Security analysis of protocols used in IoT solutions

Andrea Olivari

June 20, 2018

Contents

Chapter 1

Introduction to IoT

Internet of Things (IoT) is a neologism, introduced by Kevin Ashton [**?**] in 1999, which has become popular in recent years due to growing interest from companies and because nowadays it is possible to produce small devices with a reasonable computational power; 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 measures how many kilometers we have walked during the day, to smart homes that control the air conditioning system based on temperature and pollution in the air, or even smart cities where semaphores adjust their times based on the traffic condition.
But these are only three examples, IoT extends also to other fields like: health care, industrial monitoring, self driving cars and agriculture.

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

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[**?**], by assuming this forecast as real, it means that lots of devices will be deployed

in the network, but exposing a device or an host on the Internet also means exposing it to possible threats.
An interesting case of insecure IoT system is given by a luxury hotel in Austria [**?**], the electronic key system was attacked by hackers that were able to lock out (or in, it depends) guests by their rooms, until they paid a ransom; the system lacked a fundamental security component in order to avoid intrusion.
Long story short: after the attack, the hotel reverted to physical keys.
By this real case scenario we can imagine other problematic situations that does not imply an attack, for instance: if a blackout occurs, a guest is 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 aspects of our life, an electronic keys system is surely comfortable, you can configure the electronic key as you want in few seconds, on the other hand is quite easy to overwrite the content of an electronic key, while it is a bit more difficult to craft the copy of a physical key (if you are not a professional burglar), because you first need to steal it and duplicate it.
Another interesting and funny case of insecure devices is provided by CloudPets[**?**] a company that produces toys, and in particular: Internet-connected fluffy animals toys, these toys were able to connect via Bluetooth to a smartphone's app in order to download and upload audio messages; it is possible to access the voice recordings without any authentication if you known the exact URL at which they were stored.
The Guardian states that "The personal information of more than half a million people who bought Internet-connected fluffy animals has been compromised", it does not sounds like a good advertisement when the personal informations are: username, password, profile pictures and voice recordings.
Note: the passwords were cracked easily because they were extremely easy to guess ("password", "123456", "qwerty", "qwerty123456" and so on).

Another interesting thing to highlight is that "The personal information was contained in a database connected directly to the Internet, with no usernames or passwords preventing any visitor from accessing all the data", this could means that the entire application was developed without any security requirements or security knowledge.

The security aspect of an IoT application is not related only to external attack by criminals but it also extend to reliability; 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 most proper one that fits well your business logic; also take in mind 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 the constrained device with low computational power to high performance cluster machines; basically the dirty work is taken over by lots of low power devices which communicate with the server, that will elaborate this data or forward it to the cloud in order to elaborate them in a faster way.
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 developement team should be formed by 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.

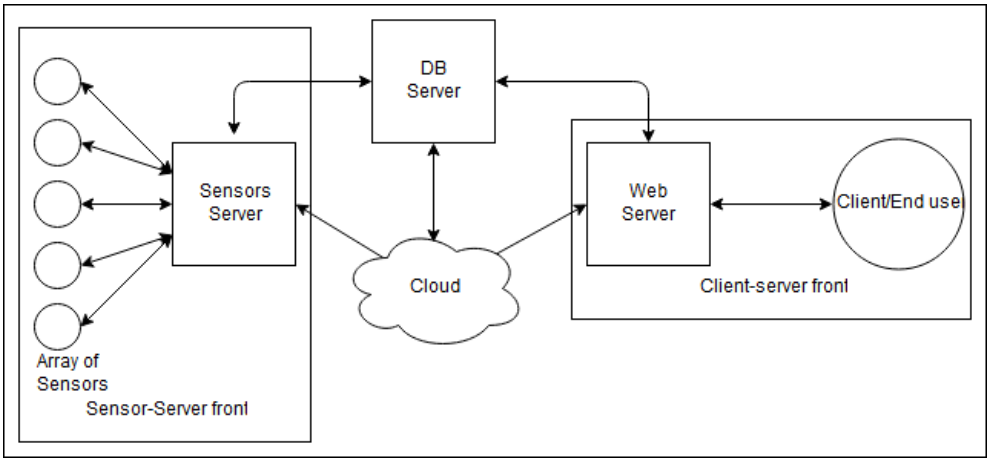

Figure 1.1: Possible structure of an IoT application

1.2 Requirements of an IoT system

In IoT systems it is common to deal with a lot of high data rate communication involving constrained devices, probably powered with batteries, having limited (sometimes extremely limited) computational power, bandwidth and RAM.
For this reason, we need some standard providing low-latency with low-energy consumption even at lower bandwidths.

More in details, these are the main requirements for an application network composed by a relevant number of constrained devices:

- be able to scale to larger sizes and operate for a long time (years, in some scenarios) without manual intervention
- long battery life for devices
- granting good quality services
- low complexity and low cost communication

1.3 Argument of the thesis

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

With regard to the MQTT protocol, I will also talk about 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 productive system should be designed.

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

Let's start to discuss them starting from the first one, MQTT.

Part I

MQTT

Chapter 2

Protocol Overview

2.1 Introduction

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

- minimal battery loss
- minimal bandwidth
- be easy simple to implement
- providing a 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 *(blood pressure, insulin, heartbeat)*
- Fitness devices *(fitbit)*

- Location services
- Home automation kits used in *smart homes*
- Inventory tracking *(systems designed to track movements of inventory)*
- Automotive telematics
- Instant messagging applications *(for instance, Facebook Messenger)*

2.2 Why is MQTT perfect for IoT?

Internet of Things requires a *real-time event-driven communication model*, besides it must be possible to send information in *one-to-many* modality. Since we mainly deal with sensors sending lots of data continuously, we need to reduce as much as possible the dimension of data packets to be sent, however in a reliable way.

HTTP protocol is not suitable for this purpose because it is based on the request/reply paradigm and has a too 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.

For all these reasons, *Google Trends* highlights a strong and growing interest related to this protocol.

2.3 Architecture and basic concepts

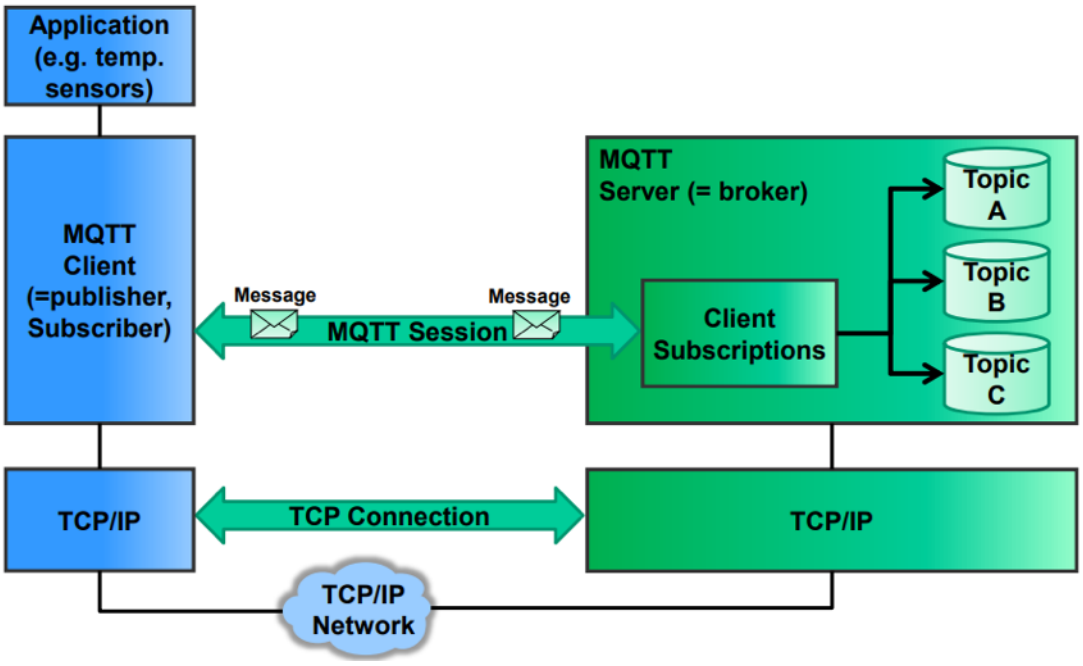

Figure 2.1: MQTT Architecture

2.3.1 Publish/Subscribe pattern

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 publishing and subscribing to topics; we are going to discuss them in the section 2.3.2. The sender of a message is called *publisher*, while who is going to receive 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.

2.3.2 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 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 who will receive messages, in fact if a publisher `P` publishes a message on the first topic, and a subscriber `S` subscribes to the second, this last one will not receive anything because it is not subscribed, and consequently not interested, in messages regarding the light sensors placed in the kitchen.

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 receiver*, 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

We will see in figure 2.3, 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.

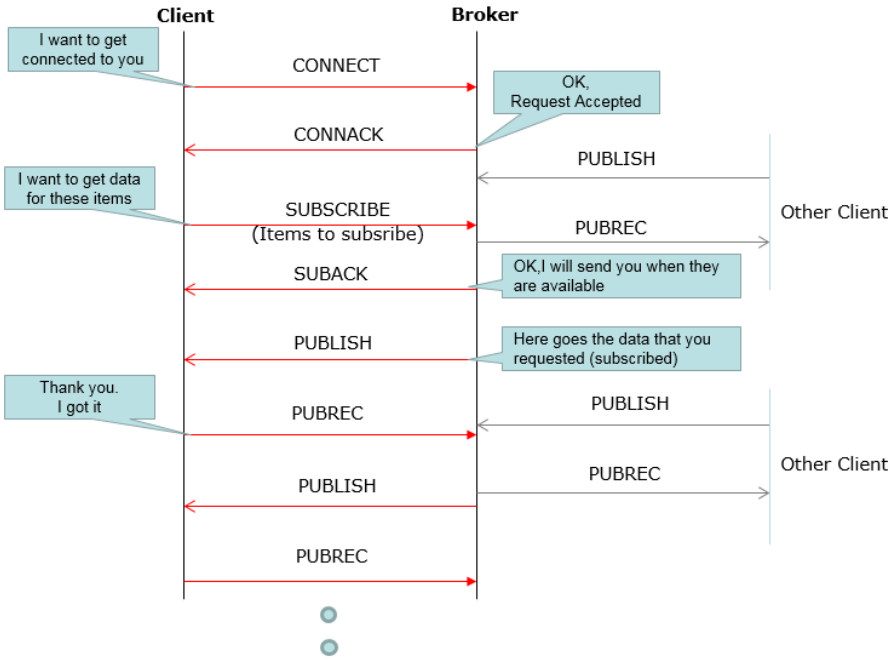

Figure 2.3: Publish/Subscribe combined with Topics

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

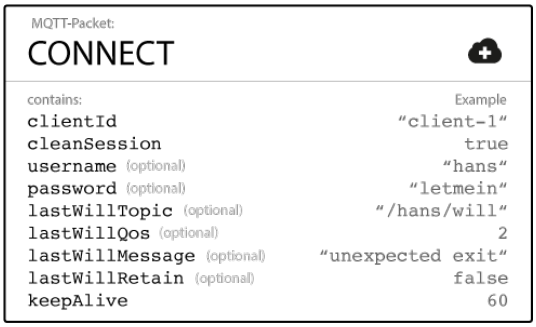

Figure 2.4: CONNECT packet

where:

- *clientId* is a identifier, unique per broker, each client must have.

If you do not need a state to be hold 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 plaintext, 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 (this last one 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.

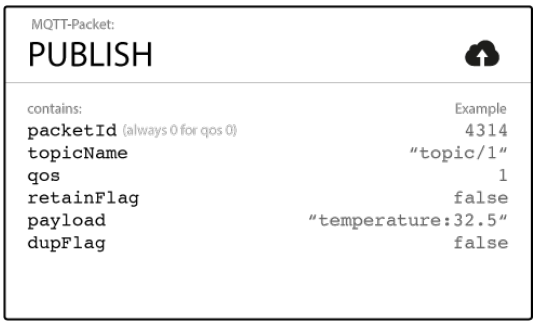

Figure 2.5: PUBLISH packet

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.

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 2.3.3 we discuss aspects regarding the quality of the service.

2.3.3 Quality of Service Levels (QoS)

We have briefly introduced the structure of the `PUBLISH` and `SUBSCRIBE` packets in section 2.3.1; 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 exist 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 receiving duplicated messages.

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

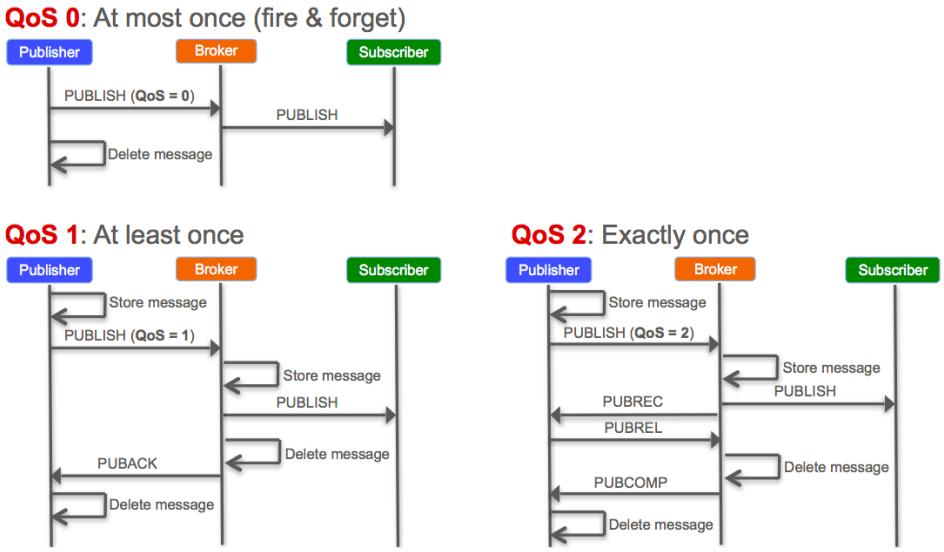

Figure 2.6: MQTT QoS levels

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 think to 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.

- in this last scenario, we assume that for some reason there is a diabetic person having an auto-injecting insuline device applied to his body, which expects to receive an MQTT message when an extra injection is needed.
  Obviously, in this case it is necessary to receive one and only one message, so we must choose QoS2.

2.3.4 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.

2.3.5 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's 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 the client.

2.3.6 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.

2.3.7 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 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 this real-world example in order to understand how much this protocol is used, and how many times its security is completely ignored.

In this example we are going to use a search engine, called *Shodan* [**?**].
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 its filters, searching for *"port:1883"*.

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

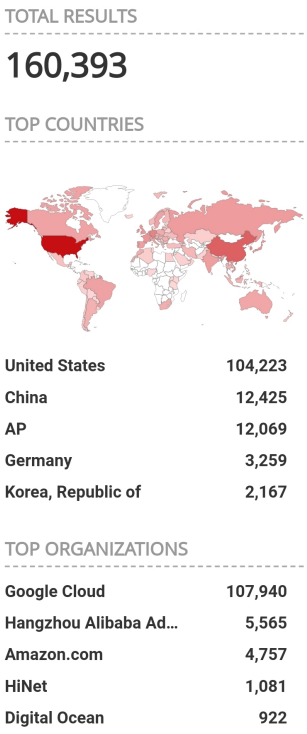

Figure 3.1: MQTT brokers found using Shodan

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, the more interesting thing is that such a number is growing up very quickly: just to realize, two weeks before the same research gave me 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 lead 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 security[1].
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 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.

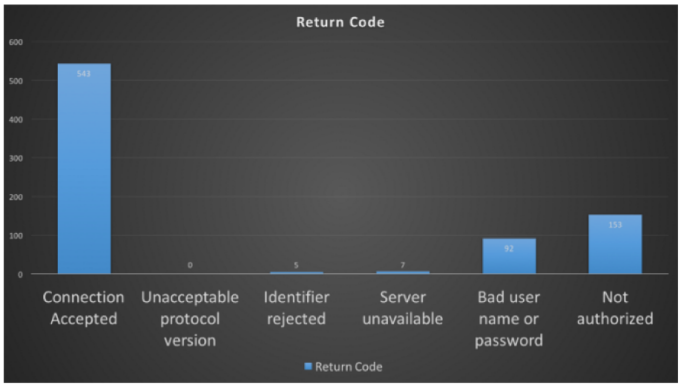

Figure 3.2: MQTT brokers return code stastistics

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

Now that we have realized how much MQTT security is neglected in real-world applications, let's start talking in details about this fundamental aspect.

[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 handhsake 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 [**?**]

- Always use certificates from trusted Certification Authorities. Using self-signed certificates or allow-all approach is not recommended

- Including additional security mechanisms, like payload encryption and authorization mechanisms, in addition to TLS is a good choice.

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 2.3.5 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 [**?**] would be enough to sniff them without effort, as we can see in figure 3.3.

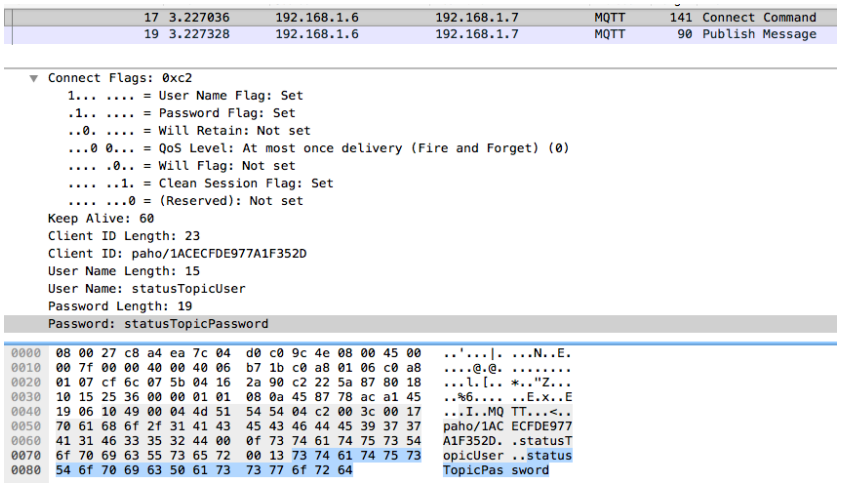

Figure 3.3: MQTT credentials sent in plain text

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 encrytped 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.
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 a own local CRL to consult, but clearly it has to remain updated getting the updated list 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 in details 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 the certificate; you simply ask your certification authority, which is going to do the boring job for you.
     Let's see a simple example [**?**] 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 to Bob her certificate
     * 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*).
We can better understand this concept with an example related to web applications:

Talking about Facebook, let's suppose to have a standard user and a developer. When we visit Facebook, we are asked to insert username and password to let the application identify us and this is the authentication part, equal for everyone. Then, the authorization process is different for the two users, in fact the standard user will have a more restricted access to resources, compared to the developer.

Coming back to MQTT, let's try to understand which constraints can be applied to clients.
A client, once connected to a broker, can publish messages and subscribe to topics, nothing more.
We may notice that 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 proper return code which tells the client that its subscription was denied, nothing more.

In section 3.4 we discuss another form of authentication/authorization mechanism usually implemented in other protocols, like HTTP, and we will see wether and how it is possible to combine it with MQTT.

3.4 MQTT combined with OAuth 2.0

OAuth [**?**] is a standard for access delegation, used to allows a third party to access a resource owned by a resource owner without giving unencrypted credentials to the third party.

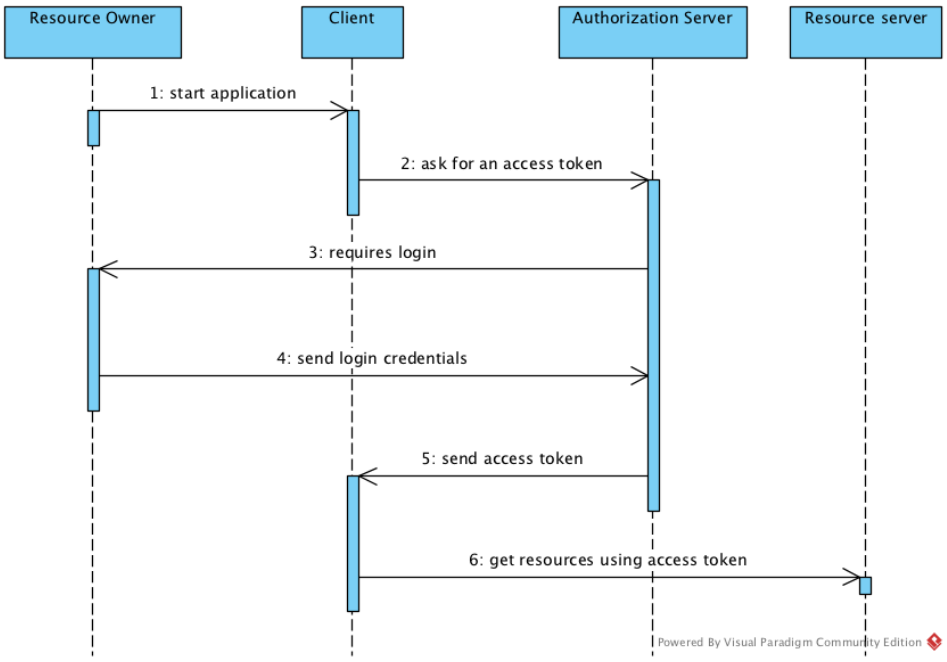

Figure 3.4: OAuth working scheme

To make it clear, we may think to Spotify which allows its users to login with their Facebook account, then once they have authenticated, it gets their information (resources), like name, surname and maybe something else, from Facebook resource servers.
More in details, referring to Figure 3.4, we have that Spotify is the client, Authorization server is the Facebook Authorization server, in charge to control the access to all the resources, and the Resource server is the Facebook Resource server where there are contained the information Spotify is going to take.
The resource owner (the user) will login on Facebook communicating directly with its Authorization Server, without giving any credential to the client. Once Facebook authenticated the resource owner, it sends the access token to the client.
So, the idea is pretty simple: we do not want to save our credentials within the client and use them every time to authenticate us because it could be dangerous, therefore the solution is using access tokens.
Clearly, if we do not use encrypted communications everything is pointless because an attacker could easily sniff our access tokens and use it to access

our own resources.

What is the token's structure?

Usually, OAuth uses *JSON Web Tokens (JWT)*, JSON Objects with base64 encoding. They contain an header, a payload and a signature, where:

- The header contains information about the algorithm used to generate the signature
- The payload contains information about the token, like expiration date, scope, issuer, issue date, etc.
- The signature is used to certify that the token has been generated by the trusted authorization server

How can we exploit this standard in MQTT?

First of all, we have to clarify that OAuth is mainly designed for usages with HTTP, so over any other protocol is out of scope, but we can analyze the possible MQTT workarounds.
In the previous example the login to the authorization server expects a human action, but IoT devices are often standalone. Therefore, no human can help them in this OAuth authentication process; they have to be both the client and the resource owner, but this is against the OAuth principles since it was born to avoid clients to hold their credentials locally.

Anyway, since IoT devices are standalone, this is the only way to use OAuth: storing the credentials within the devices, use them to get a valid access token from an Authorization Server, and finally use it to communicate with the broker.
Obviously, the expiration time of the access token must not be too small, otherwise the client should ask for the token every time it wants to communicate with the broker, and this would be equivalent to sending the username and password to the broker at the beginning of each communication.

Probably, the real pros of using tokens in MQTT is that they can be used to get client's authorizations quickly, saving search time to the server: if no token is used, the broker will have to look up for users' authorizations in a

database (after their authentication); a faster and more convenient way is to specify the permissions granted to the client within the token (for instance, in the payload).
This way is secure, because, as already said, the broker can verify the authenticity of the token by checking its signature.
The access token will be used in `CONNECT` packets, and optionally in `PUBLISH` and `SUBSCRIBE` ones. Before making this last decision, since there are tons of pub/sub messages, it is important to consider how much token validation afflicts performances, and, if necessary, think to some caching method.

Conclusion: OAuth is a valid authentication mechanism, but it is difficult to combine with MQTT; besides, before deciding whether to use it or not, it is fundamental to evaluate if the few obtained advantages are worth the effort of implementation.

3.5 Simple Data Encryption

Since, as we have already said, some devices have too low power to use TLS. In this section we discuss how those constraint devices may encrypt only specific data, like the payload in the `PUBLISH` message.

Using this strategy sensitive data, let's 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 3.5 explains this scenario, called *End-to-End*, perfectly.

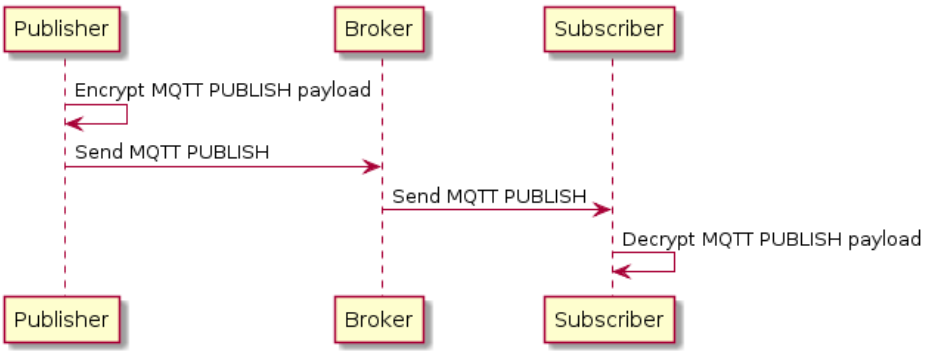

Figure 3.5: MQTT communication using End-to-End encryption

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 3.6.

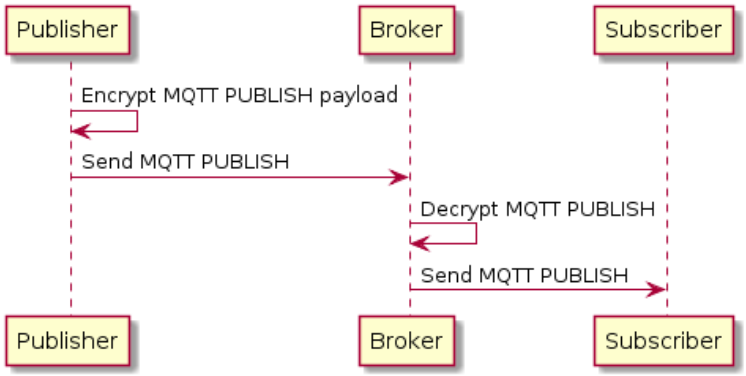

Figure 3.6: MQTT communication using Client-to-Broker encryption

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 cannot 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 specific data only is a method to add a security layer to communication, and can be an option for constrained devices which cannot rely on TLS.

- even if TLS is heavier, encrypting and decrypting every message can be too 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 steal our sensitive information by sniffing, 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 provisioning of them.

In section 3.6 we will see another fundamental security aspect: data integrity.

3.6 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 receiveing 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 cryptographic Hash function (usually *HMAC*) combined with a cryptographic key. Since only trusted clients know the secret key, we can discover if a message come 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 certainity 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.

Integrity checks is always a good choice: 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.

In section 3.7, we see how to make a MQTT system even more secure by using a not very common, but very powerful, security architecture called *Software Defined Perimeter*, or simply *SDP*.

3.7 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.

3.7.1 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. [**?**]

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

3.7.2 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 3.7 shows an example of a possible SDP structure.

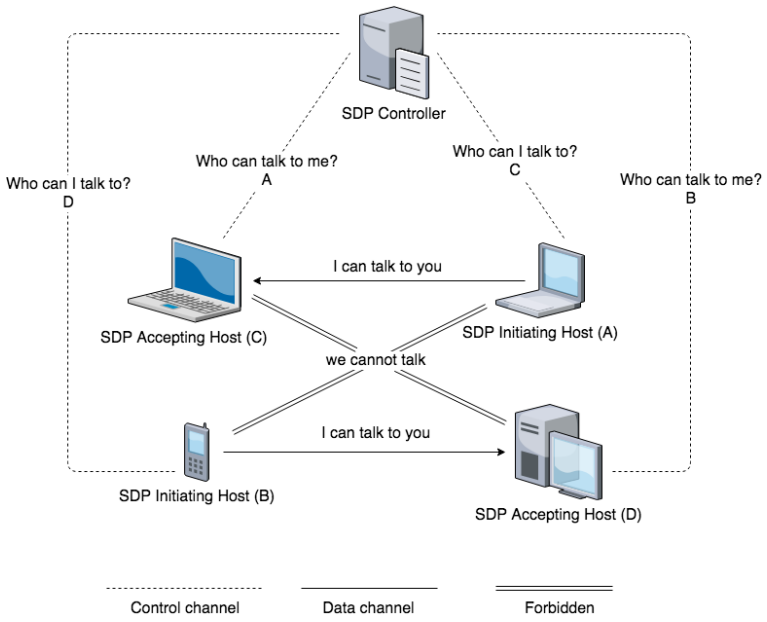

Figure 3.7: Example of SDP structure

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 infomation about encryption are sent to both Accepting and Initiating hosts.

5. Now Initiating Hosts can communicate with Accepting Hosts.

3.7.3 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 can be replaced using the so-called *Single Packet Authorization (SPA)* [**?**].

What is a Single Packet Authorization? [**?**]

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 unaccessible 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 doors; then the firewall updates its own rules in order to allow the sending host to connet to the desired port.
This old system is basically flawed and vulnerable to replay attacks, as we can see in Figure 3.8, in fact an attacker can sniff the attempts and replay them, obtaining the access to the service's port.

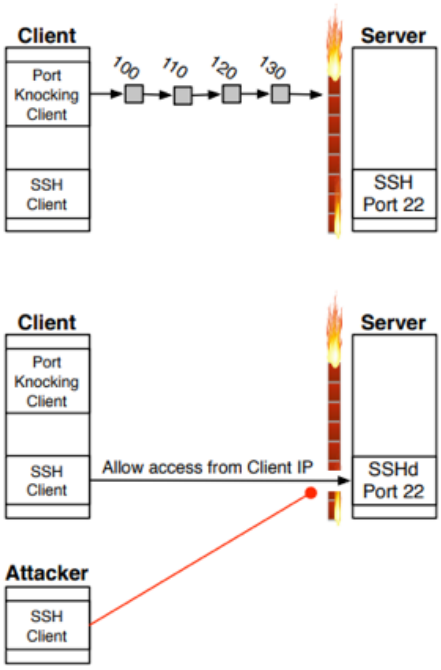

Figure 3.8: Port Knocking Replay attack

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 exists a way to share it with authorized clients
- encoding the packet using MD5 makes almost impossible for attackers to steal the secret password

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.

Important: to make it 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.

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.

Let's analyze some possible alternative solutions to TLS to face this problem, but keeping in mind that they are simple workarounds, and *they should be considered only if TLS is absolutely impossible to use.*

- since authorized users already know the secret password, using it as cipher key might be an idea. Using the AP password also as cipher key might be dangerous because if an attacker is able, somehow, to discover it thanks to some intercepted messages, then it will also get access granted to the server.
  On the countrary, if an attacker can crack the MD5 message used for AP authentication, he will be able to get access to the server and send messages to the broker.
  If we assume that the server rejects invalid IP addresses, the attacker will have also to properly spoof its IP.
  Cracking the password of some intercepted messages is not easy at all, but it is better to be as careful as possible.

- an alternative way might be sending a cipher key to the user once he has successfully authenticated to the server, encrypting the message and generating the MAC using the password as a key.
  In order to guarantee more security, it would be better to generate a different cipher key for each client ip instead of a shared one. Why? Let's assume that an attacker wants to sniff some messages with the hope of cracking their cipher key: if every client uses the same cipher key the attacker will have a much bigger number of samples to work with.
  Besides, in the event that all the clients share the same key, anyone (a valid client or an attacker that have stolen it) could pretend to be another valid client using spoofing techniques, but in this case, since each client has its own cipher key, this is not possible.

The bad aspect of this solution is that the server would have to keep a, potentially huge, list of client IPs and assigned cipher keys, continuously updated.

As already said, any of these strategies is secure as employing TLS.

In section 3.8, we wrap up the best practices to build a secure MQTT system we have seen so far.

3.8 Best practices

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

- hiding devices and limiting 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

- 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).

- force the client to a further authentication mechanism, that is using credentials to login to a broker. This step is optional, since the client has already authenticated on the server sending the AP, and it seems to be a bit redundant but it might be an additional option.

- encrypting communication using TLS.
Standard TLS is already secure enough, but 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.
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.
Then, even the server will be sure that it is talking to a specific, as well as trusted, client.

- thinking 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.

- using tools to detect brute force/dictionary attacks and lock out attackers.

- choose the proper QoS level according to your needs.

In the next section, we will try to apply in practice the theoritical concepts seen so far, developing an homemade secure MQTT system.

3.9 Homemade secure MQTT system

This example can be seen as a kind of pratical 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 are able to 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 attacks 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.

3.9.1 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 -P INPUT DROP
3  iptables -P OUTPUT DROP
4  file="trusted_client_ips.lst"
5
6  while IFS= read line
7  do
8      iptables -A INPUT -s $line -p tcp --dport 62015 -j
           ACCEPT
9      iptables -t filter -A OUTPUT -d $line -p tcp -j
           ACCEPT
10
11 done < "$file"
12
13 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.
Note: an attacker might try to perform brute force or dictionary attacks against this daemon, so we will explain later how to easily protect our system from them.

3.9.2 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 is 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"
                                    %(client_ip,service_port)
8
9    cmd2='echo "iptables -D INPUT -s %s -p tcp --dport %s
10         -j ACCEPT|at now+2 minute;' %(client_ip,service_port)
11
12   os.system(cmd1)
13   os.system(cmd2)
14
15
16 def checkAP(received_md5,client_ip):
17   timestamp = str(time.time())
18   timestamp = timestamp[0:timestamp.find('.')] # avoid ms
19   str_to_encode="%s:%s:%s" %(timestamp,client_ip,SPA_passwd)
20   expected_md5=hashlib.md5(str_to_encode).hexdigest()
21
22   if(expected_md5==received_md5):
23     grantAccessToClient(client_ip)
24   else:
25     log_file.write("[%s] %s - Invalid MD5 received\n"
                                        %(timestamp,client_ip))
26     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 him 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 3.9 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.

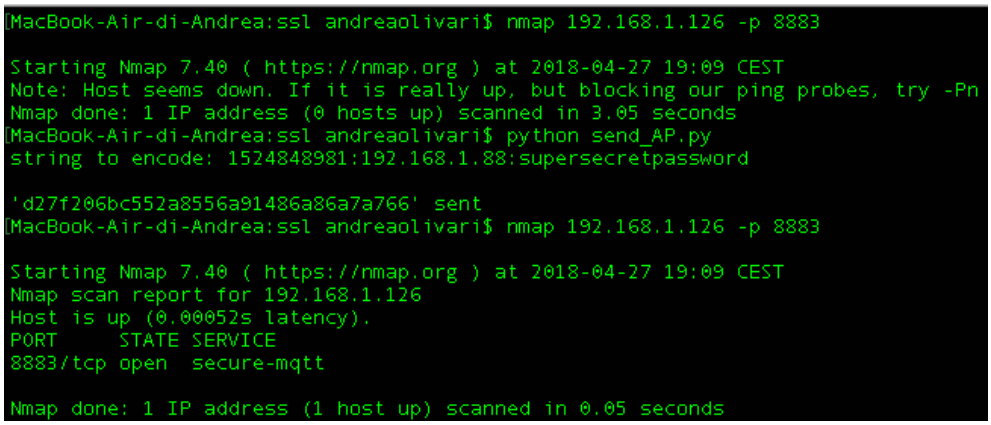

Figure 3.9: Nmap scan before and after sending a valid AP

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,
       password,callback) {
18     callback(null,(username===broker_username &&
         password.toString('ascii')===broker_password))
19   }
20 });
```

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,
                                                retain=
                                    retain_msg)
3
4 client = mqtt.Client()
5 ssl_version = ssl.PROTOCOL_TLSv1_2
6 client.username_pw_set(brk_username,password=brk_password)
7
8 client.tls_set("broker_cert.pem",cert_reqs=ssl.CERT_REQUIRED,
                                    tls_version=ssl_version)
9 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 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 in the company I often had to deal with tasks related to receiving data from a source and their subsequent forwarding to a destination, like a bridge, and most of times I worked with two protocols: HTTP(s) and MQTT.

In the next page, I would like to discuss two real-world cases I have worked on, which should make us reasoning about how a productive 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 realiable monitoring and management system for their machines using a Cloud architecture.
This proposal's goal is 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
- create a so-called *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.

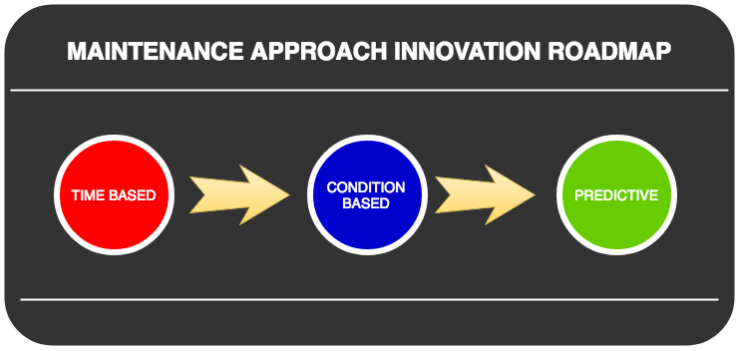

Figure 4.1: Maintenance approach innovation roadmap

A *Time-based* maintenance approach is the simplest one, and unfortunately the most used, so when a machine breaks down, the customer contacts the company support 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

- the customer uses data from its 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

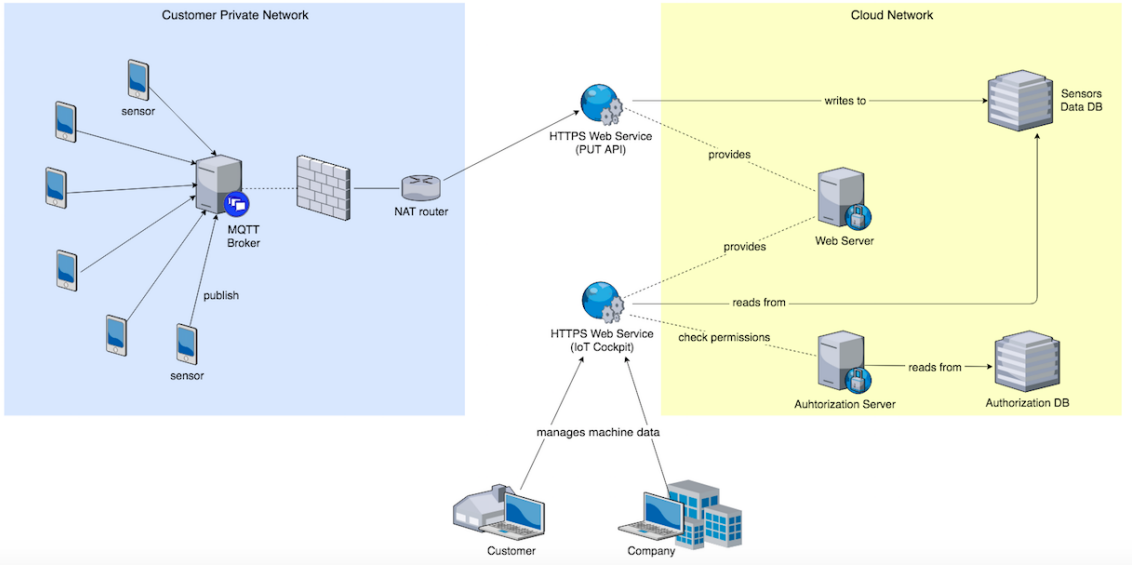

Figure 4.2: Factory Architecture

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 4.2, the customer runs a MQTT broker on a local server to which the company's sensors publish their data.

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, where these last ones publish their 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 defintely helps, since an attacker must physically access the building to connect its malicious device, but of course this is not always possible, especially when devices belonging to the same network are not located in the same building.
In that case it is necessary to create a secure wireless network protected by strong passwords.

As shown above, 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 data collected by the sensors to the cloud.
Before going on, I would like to clarify that putting a firewall in front of the server is a good practice and gives the network a robust protection, but in this case, having a *NAT* router which does not any port forwarding and no mapping between the public IP address and the internal ones, using it seems to be a little redundant.

So, we were saying that there is one outgoing connection allowed: this is nothing but the request sent by the local server to transmit sensors' data to the cloud.
To do that, it exploits 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 required, so when a customer's broker sends sensors' data to the API it will send even 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 will discuss more in detail about this at the end of Section 4.3, related to the Connected Machines architecture.
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 requires a user authentication and can be accessed from any network.

4.1.2 Connected Machines

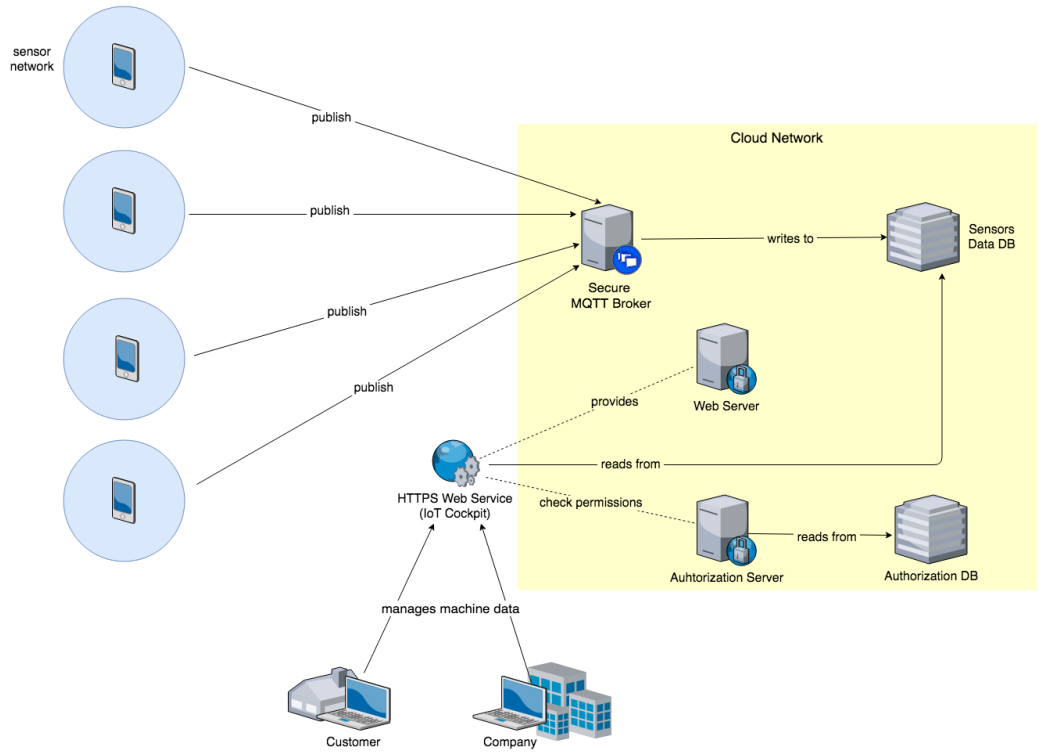

Figure 4.3: Connected Machines Architecture

This second system architecture deals with a different and more difficult to protect scenario.
In this case, there are several customers' machines positioned in different places, probably far from each other, therefore no common and secure network protecting them, but simply standalone machines and their related sensors.
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 publishes 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, so mutual authentication is required again.

Now that we have seen that mutual authentication is used in both the architectures, I would like to briefly discuss this implementative choice: previously, I told you that using mutual authentication could be replaced by authentication credentials sent along with POST data in order to spare computational time; in this last architecture it would be necessary to create credentials for each individual sensor, but we know that the number of sensors can be huge and this can be a little inconvenient.
Unless you use access tokens, using login credentials and sending them at every request would be extremely heavy for the server that should verify them, browsing a very large database continuously; this would result in a perceptible slowdown in the application layer and a large memory consumption.
To achieve better performances, someone may think to use the same credentials for multiple sensors, or even all of them, in order to avoid having to deal with a huge number of records, but this would be a risky choice because if the credentials of a sensor are stolen, then many will pay the consequences.
Using client certificates, this problem does not exist, in fact if the private key of a sensor's certificate is stolen, only that specific sensor will be in trouble.
As we have already seen, another benefit of using client certificates is that authentication is moved from the application layer to the transport one.

Here we have a practical example of what we discussed before in the section dedicated to the construction of an homemade MQTT system: 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 point-

less; 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 single perfect solution covering all cases, but it is necessary to find the most proper one for the specific case 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 remain updated 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, for obvious reasons of privacy again, I will not be able to give a name to the customer I am going to talk about.

I worked on a project where 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 emerging protocol deployed by ZigBee Alliance, for controlling and monitoring applications, covering 10-100 meters within the range.
This communication system is less expensive and simpler than Bluetooth and Wi-Fi and its range can be extended using routers, as we will see in a minute.

Nowadays, ZigBee is mainly used in the following fields of application:

- *Industrial automation*: industries exploit ZigBee to manage and continuously monitor critical equipments.
  Why ZigBee and not simply Wi-Fi or Bluetooth? Because it has lower communication costs than Wi-Fi and a higher communication range than Bluetooth, hence it optimizes the control process (theoretically).

- *Smart Home*: 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, this is the most common ZigBee application.

In section 5.2 we start to analyze the architecture and the basic concepts of this protocol.

5.2 Architecture and basic concepts

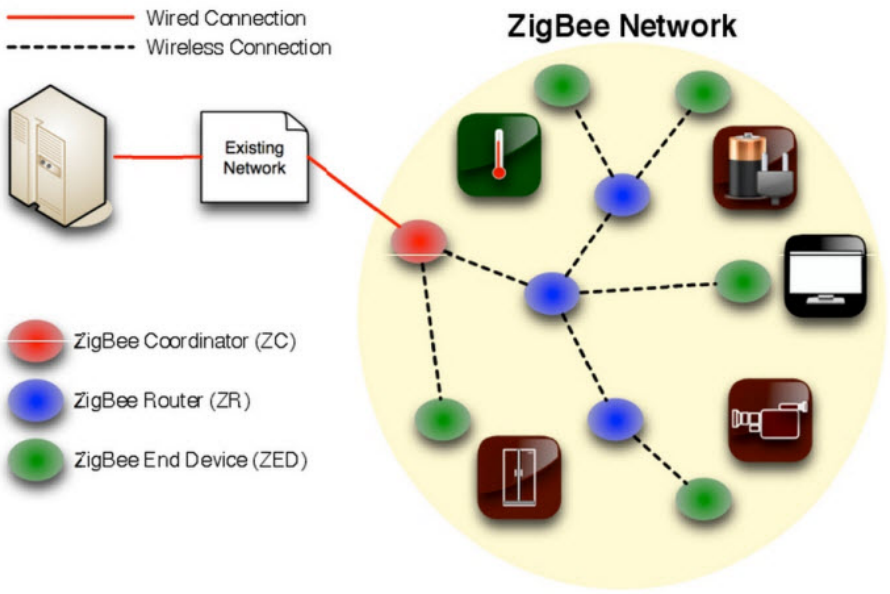

Figure 5.1: ZigBee Architecture

5.2.1 ZigBee components

As we can see in figure 5.1, the ZigBee system consists of three types of devices:

- *Coordinators*: they manage the overall network and they are responsible for starting it, allowing other devices to join it, keeping trace 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 intermiediary 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.

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*).

5.2.2 ZigBee network topologies

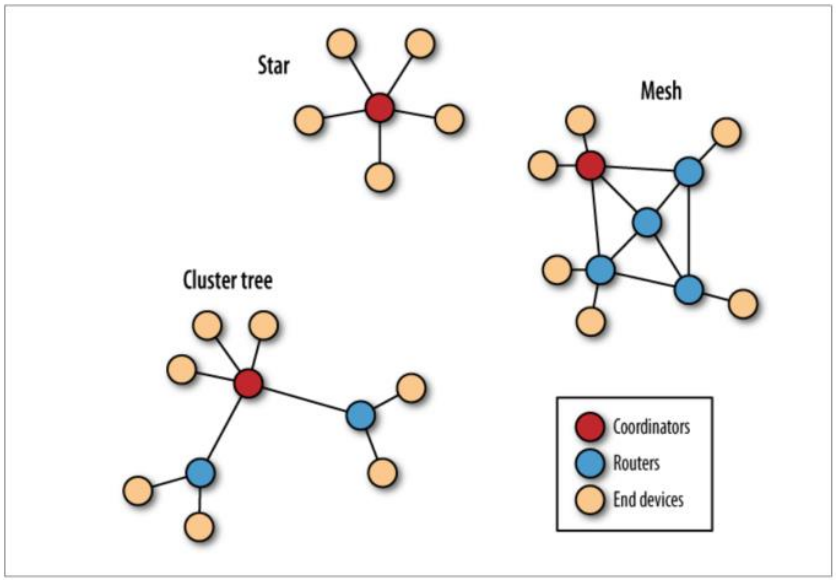

Figure 5.2: ZigBee Topologies

- *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 likely 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 setup than the previous ones.

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

5.2.3 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.

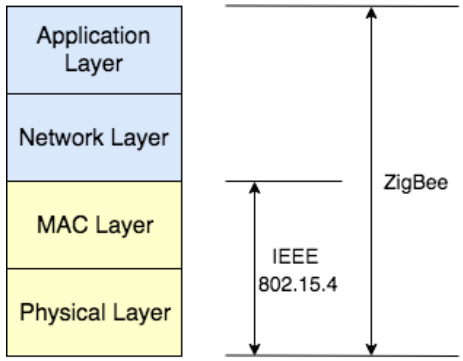

Figure 5.3: ZigBee Protocol Stack

Let's briefly see them:

- *Physical Layer*: it provides the basic radio communication capabilities, such as power, channels, modulation, demodulation, transmission rate, etc.

- *Medium Access Control layer (MAC)*: it manages data transmissions between neighboring devices (point to point). It includes services like transmission retry and acknowledgment management [**?**].

- *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.

5.2.4 ZigBee communication modes

ZigBee employs two communication ways:

- *Beacon* mode: the coordinator sends a beacon message periodically to the devices and these last ones wait for it before sending messages addressed to the coordinator.
When the transmission is completed, the coordinator set a schedule for the next beacon so that the device can go to sleep till that moment.
Beacon intervals can be from 15 ms up to 252 seconds.
So, in this mode, all the devices know when to communicate with the coordinator.
Beacon mode is mainly used when the coordinator runs on batteries, hence it needs to save as much power as possible.

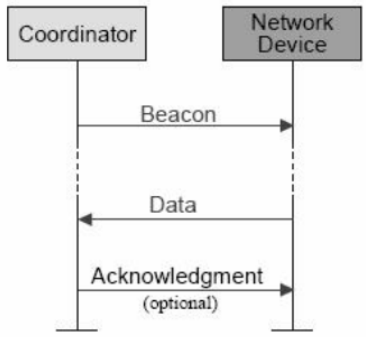

Figure 5.4: Beacon mode

- *Non-Beacon* mode: in this case the coordinator is mains-powered and wait continuously for data from the devices, which can transmit at random intervals.
This mode is used in systems where devices are almost always 'asleep' (for instance, in smoke detector systems).
Clearly, it might happen that an end device finds the channel busy and in that case the coordinator will miss the call.

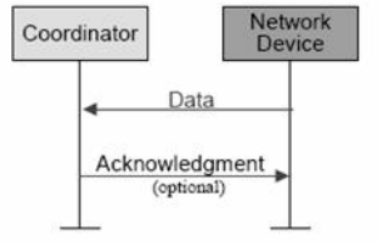

Figure 5.5: Non-Beacon mode

In chapter 6 we start to discuss the security aspects of this protocol.

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.

Let's start discussing some useful security assumptions in section 6.2.

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 are going to 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 [**?**] on which they are based:

- communication between different stack layers on the same device is not encrypted
- cryptographic keys are securely stored in quite-resistant 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: *distributed* and *centralized*.

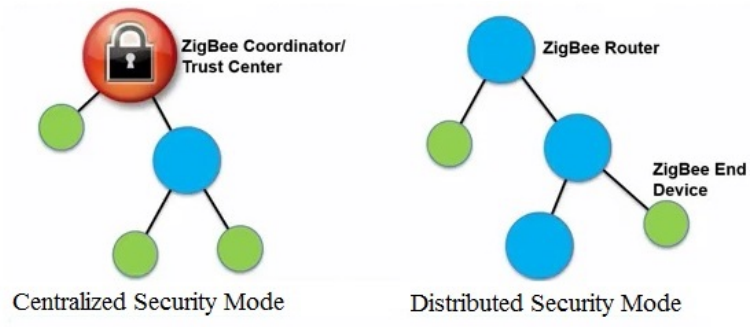

Figure 6.1: ZigBee Security Models

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

We will talk better about this in section 6.4, but just to understand what we are going to say we should know that each network has a key used to encrypt broadcast messages, called *network key*, and a key shared by 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 every-

one 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 necessarily the network key to encrypt messages.
The TC will also create and provide a new network key periodically, in order to protect the network from possible attackers, who can try to steal it.

The first model is certainly simpler but even less secure, and almost not used, so we will focus on this second one.

6.4 Security keys types and management

We have already mentioned that a ZigBee network exploits two main types of keys to communicate securely: *network keys* and *link keys*.
Actually, there is also a third key, called *master key*.
Let's see them in details:

- the *network key* is a 128-bit key generated and distributed by the Trust Center 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.

Simply, a high-security network key is sent encrypted, the other one in plain-text.
More specifically, the standard network key is sent in plain-text 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 [**?**]:

- *Pre-configured Global Link key*: it exists only for one reason, 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 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 the *Matyas-Meyer-Oseas* 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.

• the *master key*: This is a shared key used, during the execution of a symmetric-key key establishment protocol (*SKKE*), to generate link keys [**?**].
Without going into too much detail, if a initiator device and a responder device share the master key, they can use it to establish a link key following these three basic steps:

1. the exchange of ephemeral data
2. the use of this ephemeral data to derive the link key
3. the confirmation that this link key was correctly computed

This key provisioning way is almost never used, so we will focus only on the two previous ones.

Now that we have analyzed the ZigBee security keys and their provisioning ways, 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)* [**?**], 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, but we will not delve into them as they go beyond the context.
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 Trust Center, as usual, which will generate a random Application Link Key LK_{ij} and send it to both the devices using their respective Trust Center Link Key.

6.4.1 Over-The-Air Updates

We have already mentioned that in a centralized architecture the TC periodically updates and distributes the network key in order to not give attackers time to try to steal it.
Clearly, the updates of the network key are sent over-the-air, so they must be encrypted with some keys, which usually are either the current network key or the Trust Center Link key (unique or global, depending on the scenario). Obviously, the second one is a better and more secure choice, because if an attacker has already stolen the current network key, and the Trust Center

[1] https://www.ecmqv.com/

uses such key to encrypt the new one, then the attacker will always be able to get the updated network key.

Keys are not the only things we may think to update over the air, in fact ZigBee gives the user the chance to update a device, adding new features as well as applying security patches, if necessary.
Clearly, if this kind of communication is not protected enough, it might happen that devices accept some critical fake updates from untrusted sources.
Assuming, for instance, we want to send an updated code image to a device to apply a security patch, we firstly sign the image with a private key k_1, then we send the image over the network using a second cipher key k_2; doing this only the end device will be able to decrypt it and verify the validity of the image thanks to its signature (it is clear that the end device must know the two keys used to encrypt and sign).
Once the device received the message, it decrypts and validates it, then updates itself.

6.5 Protocol Stack Security

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

6.5.1 802.15.4 Security

Security Control Header

Let's start taking a look at figure 6.2, representing the structure of a MAC frame [**?**].

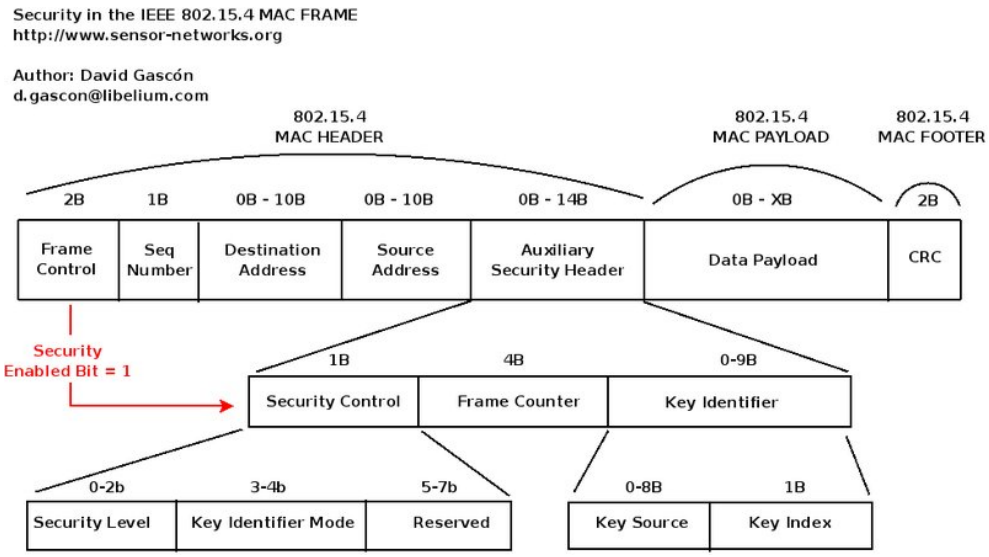

Figure 6.2: ZigBee MAC Frame's Structure

As we can see in figure 6.2, 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 6.5.2.

- *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.

6.5.2 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.

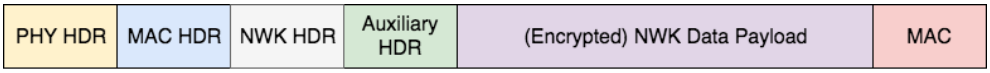

Figure 6.3: ZigBee NWK Frame's Structure

Figure 6.4: ZigBee APS Frame's Structure

CCM mode combines the *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.
In other words, 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.
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-text, 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 actually it 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.

In section 6.6 we will discuss the most relevant ZigBee vulnerabilities.

6.6 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 respected. If an attacker is able to extract the network key from a network device, then all the communications within the network will be at risk.

- *Key transportation*: we have seen before that if the Trust Center operates in residential mode, it could happen that the network key is sent in plain-text; 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 crafts 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).

- *Predictable sensor polling rates*: as we know, end devices can put themselves in sleep mode and wake up at regular intervals to poll the coordinator for data inputs or, in beacon mode, when it receives a beacon from the coordinator, in order to save their battery-life.
This mechanism gives the attacker an advantage: it can send crafted messages to the device at regular intervals even if it is not necessary, forcing it to wake up and process the received messages.
Switching from sleep mode to active mode requires a quite big amount of energy, as well as processing invalid messages, as we have just seen.

So, this is nothing but an extension of the previous Ghost Attack, which has been proved to be extremely effective.

• *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 of public domain, so available to a possible attacker.

• *Association Flooding*: by default, there is no integrity checks on acknowledgment packets, therefore an attacker is able to send fake ACK 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 impossibile 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 the simple sending of tons of messages towards victim devices. For instance, an attacker could replay all the messages it is able to craft (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.

6.7 Considerations and best practices

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

- the protocol itself is not much secure, regardless of its implementation
- the protocol works well if all the devices within the considered network are able to protect themselves from keys theft
- choose carefully the physical devices based on the role they will have within the network
- 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 often they are 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*, before deploying it (especially in productive scenarios)
- you can specify the list of devices which are authorized to access the network, 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.
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

During my internship, I learned that there are so many aspects to consider to make secure and reliable applications.
If you are developing an homemade application, you probably have no deadline, no customers to satisfy.
On the other hand, if you are developing a business application for a customer you must take care of:

- Relationship with him
- Deadlines
- Support for the application after the release

In order to maintain a solid relationship with your costumer, deadlines must be respected, and here the development of the application comes to play; including security in the development of an application may seems easy at the beginning, but it is not so simple as it may seems and it may cost lots of time, and when there is a deadline the goal is to finish the job, no matter if the application is secure or not.
This is one reason why some developers simply avoid security aspect during the development of an application.

Another reason is about money: some customers are not interested or not even aware about security issues that an application could have, so they are simply not interested in paying for it, but if a security problem occurs they will surely come yelling at you.

If you think you will spend the same time making a secure version of an application and an unsecure version of it, then you should obviously follow the first way. On the countrary, if the required working time increases in a relevant way, it is fundamental to find a solution together with the customer.

Usually, when company A sells an application to company B, company A provides technical support and updates for a certain period of time, depending on the specific scenario.
If there is no time to add a feature at the first release, it could be added in the next ones.

During the software development, the development team adopts specific software development strategies; ABO DATA adopts an agile software development strategy.
This is the right choice in scenarios where application's requirements change frequently, but from the security point of view this may lead to flaws in the code.

The security part on an application should be defined in way that allows the development team to:

- edit it as least as possible
- update it easily

Just to make an example: let's suppose that the hashing algorithm employed in our application becomes insecure for some reasons, so we must update our application, but updates are expensive; if our code is monolithic it will be hard, otherwise if the code is modular it will be easy.

Another thing to keep in mind is that in the IoT world we often deal with constrained devices; if, for some reason, we need to change an already employed algorithm with a newer one, we should check if our constrained devices are capable of handling it. Sometimes it is necessary to stick with the old algorithm, or introduce more powerful devices.
This is another important to take care about: *find a proper trade-off*.

Implementing a security feature or fixing security flaws in software already in an advanced development stage is difficult: it may require substantial

changes to the whole project, making difficult to meet the deadline.
Security should be part of the application since the requirements phase and corrective maintenance should be limited whenever possible.

In order to produce good software we must focus on two aspects: the development process and the software security testing.
Talking about the first one, Microsoft proposed an interesting methology called *Security Development Lifecycle (SDL)* , that provides a standard approach for security development, involving:

- Requirements
- Design
- Implementation
- Verification
- Release
- Response

The *Requirements* phase involves the best security practices, taken from industry standards or solutions adopted on responses to problems faced in the past.
In this phase the functional security requirements to be implemented are defined, and they are similar to those used to define the features of the application, with the only difference that they focus on security.
If the methodology is agile, then the requirements are expressed as user stories; ithese stories concern the security aspects from the user's point of view.

During the *Design* phase, a threat model is defined in order to understand how our system could be attacked.
The threat model is used to understand the attack surface of certain features and the common attacks that the development team must face.
It is also important to define a way to mitigate the problem if certain attacks have success, and it is crucial to identify security issues early on.

The *Implementation* phase is the most difficult one because writing secure code is difficult, and even if you have a dedicated security team, it might

happen that a programmer introduces a security bug in a code currently considered not critical.
It is important to define coding guidelines in a similar way to best practices in order to avoid mistakes by programmers.
It is also important to use tools for static and dynamic application security testing (*SAST* and *DAST*): the first one is used to identify possible vulnerabilities in the source code, while the second checks the application at runtime.

During the *Verification* phase, in addition to the standard tests on the application, we should perform other activities related to security, like:

- Security functional testing
- Vulnerability Scanning and Penetration Testing

Of course, both of them require specific applications and tools.

When the software is finally *released*, it is shipped to customers or made available for download; of course there will be imperfections.

During the final *Response* phase, the customer will report faced problems occurred with the application and ask for updates.

About security testing there is a lot to say because it covers different activities. The following is a brief overview:

- Discovery: this activity does not search flaws in the code but it only focus on identifying the software version in use.
  This is important because sometimes it is possible to discover potential vulnerabilities by only knowing this information.
- Vulnerability scan: this activity is carried out with automated tools that search for known vulnerabilities in the application.
- Penetration test: it is a simulation of an attack, so it tries to exploit flaws in the application like a real attacker.
- Security Review: verifies if security standards have been applied to the system; it is done via analysis of the code and different build of the program.

Security testing can be conducted either in a black box way, where the tester does not know details about the system and its architecture, or in a white box settings where the tester has a deep knowledge of the system and it may have access to the source code.
The testing phase is automatized as much as possible.
Since each application is different from the others, the security testing techniques should be adapted to the target application taken.
Of course, security testing is quite more difficult compared to standard testing (that is, by itself a difficult task), for this reason companies prefer to avoid it or pay third party that are specialized in this particular activity.
Also take in mind that a security testing may not provide you all the vulnerabilities in your system, but only a certain amount; for this reason it is important to take care about your software periodically; when you change a part of your application, you should check that changes do not introduce flaws.

Another security aspect to mention is *privacy*: even if they are two distinct concepts, they are related because without security there is no privacy.
I think it is important to discuss it a bit since recently the *General Data Protection Regulation (GDPR)* has been approved by the European commission in order to strength the protection of personal data of citizens of the European Union.
This new regulation should allow citizens to have more control over their data and it should make sure that companies implement secure enough system to avoid data breach. This is also enforced by Article 25, which requires data protection to be designed into the development of business processes[**?**].
The GDPR also take into consideration the trade of data between companies, which is an hot topic thanks to recent events[**?**].

Even *documentation* and *refactoring* are important while dealing with security; most of the time documentation is neglected because it takes lot of time and is not payed by the customer. Good documentation is necessary to understand the code and avoid the introduction of flaws, especially in big projects. Refactoring is fundamental because it permits to better organize and maintain the code.
For instance, if we keep only one copy of a vulnerable code rather than several ones, we will spare time to fix it.

Another important role is played by *maintenance*, which should make us consider that every aspects in software development, including little details, should be treated with particular care in order to produce secure software.