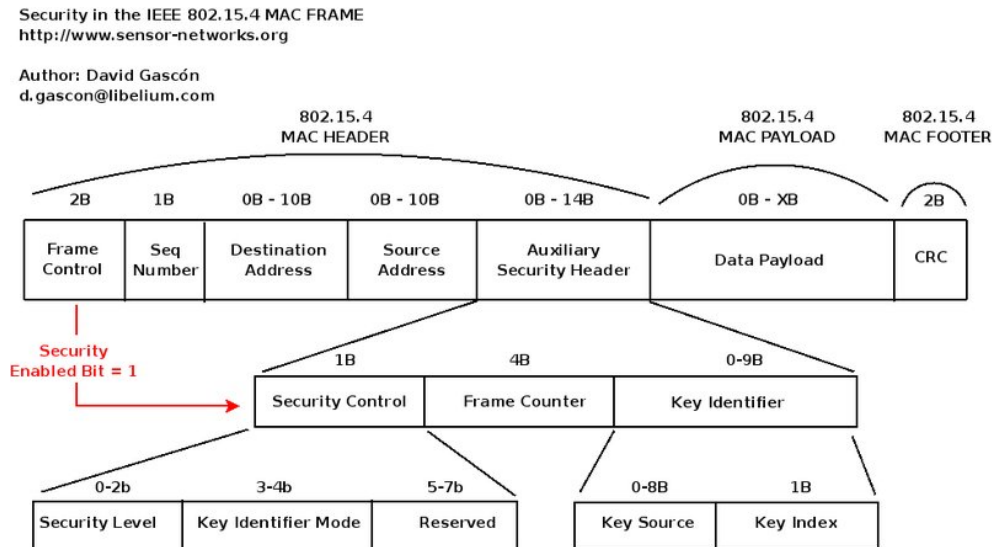


Figure 6.2: ZigBee MAC Frame's Structure

IEEE 802.15.4 MAC frames contain a specific header dedicated to security, called *Auxiliary Security Header*.

This header is enabled only if the *Security Enabled* bit, contained in the *Frame Control* field, is set to 1.

There are three subfields inside the Auxiliary Security Header:

- *Security Control*: it is used to specify the type of protection provided by the network.

The most important subfield of the Security Control field is *Security Level*, which can assume eight different values, each of them providing a different degree of encryption and integrity checks, and is the place where our global security policy is set.

Security Level	Security Attributes	Data Encryption	Integrity Check
0x00	None	N	N
0x01	AES-CBC-MAC-32	N	Y
0x02	AES-CBC-MAC-64	N	Y
0x03	AES-CBC-MAC-128	N	Y
0x04	AES-CTR	Y	N
0x05	AES-CCM-32	Y	Y
0x06	AES-CCM-64	Y	Y
0x07	AES-CCM-128	Y	Y

– *AES-CTR*: data payload is encrypted using AES 128-bit counter mode.

– *AES-CBC-MAC*: data payload is not encrypted, but a Message Authentication Code (MAC) is computed and attached to the end of the data payload.

The MAC is created encrypting information from the 802.15.4 header and the data payload. Its length can be 32 bits, 64 bits, or 128 bits, depending on the specified security level, but in any case it is calculated using a 128 bit key.

– *AES-CCM*: it is the combination of the two previous ones.

More details about this algorithm will be discussed in section 6.4.2.

- *Frame Counter*: it is a simple counter given by the source of the current frame in order to protect the message from replay attacks.

- *Key Identifier*: it contains useful information to know which key we are using with the node we are communicating with.

Access Control List (ACL)

Each 802.15.4 device can have an *Access Control List (ACL)*, which is a list containing the so-called “*trusted-brothers*” and their respective security policies.

More in details, for each trusted brother we store the following information:

- *Address*: the physical address of the node we want to communicate with
- *Security suite*: the encryption algorithm to use (*AES-CTR*, *AES-CCM-32*, etc)
- *Key*: the 128-bit key used in the AES algorithm
- *Last Initial Vector (IV)/ Frame Counter*: this is an incremental value used 1) by the source to encrypt the message and 2) by the recipient as frame counter to avoid replay attacks.

In simple words, when a node wants to communicate with another node (send or receive a message) it looks at the ACL to see if it is a trusted brother; in that case it uses the security measures contained in the respective security policy.

6.4.2 ZigBee Security features

The ZigBee security architecture includes security mechanisms, such as frame protection, at three layers of the protocol stack: MAC, Network and Application.

Each layer is responsible for its own security processing but, as already said, the keys and the security level used by the MAC Layer are established by the upper layers.

Frame Protection

In addition to the security features provided by IEEE 802.15.4 standard, ZigBee adopts *AES CCM** to protect network and application layers frames. As for the MAC layer, frames have an Auxiliary Security Header and a payload field, which can be encrypted and contain a MAC for data integrity. Figure 6.3 and 6.4 show you the structure of these two frames.



Figure 6.3: ZigBee NWK Frame's Structure



Figure 6.4: ZigBee APL Frame's Structure

CCM mode combines *CBC-MAC* and *CTR*: *CBC-MAC* is used to generate a tag for the frame, and *CTR* is used to encrypt the frame itself as well as the generated tag.

CCM mode* is nothing but a minor variation of *CCM*, which includes all the features of *CCM* and, in addition to them, it offers encryption-only and integrity-only capabilities.

Using *AES-CCM** is convenient because it can cover all the security levels provided by the 802.15.4 standard, simply turning on, or off, the integrity-only and the encryption-only capability.

Clearly, the receiver uses *AES-CCM** and the proper key to decrypt the encrypted frame and the tag, then he generates the MAC on his own in order to compare it with the received one.

The key used to encrypt and generate the MAC for a frame, is the one specified in the receiver's ACL; ZigBee simplify the protocol security, using the same key for all the layers.

Replay Attacks Protection

We have already mentioned that ZigBee offers protection from replay attacks, using a frame counter.

More in details, every node in the network has a 32-bit frame counter which is incremented at every packet transmission.

Besides, each device tracks the previous frame counter of every node with which it has communicated because, doing this, if a node receives a packet from another node with the same or lesser frame counter value than it had previously received, the packet will be dropped.

It is important to say that the frame counter is reset to 0 when the Trust Center updates the network key.

Device Authentication

Clearly, only authorized and authenticated devices can join the network.

The Trust Center is responsible to accept new devices that want to join the network, but before doing that the device must be able to receive a network key and set proper attributes within a limited given time to be considered authenticated.

More in details, there are two different procedures for authenticating devices:

- *Standard (Residential) mode*: if the new joining device already has a pre-configured network key (unlikely), it will receive an all-zero network key from the trust center as part of the authentication procedure.

If the new device does not have a network key and not even a link key (which must be known by the Trust Center obviously), the TC sends the network key in plain, which is a quite important vulnerability.

Since the new device does not know the address of the trust center, it will use the source address of the received message to set the trust center address.

- *High Security (Commercial) mode*: in this second, and more secure, mode the Trust Center never sends the network key over an unprotected communication channel.

If the device has a valid pre-configured link key, global or unique, to communicate with the TC, then this last one will send the network key

to it, and will consider it as authenticated.

A global link key requires less memory to the Trust Center, but it is less secure than the unique link key.

Instead, if no pre-configured link key is owned by the device, then the Trust Center expects it to have, at least, the master key by which it is possible to start the key establishment protocol, after that the device and the TC will share the link key.

The new device has a limited time to complete the key establishment, otherwise it has to leave the network and retry the authentication procedure again.

When and if the link key is confirmed, the Trust Center considers the device as authenticated for commercial mode, and can send the network key to it through a secure channel thanks to the just established link key.

This second mode seems to be more secure than the previous one, and surely it is, but it actually has a vulnerability: if the new device does not share a master key with the Trust Center, this last one will send it over an unprotected link.

Network Interference Protection

It might happen that nearby wireless networks, even Wi-Fi or other ones, create physical interferences to our ZigBee network.

Without entering in too many details, it is important to know that ZigBee and IEEE 802.15.4 try to reduce the presence of interferences using low RF transmission power, low duty cycle and the CSMA/CA mechanism.

Often, this is not enough, so ZigBee provides two other strategies to avoid interferences:

- *Collaborative*: ZigBee network and other networks work together; in simple words, when one network is active, the other one remains inactive in order to avoid packet collisions.

Clearly, this method expects the two networks to share a communication channel to manage the collaboration.

- *Non-Collaborative*: in this second method, ZigBee network has no communication channel shared with nearby wireless networks, but it

uses several interferences detection techniques to avoid as many interferences as possible.

We will not discuss the following techniques, since they are outside the context, but for completeness I will provide a list of some of them:

- *CSMA/CA*
- *Signal spreading-spreading methods*
- *Frequency Channel selection*
- *Adaptive Packet Length Selection*

We conclude this section saying that ZigBee takes care about security, as we have understood so far discussing its security measures.

But we have also seen some important vulnerabilities in the exchange of keys, that, just right now, makes the protocol not perfect at all.

This is not surprising since ZigBee was designed to keep devices low-cost and low-energy.

6.5 ZigBee vulnerabilities

The aim of this section is to discuss the most relevant ZigBee vulnerabilities and some practical attacks to exploit a ZigBee network.

Some of them are related to the specific implementation, while some others are strictly related to the protocol itself.

Let's see each of them in details:

- *Key storage*: previously, we mentioned that ZigBee assumes that keys are securely stored within the devices, but this is not always the case.

If an attacker can extract the network key from a device, then all the communications within the network will be at risk.

- *Key transportation*: if the Trust Center operates in residential mode, it could happen that the network key is sent in plain; clearly, this is a quite important vulnerability since we transmit the key shared by all the devices within the network to send broadcast messages with no protection at all, compromising the whole network.

- *Ghost Attack*: frame counter checks allow recipient devices to ignore duplicate messages, but a *Denial of Service (DoS)* vulnerability persists: if an attacker captures an authentic message and replay it, the recipient will discard it but it will spend time and energy to check its validity.

If an attacker performs tons of replay attacks against the same target, the battery life of this last one will be drastically reduced (especially considering that we are talking about low-power devices).

- *Manufacturer-defined Link Keys*: manufacturers set default link keys within their devices, and sometimes these keys are used to join ZigBee networks.

Using manufacturer default link keys is a bad choice, because they are usually publicly available.

- *Association Flooding*: by default, there is no integrity checks on acknowledgment packets, therefore an attacker can send fake ACKs to devices, and they will be considered as authentic.

An attacker, within an already compromised network, can keep on

reading messages sent by a device even if the respective recipient is not responding, simply sending fake ACK packets to the sender every time it sniffs a message from it.

- *No protection from DoS attacks:* ZigBee has no known denial of service protection mechanisms.

There are several ways to perform a DoS attack against a device:

- *Set the frame counter to its maximum value:* an attacker can send a frame to other devices setting the frame counter to the largest acceptable value and spoofing its address pretending to be another device of the network.

This last one can no longer communicate with the other devices as it is impossible to create a valid frame counter. This is the simplest DoS attack, but fortunately it is possible only in those situations where there is no integrity check on the frame.

- *Flooding:* it consists in simply sending tons of messages towards victim devices. For instance, an attacker could replay all the messages it is able to capture (or create legit messages, if it already knows the network key).

This kind of attack where the replay messages are sent to all the network devices is called *broadcast storm attack*.

- *Physical Jamming:* this is a physical attack which involves tools able to cause signal interferences big enough to prevent communication among devices.

- *No key re-using control:* a device is allowed to join the network using the same link key several times (even forever if no explicit constraint is specified), but this is a poor choice because if an attacker is good enough to steal the link key of the device, then it will be able to spoof the authentic device credentials and join the network pretending to be it.

- *Trap networks:* since a ZigBee device tries to connect directly to the first available network (or probably the one with the strongest signal) without asking anything to the user, then an attacker could create a fake network which would work as a kind of trap for the device.

- *Statistical attacks*: an attacker can obtain several information about the functioning of a ZigBee network if it analyzes its traffic properly using statistical algorithms.

6.6 Considerations and best practices

We conclude this second and last part about ZigBee by listing some considerations:

- the protocol itself is not much secure, regardless of its implementation
- the protocol works well if all the devices within the considered network can protect themselves from keys theft
- always enable frame encryption to protect frame content, but especially to guarantee their integrity
- it would be better to always use the commercial mode, which is more secure than the standard one (even if not much secure anyway)
- avoid using manufacturer link keys, and remember that they are often public
- prefer unique keys rather than global ones every time you can
- it is recommended to test your network with a dedicated security tool, like *Killerbee*², before deploying it (especially in productive scenarios)
- you can specify the list of authorized devices, using the Trust Center's Access Control List, and specify the devices with which each device can communicate using its own ACL.
- it is not recommended to use ZigBee in places covered by many wireless networks because interferences are difficult to remove and communication's quality will be poor.
- select the frequency band that generates the least interference with other wireless systems.

The considerations above should help you to understand how to build an almost secure and reliable ZigBee network, but I would like to clarify a last thing, which is probably even more important than the previous ones.

²<https://github.com/riverloopsec/killerbee>

ZigBee was born almost fifteen years ago when devices had much less computational power than today and they needed a very light-weight and low power-consumption protocol to be able to communicate with each other. Besides, despite the protocol was designed with security in mind, there are still too many protocol's features that inevitably lead to relevant security risks.

In addition to security problems, there are also many communication malfunctions such as possible interferences with other wireless networks, loss of key synchronization, loss of periodic key updates.

Nowadays, technological progress is surprisingly fast, and even very small devices often have enough computational power and battery-life to support *Wi-Fi* protocol, which is more secure, reliable, easier to configure and continuously improved.

I conclude this section, saying that, in my opinion, ZigBee is a not so secure protocol having many security problems, which should be considered only if treating with extremely constrained devices, otherwise choosing other more modern and efficient protocols, such as Wi-Fi, is a better choice.

Chapter 7

Conclusion

After having discussed the main theoretical security concepts of these two important IoT protocols, I would like to conclude my thesis making some considerations about their current importance and business reality usage.

In 1999 MQTT was developed as a lightweight messaging protocol to enable low-energy monitoring of an oil pipeline. Nowadays it is considered as the most efficient protocol for IoT devices and was adopted as an international standard for IoT by *OASIS*.

More and more people and companies show interest in MQTT, as confirmed by *Google Trends*, which highlights a strong and continuously growing interest related to the keyword “MQTT” in the recent years, as we can see in Figure 7.1.

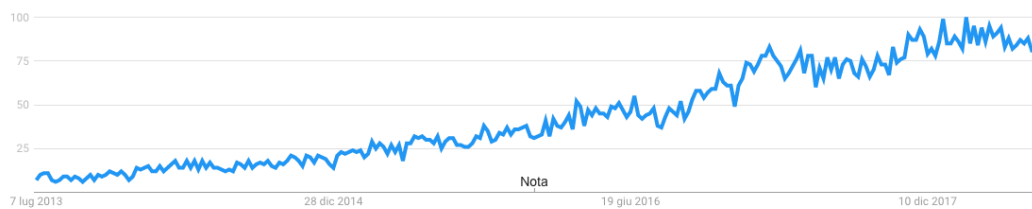


Figure 7.1: MQTT - Google Trends Statistics

Previously, we have seen in Figure 3.1 a relevant number of active MQTT servers obtained using Shodan.

Being it a rapidly growing number, I decided to draw a graph better representing the number of active MQTT servers I found day by day in the last two weeks.

Figure 7.2 shows us the obtained result.

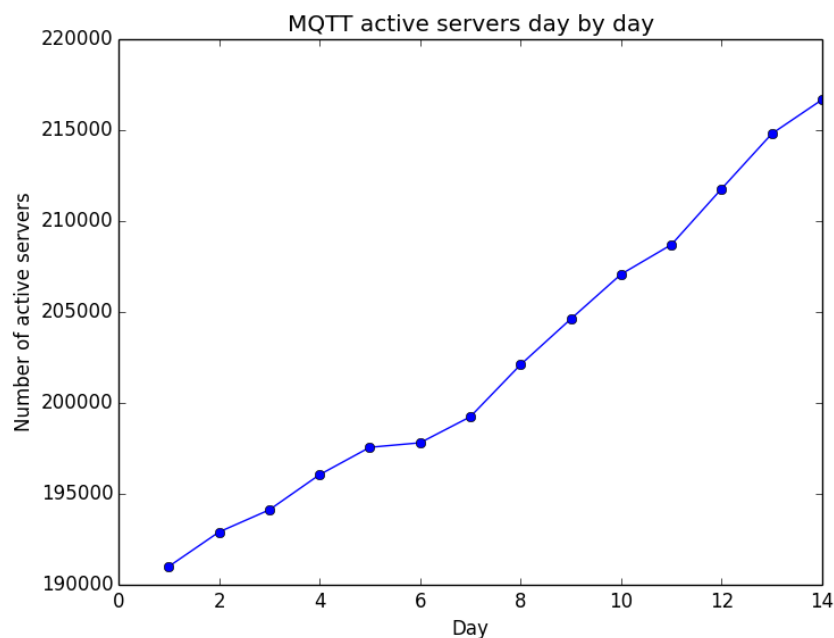


Figure 7.2: MQTT daily active servers

I would have also liked to draw a graph representing the number of insecure servers over time, but unfortunately for doing so it is necessary to pay a fairly expensive Shodan API key.

Anyway, considering what we have said so far, I guess that the graph would be very similar to this last one.

MQTT meets all of IoT's needs, in fact it is specialized in connecting low-bandwidth devices having limited CPUs, bringing a very little overhead, with a central hub. Just think about the huge number of systems involving energy monitoring sensors pushing communication to a remote monitoring location for analysis.

Due to its simplicity and efficiency, MQTT is adaptable to a wide variety of applications in different industries. Even ABO DATA uses this protocol for most of its applications.

Furthermore, making secure MQTT communication is pretty simple thanks to the TLS support, but despite this, during my internship I learned that there are so many aspects to consider before making secure and reliable applications, such as customers' needs and deadlines.

The main reason why many MQTT systems are not secure is that customers are not interested or not even aware about security issues. Besides deadlines must be respected, and what I have personally seen is that customers are much more interested to working applications rather than secure ones, hence most of times the first goal is to simply complete the task without considering security issues.

Talking about ZigBee, Figure 7.3 shows us that the interest in this protocol has almost remained constant in the recent years.

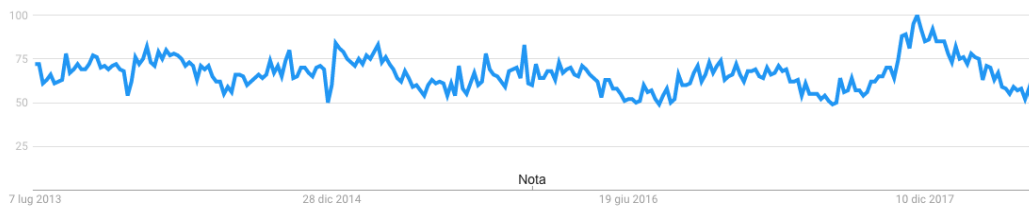


Figure 7.3: ZigBee - Google Trends Statistics

ZigBee, similar to MQTT, is primarily designed for very constrained devices, having an even lower overhead, thanks to its low-rate communication. The relevant differences between them are the standards on which they are built, the security measures (e.g. ZigBee cannot simply rely on TLS as MQTT does), the possible applications and the efficiency as well as the reliability.

Unfortunately, I have not had the opportunity to work with this protocol during my internship, so all I know is what I have read or heard from external

sources.

What I think about this protocol is that it is still not considered as important and revolutionary as MQTT, because it is mainly suited for smart home applications and domotic, which are interesting applications but currently still not much in vogue and necessary.

Besides, this protocol has many security vulnerabilities and communication problems (e.g. missing network key transports/updates, many hard-to-remove physical interferences).

Finally, since technology is growing extremely fast, even small devices are becoming more and more powerful and able to support other heavier wireless protocols, like Wi-Fi, which is definitely much more widespread.

Moreover, researchers are being to bring down power requirements of Wi-Fi based systems, so that they can perform at low energy levels like ZigBee does.

Today we almost trust the security of our Wi-Fi networks, but it has taken a few improvements over the years to get there. The same might happen with ZigBee one day: more usage will encourage product providers to improve the way security is handled and make it trusted as Wi-Fi is today [19].

Bibliography

- [1] Kevin Ashton. "<https://en.wikipedia.org/wiki/KevinAshton/>".
- [2] Maciej Kranz. Success with the internet of things requires more than chasing the cool factor. 2017.
- [3] Louis Columbus. 2017 Roundup Of Internet Of Things Forecasts. <https://www.forbes.com/sites/louiscolumbus/2017/12/10/2017-roundup-of-internet-of-things-forecasts/#5866d27d1480>, 2017.
- [4] Andrew Tannenbaum. Why do iot companies keep building devices with huge security flaws? <https://hbr.org/2017/04/why-do-iot-companies-keep-building-devices-with-huge-security-flaws>, 2017.
- [5] MQTT RFC. <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf>, 2014.
- [6] Cirrus Link. <https://www.cirrus-link.com/>.
- [7] HiveMQ Team. MQTT Essentials. <https://www.hivemq.com/blog/mqtt-essentials/>.
- [8] Shodan. <https://www.shodan.io/>.
- [9] Digicert Certificate Inspector. https://www.digicert.com/cert-inspector-vulnerabilities.htm#certificate_vulnerabilities.
- [10] Man in the middle. https://en.wikipedia.org/wiki/Man-in-the-middle_attack.

- [11] OCSP. https://en.wikipedia.org/wiki/Online_Certificate_Status_Protocol.
- [12] 01net. Proteggere la rete con il software defined perimeter. <https://www.01net.it/proteggere-rete-software-defined-perimeter/>, 2018.
- [13] Juanita Koilpillai. Is mqtt secure for the iot? only with an sdp! <http://www.waverleylabs.com/is-mqtt-secure-for-the-iot-only-with-an-sdp/>, 2016.
- [14] Sebastien Jeanquier. An analysis of port knocking and single packet authorization. <https://www.securitygeneration.com/wp-content/uploads/2010/05/An-Analysis-of-Port-Knocking-and-Single-Packet-Authorization-Sebastien-Jeanqu.pdf>, 2006.
- [15] ZigBee Stack Layers. https://www.digi.com/resources/documentation/Digidocs/90002002/Content/Reference/r_zb_stack.htm.
- [16] Xueqi Fan, Fransisca Susan, William Long, and Shangyan Li. Security analysis of zigbee, 2017.
- [17] Vishruta Rudresh. Zigbee Security: Basics, part 2. <https://research.kudelskisecurity.com/2017/11/21/zigbee-security-basics-part-2/>, 2017.
- [18] David Gasen. Security in 802.15.4 and zigbee networks. <http://www.libelium.com/security-802-15-4-zigbee/>, 2009.
- [19] Cees Links. ZigBee and the Smart Home: Is security an issue? <https://www.ecnmag.com/blog/2013/09/zigbee-and-smart-home-security-issue>, 2013.